**EUROPEAN UNIVERSITY OF LEFKE**

FACULTY OF ENGINEERING

**Graduation Project II.**

**Project topic: Chat Application.**

**Name: Chimarokem Divine ORJI.**

**Student Number: 174413.**

My project is a chat application with video call functionality. The name of this app is **helomi**. **Helomi** a platform where users can communicate with one another at real-time no matter how distant they are from each other.

**Supervisor**

Prof. Ezgi Deniz Ülker

**Publish Date**

June 2021

Table of Content

[My Name & Surname---------------------------------------------------------------------------------------i](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759951)

[My Student Number-----------------------------------------------------------------------------------------i](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759952)

[My Supervisor Name---------------------------------------------------------------------------------------------i](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759953)

[Publish Date---------------------------------------------------------------------------------------------------------i](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759954)

[1. Introduction------------------------------------------------------------------------------------------------------1](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759955)

[1.1 Problem definition-----------------------------------------------------------------------------------------1](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759956)

[1.2 Goals---------------------------------------------------------------------------------------------------------1](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759957)

[2. Literature Survey-----------------------------------------------------------------------------------------------2](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759958)

[3. Background Information--------------------------------------------------------------------------------------3](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759959)

[3.1 Required Tools and software and Technologies--------------------------------------------------3](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759960)

[3.2 Other software---------------------------------------------------------------------------------------------3](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759961)

[4. Modules--------------------------------------------------------------------------------------------------------4](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759963)

[4.1 The Back-end------------------------------------------------------------------------------------------4](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759964)

[4.2 The Front-end------------------------------------------------------------------------------------------4](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759966)

[5. Risk Analysis-------------------------------------------------------------------------------------------------5](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759968)

[6. Ethics-----------------------------------------------------------------------------------------------------------6](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759969)

[7. Conclusion-----------------------------------------------------------------------------------------------------7](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759970)

[7.1 Benefits--------------------------------------------------------------------------------------------------7](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759971)

[a. Benefits to users -------------------------------------------------------------------------------------7](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759972)

[b. Benefits to me ---------------------------------------------------------------------------------------- 7](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759973)

* 1. . [Future Works----------------------------------------------------------------------------------------7](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759974)

[8. References ---------------------------------------------------------------------------------------------- 8](file:///C:\Users\DELL\Downloads\templatenew.docx#_Toc54759975)

**1. Introduction**

1.1 Problem Definition

* People can’t communicate except they meet each other physically.
* Two or more people cannot share ideas with one another except they travel and gather together which will much effort, waste of time and inefficiency for a group of company workers.
* In the case of a situation where there is a lockdown because of a local or worldwide disease outbreak like the Covid-19 pandemic, people cannot meet with each other until they wait for the lockdown to cease.

My project will be useful because there are little or no errors. Once a user sends a text message to whomever he wishes, that message gets sent to the exact person. Also, when a user initiates a call to any of his friends, the person being called is notified about the incoming call. There are no flaws in these functionalities.

1.2 Goals

* The project aims at eliminating the problem of distance faced by two users in the course of the need to communicate.
* My aim in this project is to reduce the need of users to gather together before they can communicate and share ideas, thereby saving time and energy.
* The project will make communication efficient since it provides a real time communication (RTC) through a video call.
* In the case of a global lockdown, company workers/groups will be able to work from home, so their company keeps growing.

**2. Literature Survey**

**Helomi** enables different users to chat with another through sending text and also through real-time voice/video communication. Below is a list of differences and similarities between my project and other existing similar applications:

1. **Facebook Messenger**:

Facebook Messenger enables users to exchange text between one each other. Facebook has the friend suggests to a user through machine learning. **Helomi** suggests friends to a user based on a user’s text search. Facebook allows a user’s public posts to be viewed, liked or disliked and commented on by his friends. **Helomi** doesn’t have that functionality.

1. **Whatsapp:**

Whatsapp is allows sending texts, making audio/video call. Whatsapp also notifies a user at real time when there’s a new incoming message. **Helomi** has that functionality also.

**Helomi** does not have that functionality that allows multiple users to chat in a group.

Whatsapp has end-to-end encryption. **Helomi** doesn’t have end-to-end encryption

1. **Zoom:**

Zoom has (mainly) conference call ability but does not store chats. It is specially made for conference call. **Helomi** doesn’t also store chat messages sent and received in a peer-peer-peer call.

**3. Background Information**

3.1 Required Tools, Software and Technologies

Required software:

* + Visual Studio Code:

It’s a good code editor because of its great support for software development.

* + Google Chrome:

To view web pages. It is fast.

* + The command prompt (Window’s shell):

To run all commands including the one for React.js and Node.js that will make the app run locally in development stage.

3.1 Required Tools, Software and Technologies

Other software:

* + Postman:

It’s great for testing requests to the endpoints exposed by back-end code.

* + Figma:

It’s the most modern software for UI/UX design. One can view his design live

* + MongoDB Compass:

This is the default database viewer for the MongoDB database. It is a cool database GUI.

* + Git:

Used for repository

Programming languages / frameworks:

* + HTML/CSS/Sass:

HTML for marking up text. Sass for pre-processing and writing cleaner and structured CSS.

* + JavaScript
  + React.js/React Native:

React is the most popular front-end library/framework for building sweet user interfaces.

* + Node.js/Express.js:

Node.js is powerful and scalable as a back-end programming language. Express.js is a light-weight framework of Node.js. It comes with a server (unlike PHP)

Technologies:

* + Sass:

Sass is used to write CSS efficiently and in a more structured manner.

* + Redux.js:

Redux is a powerful state manager for handling huge global states in an application. There are many states handled in **helomi**.

* + Socket.io:

This allows sending text in real-time. I’ll be using this because all the functionalities in **helomi** involves real time communication

* + Web RTC:

This allows for real-time exchange of audio and video media streams.

* + JSON Web Tokens (JWT):

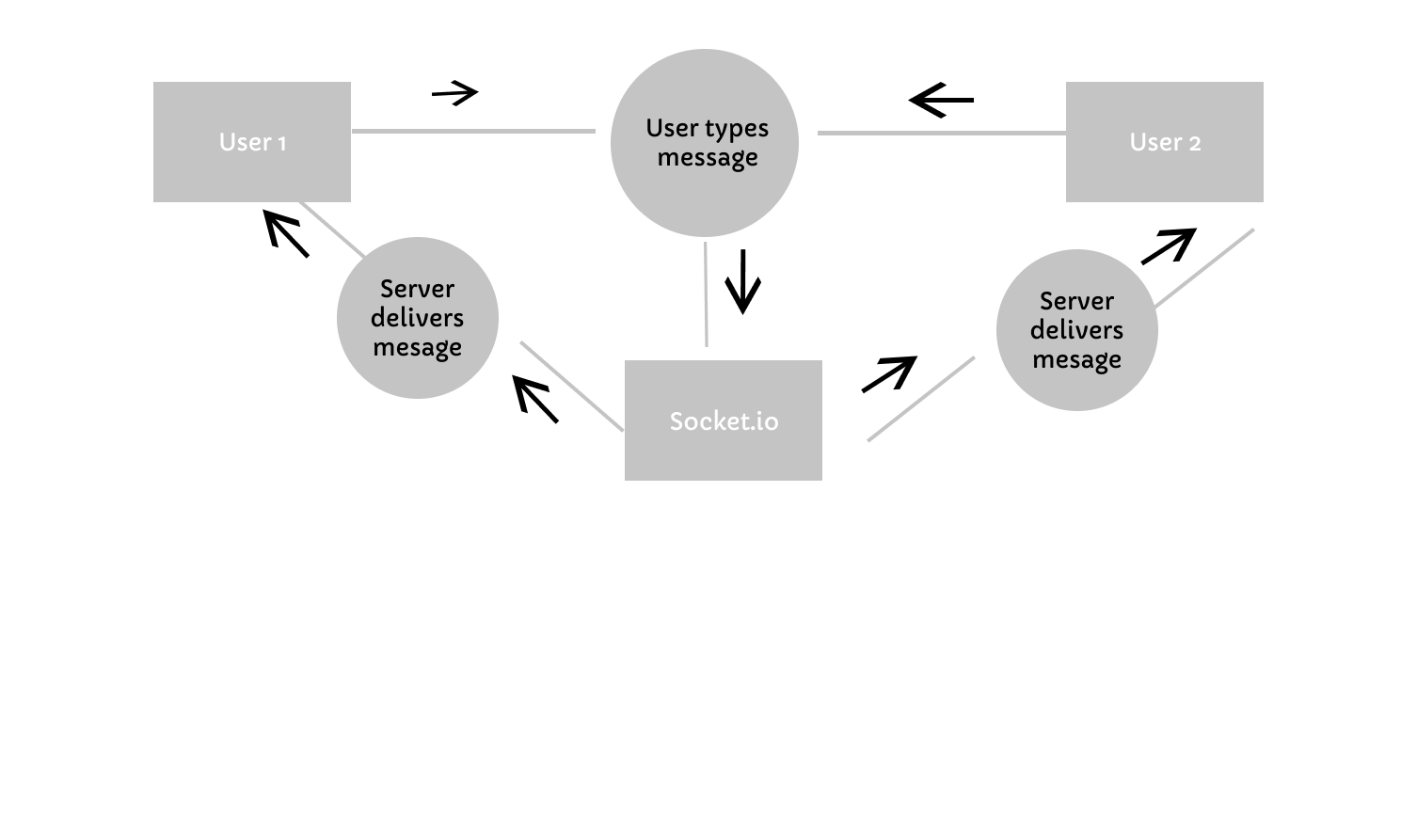
I will be using JWT for authentication. It’s a technology that is used to authenticate and verify a user.

Databases:

* + MongoDB/Mongoose:

It’s a good database. Mongoose allows us to model our schema.

**DFD Diagram Showing the sending and delivery of messages between socket clients and the socket server:**



**5. Methodology**

My project**, helomi** is a full-stack web application, using a front-end JavaScript library (React) and a back-end framework (Node/Express).

Both the frontend and backend use JavaScript overall.

The technology stack used is called the MERN stack. **M** stands for Mongo DB, **E** stands for Express.js, **R** stands for React and **N** stands for Node.js. The MERN stack is used for building powerful and robust web/desktop applications.

**THE BACKEND: chat-application-with-video-audio-call/backend**

Once again, my backend code is written using Node.js/Express.js. The database used to store documents is Mongo DB/Mongoose.

Overall, I used the MVC architecture to structure my whole app. **M** is for Models, **V** is for Views (the frontend – React) and **C** for Controllers.

The server-side app doesn’t render a view. instead, it exposes a REST API when a client hits any of its valid routes.

All my backend is separated into many files but everything concerning the working functionality is all gathered into the **chat-application-with-video-audio-call/backend/app.js**  file. Everything concerned with the server and all in the app.js file is submitted to the server.js.

Explaining the **chat-application-with-video-audio-call/backend/package.json** file

I didn’t create this file. Same thing with th package-lock.json. These files are auto created when a Node project is initiated. This file contains all the dependencies and devDependencies (all the libraries that my app depends on). These dependencies grow as I install more dependencies. The frontend folder also has its own auto-created package.json file.

**The User Model: chat-application-with-video-audio-call/backend/models/User.js**

A new mongoose schema instance called the userSchema is created. Mongoose helps us to model our database documents. The User schema is defined to have the following fields:

* username (which is a unique string field)
* email (string field. It will be lowercased before saving into the DB. It will be validated with a function imported from the ‘validator’ library).
* password: (string field, minimum length is 6, it will not be selected by default when a user’s details are being returned by the server via a response due to security issues).
* friends (will hold an array of Mongo DB ObjectIDs, where each is a reference to some auto-generated \_id of some user).

Before any user document is being created or updated and is about to be saved, if the password field was modified, I hashed (to turn into unreadable stream of strings) the password using a library called bcrypt. This is all done in a ‘pre-save’ middleware function

**The PrivateMsg Model:**

It describes what properties that sent and received messages will have before being saved. It is modelled as follows:

* text (field holding the text content of the message)
* sender (will hold the reference to the \_id of some user who sends the message)
* receiver (will hold the reference to the \_id of some user who the message is intended to be sent to).
* isRead (will hold a Boolean, keeping track of whether the message is already seen/read or not. The default is false)
* createdAt (it holds the exact date any message will be created and saved)

The PrivateMsg collection (or table in SQL terms) is then created out of its schema and then made available for export.

Explaining the **chat-application-with-video-audio-call/backend/config.env** file

This file contains all environment variables that SHOULD not be exposed in code for security reasons. The strings stored here will only be available in the environment when deployed to the web.

Explaining the **chat-application-with-video-audio-call/backend/server.js**

In this file, all the server configurations are set up.

A new http server is set up and is connected to the app.js.

My database is hosted online. So, in the backend, I connected to it via mongoose.connect( ).

Also, a socket.io instance was created and connected to the http server. All socket.io client events/signals emitted/fired in the frontend are caught and handled here by the server.

For example, in handling the ‘private-msg-out’ signal, there’s extra info about the sender, the supposed receiver, the text content and the time the message is being sent.

A new message is being created then saved to the database. And a new signal, ‘new-msg-in’ is fired to the receiver, notifying him of a new incoming message. It is explained in the frontend section of this documentation.

Explaining the **chat-application-with-video-audio-call/backend/app.js** file:

This file is where everything about the app is initiated. The Express library is imported. The ‘cors’ library is imported to enable cors (Cross Origin Resource Sharing), so that any client (frontend) can connect to the backend. All the app.use( ) statements are all middleware. They are executed when a request comes in before any controller is assigned to handle them. They are executed in the order they are written in the code. The app.use(express.json( )) allows the sending of json response to the client.

Some base routes which are the foundation of some other routes are stated here: /users, /friends, /privatemsg.

Explaining the **chat-application-with-video-audio-call/backend/routes/userRouter.js** file:

A new express router is made.

Here, I am saying that the server should map controllers (or route handlers) that will be executed according to the corresponding request. These controllers are executed when these routes are hit through a request by the frontend. Such requests could be either a GET or POST or PATCH request. This already configured express router is then exported and registered in the app.js file.

It is the working principle the other files in the routes folder work.

Explaining the **chat-application-with-video-audio-call/backend/controllers/authController.js** file:

The authController signup( ) function is executed when a POST request is made via the /signup route is hit from the frontend to the backend. In this controller, the new user is created. It is executed when, in the sign up page of the frontend, the user enters his details and creates a new account. The details of this newly created user is then sent back as response. The user is then signed a a JWT token (json web token). This jwt is the center of authentication.

The login() function is executed when the backend route: /login is hit through a POST request. The user submits his credentials. His credentials are then gotten from the req.body object. If his username doesn’t match any user with that username in the database, an error is sent back as a response, and function returns. If it matches, a password match test is done. If no match, an error will be sent as response. Else, he will be given a token that will be required of him whenever he wants to hit any sensitive route.

As soon as a user is signed up or logged in successfully, all his details are being loaded up into the request object via the req.user = user.

The protect( ) function is used as a guard middleware will be executed whenever a sensitive route is hit. Such a route could involve getting private data. Before returning this private data, this protect( ) is run. This function goes to the request header to see if there’s a valid token. If there’s a valid token, we grant the user access to the private data he wanted.

**THE FRONTEND: chat-application-with-video-audio-call/frontend**

React which is the most popular frontend library/framework is a component-oriented frontend library. React is all about creating components. I built complex components from smaller ones.

Each React component renders jsx which is exactly like HTML code. Jsx could also be a mixture of JavaScript logic with HTML code.

I first designed the whole layout using Figma (a web design tool) and then I coded up the mark-up (without any functionality) for the whole layout of the app using pure HTML and Sass (basically CSS). Then I cut portions of this HTML code and created React components out of them. These components include the Navbar component, Login component, the Signup component, the ChatHeader component, the ChatFooter, the MessagesBox, the Chat, the ChattingSection and more others. There are more than 30 components in the frontend. These components are defined each in their own file.

To create an instance of a React project, I ran this on my command prompt:

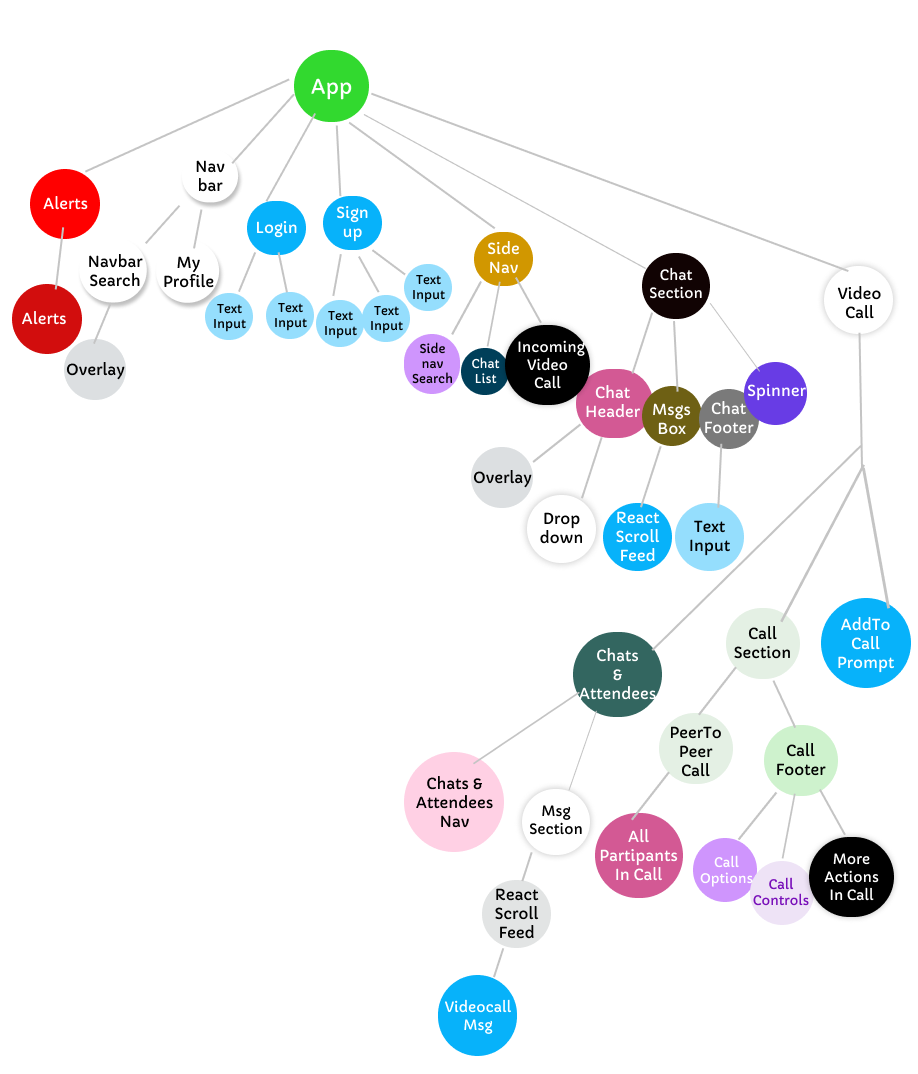
**npx create-react-app chat-application-with-video-audio-call**.

The above command only works if Node.js is already installed.

**React’s Virtual DOM:**

The React virtual DOM mimics JavaScript’s DOM in the browser. JavaScript’s DOM in the browser displays the tree structure of HTML elements which are called DOM nodes. The React virtual DOM shows the tree structure of the components and how they are linked. Which component is a child of which, which is rendered by which. To enable the display of the virtual DOM, the React dev tools have to be installed into Google Chrome.

This virtual DOM also displays local states in each component. Some of these components subscribe to some pieces of the global state which I’ll talk about soon. Below is the tree structure showing the overall structure of the **helomi** app.



**Local and Global States in the App:**

A state is simply a variable which potentially determines if or how some action will be done. A local state in a React component could be created using the predefined useState( ) method which must be initially imported from the React library. This function accepts an argument which is the initial state (Boolean or Number or String or any value of any data type could be passed as an argument). It returns an array of just two items. The first item is a reference to the state which can be accessed anywhere in the component. The second item of this returned array is the reference to an asynchronous function that is used to modify this specific local state. This function can be invoked anywhere in the component. The action of modifying this local state upon user actions (like clicking a button) allows for dynamicity in the app.

Global states are variables that could be needed anywhere in the app. Since they could be used in any component, they are stored outside globally. I am not talking about the global variable principle common in most/every programming language. An example of a global state is the isLoggedIn state that stores a Boolean value if the user is logged in (is authenticated) or not. Most components in the app are to be rendered if and only if the user is logged in. In that case, I’d have to check the value of this isLoggedIn state. If it is false, I would redirect the user to the Login component straight up. I used Redux to handle global states.

**Redux in the App:**

Redux is a light-weight powerful JavaScript library that is an industry-standard tool used to handle global states in an application. Redux could be used with vanilla (pure) JavaScript, it also has its own bindings with React so it can be used with React.

Redux allows components to either access or modify its states in pieces. In the case of modifying, redux has to know which part of the state would be modified. Because of this, reducers receive actions (plain JavaScript objects) as one of their parameters. Each reducer will check for the type/kind (a string) of action and act according to that checked type. This is done by checking the type field of the action object.

In the redux store, there are 5 types of state (categorized according to their concern in the app’s functionality):

* The **alert** state (stores states concerning notifications in the app)
* The **chat** state (stores states concerning friends)
* The **msg** state (stores states concerning messages)
* The **videocall** state (stores states concerning video calling)
* The **user** state (stores states concerning the user)

Explaining **chat-application-with-video-audio-call/frontend/src/redux/store.js**

In my redux store, I imported Redux, createStore, applyMiddleware, composeWithDevTools, thunk, and logger.

The purpose of installing them is:

use Redux in the app,

create/initialize a redux store,

to allow the actions of middleware while components contact the store,

to activate the redux devtools GUI in the browser,

to enable asynchronous dispatch of actions to the store and

to print actions dispatched to the console for debugging reasons respectively.

The rootReducer is also imported. This is the combined pieces of state put together.

Explaining **chat-application-with-video-audio-call/frontend/src/redux/root-reducer.js:**

This file exports a combination of all the 5 reducers/types of state that are imported.

Explaining **chat-application-with-video-audio-call/frontend/src/redux/user/user.reducer.js:**

This file is where I defined the manager/reducer for the redux **user** state.

Explaining **chat-application-with-video-audio-call/frontend/src/redux/user/user.selectors.js:**

In this file, I defined selectors that help to select the whole or pieces of the **user** state.

Explaining **chat-application-with-video-audio-call/frontend/src/redux/user/user.types.js:**

In this file, I defined all the types/kinds/names of action that could be dispatched to the redux store with the intention to modify a piece of the **user** state.

Explaining **chat-application-with-video-audio-call/frontend/src/redux/chat/user.action**.creators:

In this file, I defined and made available for export, functions which will create the different actions that could be dispatched by any React component. Actions could be dispatched asynchronously with the availability of the dispatch function provided by **thunk** which was imported in **store.js.**

Explaining **chat-application-with-video-audio-call/frontend/src/redux/chat/chat.reducer.js:**

This file is where I defined the manager/reducer for the redux **chat** state

Explaining **chat-application-with-video-audio-call/frontend/src/redux/chat/chat.selectors.js:**

Selectors are created here and made available for selecting the whole or pieces of the **chat** state.

Explaining **chat-application-with-video-audio-call/frontend/src/redux/chat/chat.actions.creators.js:**

Same logic but relating to the **chat** state.

Explaining **chat-application-with-video-audio-call/frontend/src/redux/chat/chat.types.js:**

In this file, I defined all the types/kinds/names of action that could be dispatched to the redux store with the intention to modify a piece of the **chat** state.

Explaining **chat-application-with-video-audio-call/frontend/src/redux/chat/chat.utils.js:**

This file contains one helper function that helps to search the user’s chats by a text query.

Explaining **chat-application-with-video-audio-call/frontend/src/redux/msg/msg.reducer.js:**

This file is where I defined the manager/reducer for the redux **msg** state.

Explaining **chat-application-with-video-audio-call/frontend/src/redux/msg/msg.types.js:**

In this file, I defined all the types/kinds/names of action that could be dispatched to the redux store with the intention to modify a piece of the **msg** state.

Explaining **chat-application-with-video-audio-call/frontend/src/redux/msg/msg.actions.creators.js:**

Same logic but relating to the **msg** state.

Explaining **chat-application-with-video-audio-call/frontend/src/redux/msg/msg.selectors.js:**

Selectors are created here and made available for selecting pieces of the **msg** state.

Explaining **chat-application-with-video-audio-call/frontend/src/redux/alert/alert.selectors.js:**

Selectors are created here and made available for selecting the whole or pieces of the **alert** state.

The same idea is applied to the videocall state also.

Explaining **chat-application-with-video-audio-call/frontend/src/App.js:**

App.js is where I combine all other main components that will be rendered or not. It is the root component of the app. Here, I did the routing, and I specified the exact component(s) that will be rendered for each specific route. These ‘main’ components refer to the direct children of the root of the tree diagram above. Most of these main components are rendered if the global isLoggedIn state is true, else I return a Redirect component whose route leads to the login page.

Here, there’s a function called mapStateToProps. This function takes some specific piece of states in the redux store and makes/maps them to become props (properties) of the App component. So we’re mapping state to props. The connect function which is imported from redux is used to connect a React component to the redux store, so that the components can contact the store whenever it needs something from it.

App.scss contains the styling (CSS/Sass) for App.js. It is the styling for the general layout for the page.

Explaining **chat-application-with-video-audio-call/frontend/src/index.js:**

This is where the whole combined App in App.js is rendered in a HTML file **chat-application-with-video-audio-call/frontend/public/index.html.** It is also where the redux store is given provided to the whole app via the Provider component imported from the ‘react-redux’ library. This gives access to the React components to access or modify the store any time, any season.

Explaining **chat-application-with-video-audio-call/frontend/src/pages/Login.js:**

There’s a local state called loginData that holds an object that keeps track of the credentials input by the user in the login form.

When the login button is clicked. Validation takes place. I ran a utility function called getEmptyFields which I defined in the **chat-application-with-video-audio-call/frontend/src/utils.js** that gets the empty fields of an object. Then I passed the loginData as an argument to the function. This function will return an array of the keys of the empty fields.

If this returned array has length > 0, For each element of the array, I immediately dispatch an action of type ‘ADD ALERT’ to reducer to add an alert saying ‘The username field cannot be empty’ or ‘The password field cannot be empty’. After 5s, I dispatch another alert of type ‘REMOVE ALERT’ to remove that specific alert. This adding and disappearing alert action produces a notifying effect to the user.

If the array is empty, I simply communicate with the backend passing the entered credentials. The backend checks if there’s any user in the database whose username and password match the credentials entered.

If a response with a ‘fail’ error is returned, I dispatch an action of type ‘ADD ALERT’ whose alert message is ‘Invalid username or password entered’.

If the backend returns an ‘error’ response, it is possible that there’s a problem with the user’s data connection.

If the response status is ‘success’, the user details (including his friends) is returned from the backend. An action of type SET USER is dispatched to the user reducer to store this returned user object as the current logged in user. Also, the isLoggedIn state is set to true. Another action of type ‘SET CHATS’ is dispatched to tell the chat reducer to set the user’s chats (friends).

Very similar procedure is applied in the Signup component when a user wants to create a new account.

Explaining **chat-application-with-video-audio-call/frontend/src/index.js:**

In index.js, our file is stating where to render our whole app. By default, React renders this whole app inside a <div class=”root”></div> element which is already inside the **chat-application-with-video-audio-call/frontend/public/index.html** file.

Explaining **chat-application-with-video-audio-call/frontend/src/components/layout/navbar/Navbar.jsx:**

This file is where I defined the Navbar component. It renders the navigation bar of the app. This component also consists of other smaller components which are defined in NavbarSearch.jsx and the MyProfile.jsx files. These sub components have separate functionalities on their own and so they’re separated into their own files.

Explaining **chat-application-with-video-audio-call/frontend/src/hooks/useToggle.jsx:**

This file is not a rendering component. Meaning that it doesn’t render any JSX (HTML like code). It is stateful. I made this component in order to reuse the toggling action it returns in any part of my app where toggling would be applied.

This component returns an array of 3 elements. The first is the current state of the thing you’re toggling, the second is a reference to the actual toggle function defined inside of this same useToggle component. The third value returned is a function that resets the state. I’m not sure I’ve used this reset state once. But it could be profitable when I continue to work on the project later.

Explaining **chat-application-with-video-audio-call/frontend/src/components/layout/navbar/Myprofile.jsx:**

This component has a local state called showFullProfile state. Its initial value is false. If this state is true, a dropdown showing the user profile details is rendered. When the user clicks his profile picture, a toggling action is done and so showFullProfile is set false again. If clicked again it is set true again. That’s the toggling action. I imported the useToggle component to implement this toggling action.

There’s a button labelled ‘Log out’. If this button is clicked, then a action of type ‘LOGOUT’ is dispatched telling the user reducer to set the isLoggedIn state to false, and to also reset the user and chat sections of the redux state. The file where the logOutUser function is defined is **chat-application-with-video-audio-call/frontend/src/redux/user /msg.actions.creators.js:**

Explaining **chat-application-with-video-audio-call/frontend/src/components/layout/navbar/NavbarSearch.jsx:**

The NavbarSearch has a search bar that accepts text input. There is a local state that keeps track of what the user types. This state is the searchQuery state. As soon as he focuses on the input with his mouse (and if there’s text in the input) or starts typing, this component communicates with the backend and tells the backend to query for all users whose usernames match the text the user typed in the search bar. This search is accurate. The backend then returns the array of users whose usernames match the input query. If the input query text didn’t match any user’s username at all, an empty array will be returned.

There’s another local state in this component – the searchResults state. It holds this returned array of users as soon as it arrives.

There’s yet another local state called the isShowingSuggestions state that will determine whether this component will render suggestions or not. This state holds the Boolean value of the length of the returned array of users. If the array length of users is empty, then the value of the state is false else it is true and suggestions will be displayed.

This array of users is an array of objects consisting of the user’s username and email. So these users with these details are displayed as suggestions.

A button labelled labeled ‘Add friend’ is also rendered upon rendering each user in the array. If this button is clicked, an addUserAsFriend action is dispatched. This addUserAsFriend( ) function is defined in the chat-application-with-video-audio-call/frontend/src/redux/chat/chat.actions.creators.js/file.

In this function, the id of the user to be added as a friend is passed. There’s a communication with the backend within this function. It tells the backend to add a user whose \_id is the passed user id. This user id is then pushed to the ‘friends’ of the logged in user and resaved (in the database). The backend returns this user just added as a friend as a response.

If the response status from the server/backend is ‘success’, a dispatch action is sent to the chat reducer to add this newly added user to the chat state of the store. Therefore, there’s now a change of state in the store which causes React components that are subscribed to the redux store to re-render. This new friend is being rendered as now a friend of the logged in user.

If the response status from the server is ‘fail’, then the user caused the error. It’s either he tried adding a user that was already his friend or that doesn’t exist or he tried adding himself as his friend. In this case a dispatch action is sent to redux to add an alert whose text content is the error message sent from the server.

There’s an Overlay component that will be rendered if and only if the isShowingSuggestions state is true. This Overlay component returns only an element that spans all through the web page. It can be configured. There’s a prop I called showIf. And I set that value to the isShowingSuggestions state. There is another prop called transparent which is not assigned anything. It gets a value of true automatically. This overlay will not be seen. In some other components, I used a non-transparent overlay. The whole overlay thing is just to hide a dropdown when a user clicks the overlay of it.

Explaining **chat-application-with-video-audio-call/frontend/src/components/Alerts.jsx:**

This component receives ALL the alerts in the redux store as its only prop and renders all of them at the top right hand corner of the page.

It is styled in chat-application-with-video-audio-call/frontend/src/components/Alerts.scss.

Explaining **chat-application-with-video-audio-call/frontend/src/components/layout/Sidenav.jsx:**

The Sidenav component will be rendered first after the user leaves the Login page and is successfully logged in.

So since it is the first component rendered upon successful login, I made the user to join a socket.io room whose name is the user’s unique \_id. This is where I wrote socket.emit('join-self', currentUser.\_id);

This way, every user who successfully logs in will be in his own private room. Socket.io works by delivering data to rooms. The names of these rooms can only be strings.

This component is a parent to three other components which are SidenavSearch, ChatList and the IncomingCallNotify (rendered if the redux videocall state called isRinging is true) components.

The Sidenav subscribes to the redux store via the mapStateToProps.

The Sidenav.scss is the CSS styles for the Sidenav component.

Explaining **chat-application-with-video-audio-call/frontend/src/components/ChatList.jsx:**

After the SET CHAT action is dispatched to the store upon successful login, and the user’s friends array have been set, this ChatList component renders these friends with their details.

It is clear that this store subscribes to the store and selects the chats array in ‘chat’ section of the store via mapStateToProps. This array is made available to the component as a prop. Each friend (chat) is then rendered to the screen. If this chats array is empty, it means that either the array is empty or the backend is still on its way sending a response upon successful login. And so, a LoadingSpinner component which renders a rotatory loader element will be shown.

If the array is not empty, a Chat component is returned for every element in the array and is passed the details of this element.

Explaining **chat-application-with-video-audio-call/frontend/src/components/Chat.jsx:**

This component renders a friend’s username, the last chat message the logged in user had with the friend, shows the notification of new unread messages sent by this friend. This component also shows if this friend is currently typing a message.

If this component is clicked, the route of the app changes to /chats/:id. The id is the \_id of the friend that was clicked. Say this friend’s \_id is 12345678, the current route will be /chats/12345678. Therefore, this friend’s unique \_id is passed to the route which can be referenced later which could be used to do powerful actions.

Explaining **chat-application-with-video-audio-call/frontend/src/components/MessagesBox.jsx:**

This component is intended to contain all messages that the logged in user has ever had with a particular friend. These messages weren’t fetched yet. If I decided to fetch ALL messages a user has had with ALL is friends upon successful login, this action is cost effective. It will have a noticeable effect on performance as the time complexity would be O(n). So if the user has 50 friends, 50 fetch requests will be made instantly. Things would be clearly slow.

So my strategy to fetch ALL messages with a friend when the user clicks on that friend (which is the Chat component already explained). Remember that when this friend is clicked, the friend’s \_id is loaded to the route. This MessagesBox component uses that friend’s \_id in the route greatly.

The msg section of the redux store is structured in a way that each friend’s \_id will hold an array of messages (each messages array are messages that the logged in user has had with the corresponding friend id).

The chat section of the redux store is structured in a way that in each friend object, there is Boolean variable called isLoadingMsgs. This Boolean tells whether we have loaded chat messages with a friend (isLoadingMsgs will be false) or not/still fetching (isLoadingMsgs will be true).

So when this MessagesBox is rendered, we get this friend’s \_id from the route via match.params.id. We then fetch messages from the backend. When this array of messages arrive, we dispatch an action called SET\_CHAT\_MSGS\_LOADING and pass false as an argument to say that messages with this chat (friend) is loaded. So its isLoadingMsgs is now false. The store is then modified.

This way when the user clicks this friend later on, we wouldn’t need to fetch these messages again because the isLoadingMsgs Boolean is now false. For other chats (friends) who haven’t been clicked by the user, their corresponding isLoadingMsgs is still true.

When the messages are fetched, they’re then being stored in the redux store by dispatching an action called FETCH\_CHAT\_MSGS, passing this already fetched messages.

This MessagesBox component connects to the store and for each of these messages, a TextMsg component is created and rendered, passing the message details as a prop (property)

Explaining **chat-application-with-video-audio-call/frontend/src/components/TextMsg.jsx:**

The TextMsg component has a msg prop. This msg prop is an object that is described in the PrivateMsg schema in the database, having 5 fields:

* \_id (auto-generated and unique)
* Text content of the message
* A sender property which holds the \_id of the person who wrote the message
* A receiver field which holds the \_id of the user who the message will be directed to
* A createdAt field which contains the exact date the message was created.

This component subscribes to the redux store and gets only the details of the logged in user which is the currentUser field in the ‘user’ section of the store.

I compared the sender field of this message prop to the currentUser.\_id. If the comparison results to true, then the message is an outgoing one else, it is an incoming one.

I then conditionally applied a HTML class to the message element depending on whether it’s incoming or outgoing.

If incoming, I applied the “textmsg--incoming” class. Otherwise I applied the “textmsg--outgoing" class. Check the definitions of this class in Textmsg.scss.

Now, the createdAt field holds a date field which is a JavaScript Date instance (to get a Date instance, type new Date ( ) in the console). This should not be rendered like that to the user. I then created a util function called getMsgSentTime.

This is where I formatted the date string into a more precise and understandable date. This function is defined in the chat-application-with-video-audio-call/frontend/src/utils.js file.

Explaining **chat-application-with-video-audio-call/frontend/src/ /contexts/SocketProvider.jsx**:

A React context is used to make any data (state) to be accessible (global) over any set of components. A *ReactContext***.**Provider is an already defined component in React that actually does the provision of the global data to other components.

In this file, the data being made available is an instance of a socket.io client. Most incoming events from the server are listened for and handled here. To send events to the server from anywhere in the app, we have to use the same socket instance, not another.

So the strategy is to use a context that will make the same socket.io client instance accessible app wide (global). I could have also stored this in redux since redux stores data globally. I preferred to use context because subscribing to the context requires lesser code and again, what is being stored is not a complex one - just a variable and a function.

In this file, I imported the redux store in order to communicate with the store by dispatching actions directly to it (store.dispatch()) and not through a reducer.

**Incoming Events Handled:**

The “new-msg-in” event:

This incoming event/signal is fired to a user when the server collects a message from the sender and is now delivering it to the user.

When this event is fired, an action whose type is ADD\_NEW\_MSG is created by calling the imported addNewMsg function, passing the message details. When the store receives this action, it adds this new message to the store. Now, React components senses this change in the redux state, and so they re-render automatically (They re-render without the page reloading. That’s the beauty of React generally. That’s what React is made for - re-rendering without reloading). All changes in redux state causes all the React components to reload.

The “user-typing” event:

This incoming event is fired to a user from the server at the point when any of the user’s friends is typing a message. This event also comes with extra info – the isTyping Boolean variable. It is true.

An action of type SET\_SOMEONE\_IS\_TYPING is created by calling the imported setSomeoneIsTyping function, passing the isTyping Boolean and the \_id of that friend that is typing. This action is then dispatched to the store.

The store knows that this action should modify only a portion of the ‘chat’ section of its state. The chat section stores an array of objects, each object has fields describing each of his friends, including where messages with each friend are already fetched or not, whether the user is typing or not, and all the database-defined fields of the User schema.

When a friend stops typing, this event/signal is fired again but this time, coming with the isTyping Boolean as false. The same action is dispatched to the store again but with a false isTyping value. This friend’s isTyping field is then set to false.

The rest of these incoming signals/events will be explained later because their arrival affect extremely different components.

Explaining **chat-application-with-video-audio-call/frontend/src/components/ChatFooter.jsx:**

This component contains a TextInput component where a user creates a new message. It consists of an input and a send icon.

This component has a local state called newMsg. This state keeps track of what the user types.

There’s a useEffect() in this component whose callback function is being executed when the newMsg state changes and it contains a text (in order words, when the user is typing). The callback in the useEffect fires a ‘typing’ signal/event to the server, passing some extra data into it, like the \_id of the friend he’s typing to send a message to, and the logged in user’s \_id. When he presses the send button to send the message, the value of newMsg holding what he typed is sent to the server passing the receiver’s \_id and the logged in user’s \_id. Then, newMsg is then set to ‘’ (nothing). This makes the input box gets cleared.

**The VideoCall Functionality:**

Explaining **chat-application-with-video-audio-call/frontend/src/components/videocall/call-section/PeerToPeer.jsx:**

When the user presses the video call icon in the chat header, a global Boolean state called isOnVideoCall in the videocall section of the global redux state is set to true. And the video call modal scales up into view. And then the PeerToPeer component mounts.

In this component, there are 3 classes. The first class defines a peer connection, called PeerConnection. The second and third are the PeerInitiator and JoiningPeer classes respectively. The last two mentioned classes are child classes of the PeerConnection class and are just used to distinguish between a user who starts a call and another who joins a call.

The functionality of the video call is sponsored by the WebRTC browser API. WebRTC enables browsers to communicate between themselves while exchanging their audio and video streams.

If the user started the call, the makeCall( ) function is called. Else, the joinCall function is called.

If the makeCall( ) is called, a new instance of the PeerInitiator class is created. Its parent class is initialized in the auto-executed constructor function where super( ) is called. Then the user is being asked for permission for the helomi app to use his camera and mic via getUserMedia( ). If this permission is granted, the user’s media stream is gotten and is loaded up into his newly created RTCPeerConnection instance via the addOwnTracks that is unconditionally called in the init( ) function of the PeerConnection class. There’s a React ref, that is made to keep reference to the current user’s video element that is rendered. Once this user’s stream is on, we connect this referenced video element to the stream via connectStreamToVideo( ).

There’s a property in the PeerConnection class called iceConfig. The object contains an array of the stun servers that will be used to search for suitable connection. When the unconnected PeerConnection instance is instanced, the onnegotiationneeded event is fired and the handleNegotiationNeeded function is run to handle that signal. This stun server helps to look for suitable ICE (Internet Connectivity Establishment) candidates that will aid connection to any other peer. Candidates still continually get sent (in my app, it reached about 6 times) until suitable candidates required for connection are gotten.

The initator of the call is made to create a offer via RTCPeerConnection.createOffer( ). This offer is sent to the user being called through a signalling server. We’re using socket.io as our signalling server.

When the user being called is notified of this incoming offer, a ringing alert is shown, giving him the option of declining or answering the call. If he clicks the ‘answer’ button, he’s asked for a permission to use his camera and microphone. Once permission is granted, his stream is connected to the video element already referenced. And his stream is added to his own peer connection (just the same way as explained in the case of the call initiator) He’s then made to create an RTC answer which has be sent to the caller. Also, the candidates that were automatically fetched upon each peer connection have to be exchanged. The two peers have to know each other’s address. Once they exchange their ICE candidates, a connection will be made between them. But they can’t see each other’s video yet.

Remember that both users have loaded their streams to their peer connection. Also remember that these two peers have each other’s ICE candidates exchanged. Now, they both receive an event called a ‘track’ event. This event is fired when their partner’s video/audio stream is loaded to their connection. Since they’ve exchanged each other’s candidates, they now have access to each other’s streams. There’s another React ref called remoteVideoRef. This ref holds reference to a video element where their partner video stream will be played. This is done in the handleTrackEvent.

**5. Risk Analysis:**

* Messages that users send to one another will not be end-to-end encrypted. So, internet providers and other similar third parties will be able to read the messages. And so, a user might see his conversation with someone randomly on the internet.
* Users might rarely use my app because of already existing similar applications like Whatsapp which could have other advanced functionality.
* It’s only about 0.1% possible that connection to the database might not happen instantly since MongoDB is not a real-time database.

**6. Ethics.**

* I’ll try my best to do the project with as much clean and maintainable code as possible, so that in the future, the code will be as easy as possible to be read and easily updated if the desire to update it to a better version arises.
* I engaged in much learning which equipped me with much knowledge of the technologies I’ll be working with.

(Resource is from item 8 at: <https://www.researchgate.net/publication/227991826_Software_Engineering_Ethics>)

* Large ideas in my project were split into sub-sections and files for easy maintainability in the future.
* I set goals of when to achieve different parts of the project in order to finish before submission time.

**7. Conclusion.**

**7.1 Benefits:**

**a. Benefits to users:**

* Users will be able to communicate even when they’re far apart.
* Users will have the regular video call

**b. Benefits to me:**

* I have gained great confidence of being strong with the MERN development stack. MERN is MongoDB + Express.js +React.js + Node.js, with Redux.
* This project can get me a job as a software developer because it is a really large project using about 12 or so technologies.

I chose this project because I was curious to know how real-time messaging works and its implementation with React.js and a industry-standard app state manager, Redux.js.

**7.2 Future Works**

I will continue with the project even after graduation. I desire to make users’ messages end-to-end encrypted later. I couldn’t do the conference call functionality. Video call was only made for two peers only. I’ll add the conference call functionality later on.

**8. References.**

[a] : Whatsapp page. Retrieved from: <https://www.whatsapp.com/features/>

[b] : Wikipedia. Retrieved from: https://en.wikipedia.org/wiki/List\_of\_Facebook\_features

[c] : Wikipedia. Retrieved from: <https://zoom.us/docs/image/features/Welcome.png>

[d]: I followed the Google’s WebRTC documentation (<https://webrtc.org/getting-started/peer-connections>)

[e] I followed the Mozilla WebRTC documentation (https://developer.mozilla.org/en-US/docs/Web/API/WebRTC\_API/)