Technical Documentation for Mindful.BN

A Mental-Health Based Chat Web App

Members:

Abdul Waie bin Hj Md Yussof

Mohamad Afiq Farhan bin A.Rose Ahmadsafri

Jingyung Tan

A documentation created in fulfillment for codingBN's Project

Submission Date:

13th June 2021

Table of Contents

1. Overview – What is Mindful.BN

Currently, there is no mental health-oriented chat application catered to Bruneians. Over the years, mental health issues had also been on the rise in Brunei. Considering that there is a lack of supportive online platform where Bruneians can talk freely about their mental health and daily struggles, Mindful.BN aims to provides an online platform for its users in Brunei to talk with someone safely. Mindful.BN is a chat-based website application focusing on mental health in Brunei. The web app name Mindful.BN is based on the proposed collaboration with a local mental-health community group of the same name consisting of counsellors who will be volunteering their time and attention in listening and talking with the users. Mindful.BN operates with the main goal to provide a safe space for its users in Brunei to talk about their life and hardships while allowing the counsellors in guiding and educating them on emotional well-being practices, on understanding themselves better and on how to approach the problems they have in their life.

2. Technologies Used

Mindful.BN chat web app is created with MongoDB, Express.js, React.js and Node.JS (MERN stack). The dependencies, libraries and frameworks required, in terms of backend and fronted, are:

Backend: MongoDB, Mongoose, Express, Node.js, Socket.io Jsonwebtoken, Moment, Bcrypt, Bson, Cors and Dotenv.

Frontend: React.js, React-router, React-router-dom, Socket.io-client, Axios, Moment, Sweetalert2, Bootstrap, Node-sass and Classnames.

The dependencies are installed via Node Package Manager with the command $ npm i <dependency>.

3. Website Structure

Mindful.BN's website structure, as of the date of submission, consists of five main pages:

1. Home Page – a landing page, a summary of what is Mindful.BN and what it offers
2. Login Page – for handling users sign-in
3. Registration Page – for handling account creation
4. Dashboard Page – for displaying chatrooms, chatroom creation and joining a specific chatroom
5. Chatroom Page – for typing, sending, receiving and displaying messages

4. Home Page

4.1 Frontend – src/index.js

Excluding the CSS and SCSS styling, the frontend for the Home Page required are src/index.js, src/App.js, src/Pages/IndexPg.js and all the .js files in src/components. Shown below is the code for src/index.js.

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";
import "./styles/common.css";
import "./styles/chatroom.css";
import "./index.scss";
import "./components/bootstrap";

ReactDOM.render(<App />, document.getElementById("root"));
```

It imports the required React.js framework, ReactDOM module, the App.js file and Bootstrap.js file. The code of src/index.js file here is used to render the code in src/App.js at the document element with the id of "root" which is found in public/index.html file (code not shown here).

4.2 Frontend – src/App.js

Code snippets to render src/App.js file is shown below which starts with importing React framework, BrowserRouter, Route and Switch from React-router-dom, all of the five main pages of Mindful.BN's website (.js files from src/Pages folder) and socket.io-client.

```
import React from "react";
import { BrowserRouter, Route, Switch } from "react-router-dom";
import ChatroomPg from "./Pages/ChatroomPg";
import DashboardPg from "./Pages/DashboardPg";
import IndexPg from "./Pages/IndexPg";
import LoginPg from "./Pages/LoginPg";
import RegisterPg from "./Pages/RegisterPg";
import io from "socket.io-client";
```

The following App() function below from src/App.js runs after src/index.js calls to render it. The [socket, setSocket] is used with React.useState and passing an initial value of null. The setupSocket function get the token (JSON Web Token) stored in localStorage (stored after user logged-in successfully). If token exists (user had login successfully and have stored token in their localStorage) but socket is false (socket does not exist), establish a newSocket connection with localhost while passing the required query parameter using the token (so that the socket will contain the token information on every connection). On newSocket disconnect, console.log the reason for debugging purposes (if any), setSocket to null (socket state will now become null), run setupSocket function again after 3 seconds (which will continue running until socket does not disconnect). If newSocket do not disconnect, set the socket as newSocket (this will change socket value to the value of newSocket which contains the token information). The subequent React.useEffect() will run setupSocket every time there is a re-render of element or data change in the website page instead of relying on the dependency array [] where the error was disabled with //eslint-disable-next-line.

```
function App() {
    const [socket, setSocket] = React.useState(null);
    const setupSocket = () => {
        const token = localStorage.getItem("CHAT_TOKEN");
        if (token && !socket) {
            const newSocket = io("http://localhost:4040", {
                query: {token: localStorage.getItem("CHAT_TOKEN"),},});
            newSocket.on("disconnect", (reason) => {
                console.log("Disconnect reason: " + reason);
                setSocket(null);
                setTimeout(setupSocket, 3000);
            });
            setSocket(newSocket);
        }
    };
    React.useEffect(() => {
        setupSocket();
        //eslint-disable-next-line
    }, []);
     return (
<BrowserRouter>
            <Switch>
                <Route path="/" component={IndexPg} exact />
                <Route path="/login" render={() => <LoginPg setupSocket={setupSocket} />}
exact />
                <Route path="/register" component={RegisterPg} exact />
                <Route path="/dashboard" render={() => <DashboardPg socket={socket} />} ex
act />
                <Route path="/chatroom/:id" render={() => <ChatroomPg socket={socket} />}
exact />
```

```
            </Switch>
        </BrowserRouter>
    );}
```

After the React.useEffect() for setupSocket() in the code shown above, next is the return() code which returns the value evaluated from the function App() and also, return any React elements to be rendered. Inside the return() code is <BrowserRouter> which is React-router-dom's Router component that allows the use of pushState of HTML5 history API (essentially, allows the pushing the state to re-direct between the 5 main pages and syncing the rendered page UI with the URL). Nested within <BrowserRouter> is <Switch>, which is React-router-dom's component that allows the rendering of the first <Route> that matches the path exclusively. Nested within <Switch> are the React-router-dom's <Route> component which contains the path URL.

```
<Route path="/" component={IndexPg} exact />
<Route path="/login" render={() => <LoginPg setupSocket={setupSocket} />} exact />
<Route path="/register" component={RegisterPg} exact />
<Route path="/dashboard" render={() => <DashboardPg socket={socket} />} exact />
<Route path="/chatroom/:id" render={() => <ChatroomPg socket={socket} />} exact />
```

The first <Route> have the URL path of "/" with the component props passed to it as IndexPg (src/Pages/IndexPg.js). The component props will only be rendered if the <Route> URL path matches with current URL of the browser. The exact is a Boolean to check if the current URL of the browser matches with <Route> URL path *exactly*. If the current URL is "baseURL/" (notice the forward slash followed by nothing afterwards), this matches exactly with the <Route> URL path which will trigger the rendering of src/Pages/ IndexPg.js. The third <Route> have similar code as the first <Route> and its importance is in rendering the Registration Page. The second <Route> renders the Login Page with render function in it (allowing inline rendering) which passes the setupSocket value (found in src/Pages/LoginPg.js) with setupSocket function found here in src/Pages/IndexPg.js, which will execute setupSocket function here when src/Pages/LoginPg.js renders. The fourth and fifth <Route>

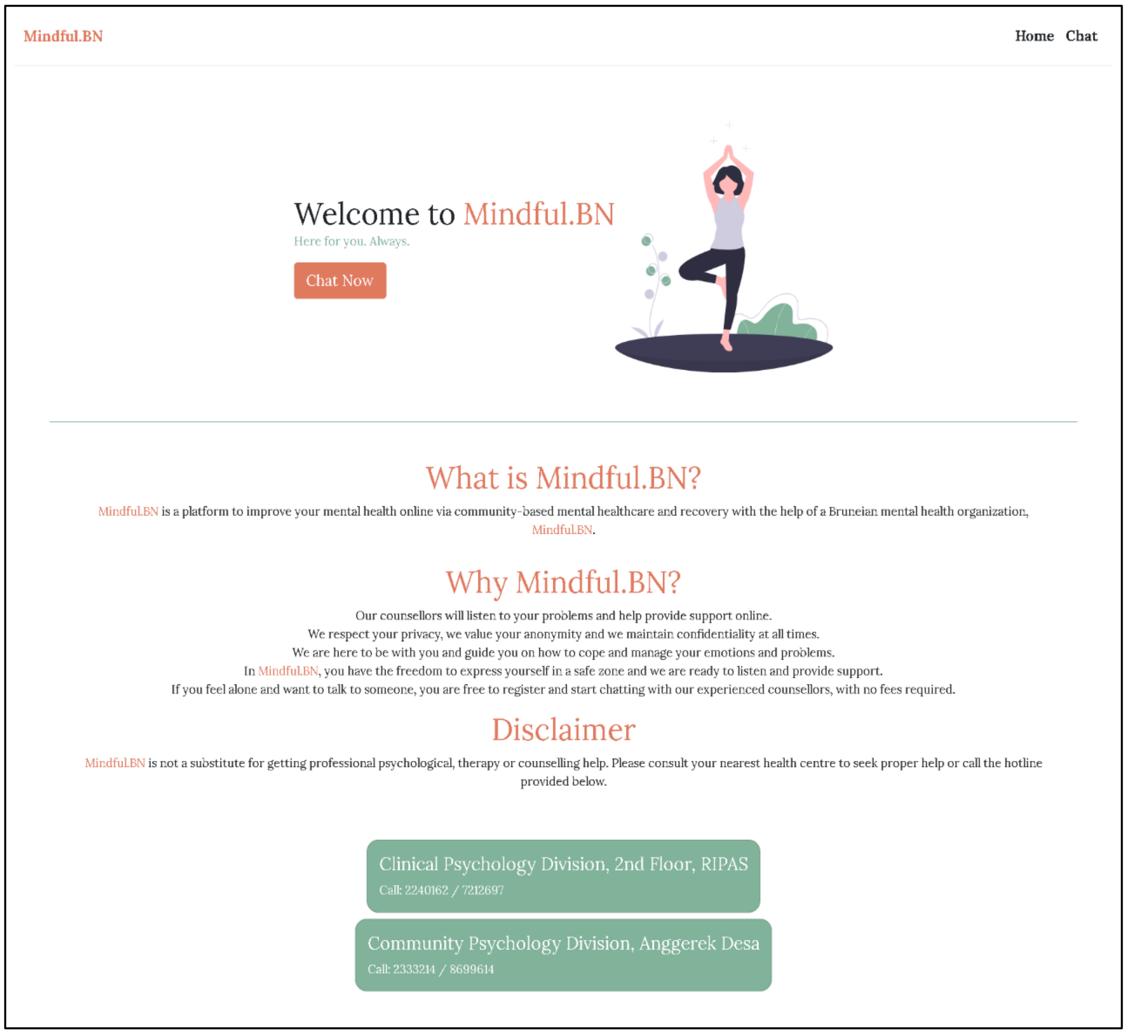
renders the Dashboard Page (src/Pages/DashboardPg.js) and Chatroom Page (src/Pages/ChatroomPg.js) respectively. Each of them get the socket value that was declared with setSocket(newSocket) in src/Pages/IndexPg.js and uses that value in their respective page.

```
export default App;
```

Finally, export the whole App function of src/Pages/App.js so that src/Pages/indexPg.js can use it.

4.3 Frontend – src/Pages/IndexPg.js

Shown below here is the Home Page rendered from the first <Route> that depend on the elements rendered from the code in src/Pages/IndexPg.js.



Welcome to Mindful.BN

Here for you. Always.

What is Mindful.BN?

Mindful.BN is a platform to improve your mental health online via community-based mental healthcare and recovery with the help of a Bruneian mental health organization, Mindful.BN.

Why Mindful.BN?

Our counsellors will listen to your problems and help provide support online.
We respect your privacy, we value your anonymity and we maintain confidentiality at all times.
We are here to be with you and guide you on how to cope and manage your emotions and problems.
In Mindful.BN, you have the freedom to express yourself in a safe zone and we are ready to listen and provide support.
If you feel alone and want to talk to someone, you are free to register and start chatting with our experienced counsellors, with no fees required.

Disclaimer

Mindful.BN is not a substitute for getting professional psychological, therapy or counselling help. Please consult your nearest health centre to seek proper help or call the hotline provided below.

Clinical Psychology Division, 2nd Floor, RIPAS
Call: 2240162 / 7212697

Community Psychology Division, Anggerek Desa
Call: 2333214 / 8699614

Here is the code for src/Pages/IndexPg.js.

```
import React from "react";
import Header from "../components/Header";
import Body from "../components/Body";

const IndexPg = () => {
  return (
    <div className="container-fluid">
      <Header />
      <Body />
    </div>
  );
};

export default IndexPg;
```

It starts with importing React framework, the Header component (src/components/Header.js) and the Body component (src/components/Body.js). The IndexPg.js function is just to render the Header component and Body component based on their respective .js file.

4.4 Frontend – src/components/Header.js

The Header.js renders the navigation bar at the top of the Home Page. It starts with importing React framework and useHistory hook from React-router-dom.

```
import React from "react";
import cx from "classnames";
import { useHistory } from "react-router-dom";
import styles from "./Header.module.scss";
const Header = ({ loggedIn }) => {
  const history = useHistory();
  const logout = () => {
    localStorage.clear();
    history.push("/login");
  };
```

The Header function passes {loggedIn} props and declare the useHistory() hook as history. Within the Header function, there is another function called logout which clears out the localStorage of any key value pairs (localStorage will be used to store unique JSONWebToken to authenticate user access to Dashboard and Chatroom Page), followed by pushing the "/login" URL path to the history stack which will re-direct the browser to Login Page.

Below are some relevant snippets of the return React code to render the navbar elements. The <a> tag with className containing "navbar-brand" is the Mindful.BN text at the top left corner which re-directs the page to the Home Page on click. The <button> with className "navbar-toggler" and data-bs-toggle "collapse" and the <span> with className "navbar-toggler-icon" is responsible for the responsive hamburger icon.

```
return (
<a className="fw-bold navbar-brand" href="#" onClick=
{() => history.push("/")}>Mindful.BN</a>
        <button className="navbar-toggler bg-secondary navbar-dark bg-danger bg-
gradient" type="button" data-bs-toggle="collapse" data-bs-
target="#navbarsExample04" aria-controls="navbarsExample04" aria-expanded="false"
aria-label="Toggle navigation">
          <span className="navbar-toggler-icon"></span>
        </button>
        <div className="collapse navbar-collapse justify-content-end"
id="navbarsExample04">
          <ul className="navbar-nav mb-2 mb-md-0">
            <li><a href="/" className={cx("nav-link px-2 link-
dark", styles.headerLink)}>Home</a></li>
            <li><a href="http://localhost:3000/login" className={cx("nav-link px-
2 link-dark", styles.headerLink)}>Chat</a></li>
            {loggedIn && (<li><a onClick={() => logout()} className={cx("nav-
link px-2 link-dark", styles.headerLink)}>Logout</a></li>)}
          </ul>
        </div>
      </div>
    </nav>
  );
};
export default Header;
```

When the screen viewport size is below 768px (Bootstrap default responsive viewport value), the hamburger icon will appear with toggleable click function that displays the list of clickable <a> tags. Clicking on Home navlink will re-direct the browser to the Home Page whereas clicking on Chat navlink will re-direct the browser to Login Page.

```
            {loggedIn && (<li><a onClick={() => logout()} className= {cx
("nav-link px-2 link-dark", styles.headerLink)}>Logout</a></li>)}</ul>
```

The next code is React's conditional rendering with inline if combined with && operator. It checks for the loggedIn attribute (passed as props in src/components/Header.js) and if there is loggedIn attribute, renders the Logout navlink which will execute the logout function mentioned previously. If there is no loggedIn attribute, render nothing in its place. At the bottom, there is the export default Header which allows src/components/Header.js to be used by importing it somewhere else.

4.5 Frontend – src/components/Body.js

The Body.js simply imports the required modules, .js files components, rendering the texts, SVG and a button. The button is the "Chat Now" button which simply re-directs with <a> tag href attribute to the Login Page so user can enter their account credentials.

4.6 Backend – server.js

The backend for the Home Page requires server.js and app.js only.

Shown below is Home Page code snippets in server.js. The server.js will import dotenv module to allow the use of .env file and the usage of its variable. The server.js will use app.js for Express.js code declared as "app" which will then listen to it via the declared PORT of 4040.

```
require("dotenv").config();

const app = require("./app");
const PORT = 4040;
const server = app.listen(PORT, () => {
  console.log(`Backend Server.js listening on port: ${PORT}`);
});
```

4.7 Backend – app.js

Show below are the code snippets in app.js for Home Page. It will require the Express.js module and declaring it as "app". The app will use express.json() middleware to parse JSON request and responses. It will use express.urlencoded to parse incoming requests with urlencoded payload while extended being true allows rich objects and arrays to be parsed, giving JSON-like response. The app will also require and use the CORS module to allow different origin to share their resource as the backend port (4040) is different than the frontend port (3000, React's default port). At the most bottom of the code (skipping some code not shown in between) , module.exports will export the app so the app can be used elsewhere (e.g. index.js).

```
const express = require("express");
const app = express();

app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(require("cors")());

module.exports = app;
```

5. Login Page

5.1 Backend – app.js

The backend required for Login Page is app.js, handlers/errorHandlers.js, routes/user.js, server.js, all .js files in models folder and controllers/userController.js

The Login Page can be accessed in the browser via the ‘Chat Now’ in Home Page or “Chat’ at the navbar navlink but the routes must be set first by requiring and using routes/user.js in app.js.

```
app.use("/user", require("./routes/user"));
```

The errorHandlers.js will be imported and the Express.js (“app”) will use the imported errorHandlers on handling the notFound page, mongoose errors (mongoDB-related). Currently, the process.env.ENV is in development (development stage), the errorHandlers.developmentErrors will be used instead of errorHandlers.productionErrors. The purpose of errorHandlers.js is to format the errors display accordingly.

```
const errorHandlers = require("./handlers/errorHandlers");

app.use(errorHandlers.notFound);

app.use(errorHandlers.mongooseErrors);
if (process.env.ENV === "DEVELOPMENT") {
  app.use(errorHandlers.developmentErrors);
} else {
  app.use(errorHandlers.productionErrors);
}
```

5.2 Backend – routes/user.js

Shown here is the routes/user.js code. It imports the Router() middleware from Express.js module as router, the exported {catchErrors} from handlers/errorHandlers.js and the exported controllers/userController.js as userController. The router.post handles POST request for the URL path within the quote marks for Login Page and Register Page, followed by running the userController.js exported login and register, respectively, while the catchErrors check and display any errors. The router.post for “baseURL/login” or the router.post for “baseURL/register” will execute only if it’s explicitly called.

```
const router = require("express").Router();
const { catchErrors } = require("../handlers/errorHandlers");
const userController = require("../controllers/userController");

router.post("/login", catchErrors(userController.login));
router.post("/register", catchErrors(userController.register));

module.exports = router;
```

5.3 Backend – server.js

In order to execute router.post for the login, the database (mongoDB) and database Schema (mongoose) must be set-up. Show below is the code for connecting to mongoDB via mongoose.

```
const mongoose = require("mongoose");
const mongoUser = <insert username>;
const mongoPass = <insert password>;
const mongoDatabase = <<insert database name>;
const mongoURI = `mongodb+srv://${mongoUser}:${mongoPass}@cluster0.y1
f9v.mongodb.net/${mongoDatabase}?retryWrites=true&w=majority`;
```

```
mongoose.connect(mongoURI, {useUnifiedTopology: true,
useNewUrlParser: true,});
mongoose.connection.on("error", (err) => {
  console.log("Mongoose Connection ERROR: " + err.message);});
mongoose.connection.once("open", () => {
  console.log("MongoDB Connected!");});
```

It starts with requiring mongoose and setting up the username, password, optional database name and the mongoURI given from mongoDB official website. Then, start to connect to the mongoURI while passing in the new parser and using the unified topology to fix mongoDB deprecation errors. On error, console.log mongoose eerors. Once mongoose connection is open, console.log to indicate successful connection with mongoDB.

After setting-up mongoDB, the following lines import all the required mongoose models to be used in subsequent sections of this documentation.

```
require("./models/User");
require("./models/Chatroom");
require("./models/Message");
```

5.4 Backend – models/user.js

Here is the code for models/user.js. It imports mongoose module and define userSchema as a new mongoose.Schema. Users information must match the specified key value (validated in Register Page) before they can successfully login to the Dashboard Page. The first key is name with the type of string that trims starting and end of whitespace, containing 5 to 20 characters and is required to be entered for successful saving in mongoDB. The email key is a string type with automatic whitespace trimming and is required. Password is a string type with automatic trimming, have a minimum length of 8 and is required. The next key is timestamps with the Boolean of true which automatically include createdAt and updatedAt entry in mongoDB for monitoring and editing in the future. Finally, it is exported as mongoose model, passing the singular mongoDB collection name, followed by what schema it refers to.

```
const mongoose = require("mongoose");
const userSchema = new mongoose.Schema({
    name: {type: String,   trim: true,   minlength: 5,   maxlength: 20,
required: "Name is required!",},
    email: {type: String, trim: true, required:
"Email is required!" ,},
    password: {type: String,                                 trim: true,
minlength: 8,required: "Password is required!",},
  },{timestamps: true,});
module.exports = mongoose.model("User", userSchema);
```

5.5 Backend – controllers/userController.js

The file starts with the importing of mongoose module, the User model as User, bcrypt module and JSON web token module. For Login Page, the exported login function will be executed asynchronously, provided that there are no errors caught. The destructured JSON email and password is defined according to the request.body to pass the value from the input field (frontend) to the backend. The login function will check first if the trimmed email input is not 0 and if it is, it will throw an error to prompt user to correct accordingly. emailRegex is used to ensure email domains is from reputable one by testing the email input against it. If password is empty or less than 8 (to ensure reasonably strong password), it will throw an error. The variable user will be defined using User model and mongoDB’s .findOne (with await) to return the first document that matches the email. This will help to check if user exist or not in mongoDB document and if there is no matching user, it will throw an error. The bcyrpt.compare the hashed password and the input password (with await) and if it does not match, it will throw an error. If it matches, it will sign in a unique JSON web token with the id of the user (information obtained from mongoDB) as the payload to be encoded and combined with PRIVATEKEY (can be found in the .env file). This token will be used for authentication purposes via frontend (LoginPg.js, DashboardPg.js and ChatroomPg.js). Note that the token is stored in the authorization request header. The res.json() will be used for displaying a customized pop-up message for the user to see, sending the icon results and the token as response.

```
exports.login = async (req, res) => {
  const { email, password } = req.body;
   if (email.trim().length === 0) throw "Please enter your email.";
  const emailRegex = /@gmail.com|@outlook.com|@hotmail.com|@live.com|@yahoo.com/;
  if (!emailRegex.test(email)) throw "Email provided is invalid!";
  if (password.trim().length === 0) throw "Please enter your password.";
  if (password.trim().length < 8) throw "Password is invalid!";
  const user = await User.findOne({email});
  if (!user) throw "Ensure the email is typed correctly.";
  const match = await bcrypt.compare(req.body.password, user.password);
  if (!match) throw "Invalid credentials! Please ensure correct credentials are entere
d.";
  if (match) {
    const token = await jwt.sign({ id: user.id }, process.env.PRIVATEKEY);
    res.json({
      message: `User [${user.name}] logged in successfully!`,
      icon: "success",
      token,
    });
  }
};
```

5.6 Frontend – src/Pages/LoginPg.js

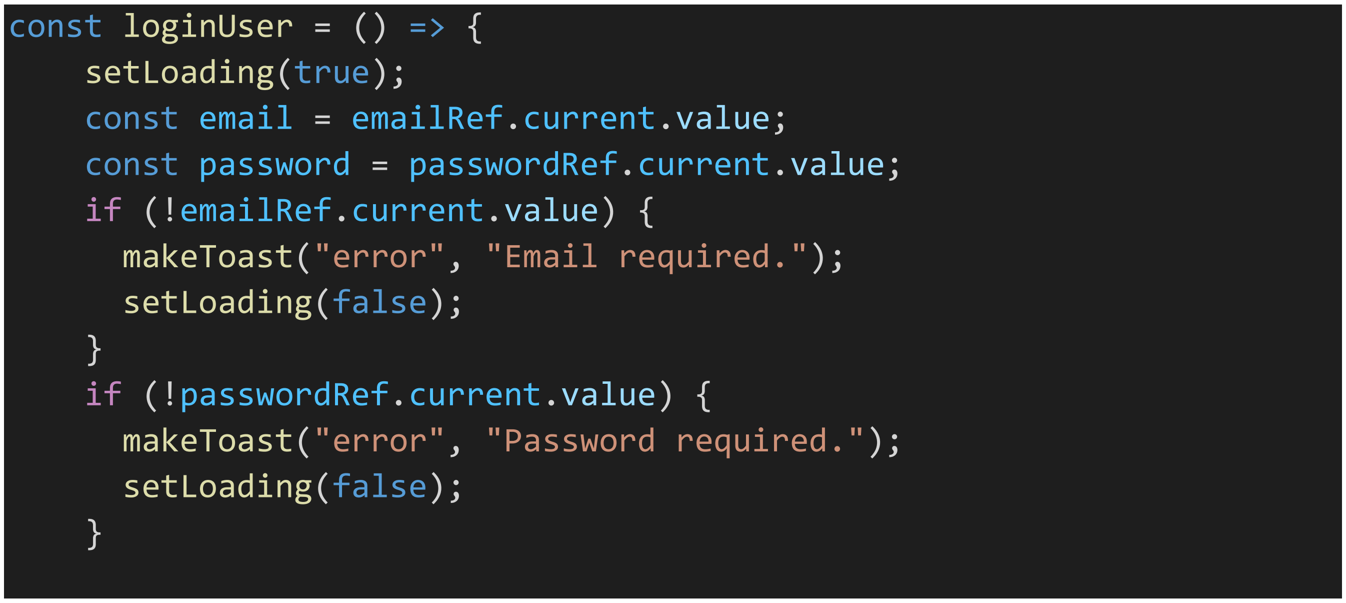
Prior to re-direct to src/Pages/LoginPg.js, src/App.js also passes its setupSocket() so that setupSocket() can be ran in Login Page. The code starts with importing axios module, React & {useState} of React framework, {withRouter} from React-router, {useHistory} from React-router-dom, makeToast function from Toaster.js (sweetAlert2 module), the Header components (navigation bar), Button components (not covered here) and an .svg file.

LoginPg function was passed with props followed by setting React state, createRef (for email and password) and useHistory(). The next React.useEffect function checks for any token stored in localStorage. If there is none, push the page to the Login Page (no noticeable change in browser) and if there is a token, push the page to dashboard, which skips the login process for the user as the token already authenticates that the user can access the dashboard page.

```
const LoginPg = (props) => {
  // set React reference hooks
  const [isLoading, setLoading] = useState(false);
  const emailRef = React.createRef();
  const passwordRef = React.createRef();
  const history = useHistory();

  React.useEffect(() => {
    const token = localStorage.getItem("CHAT_TOKEN");
    if (!token) {
      history.push("/login");
    } else {
      history.push("/dashboard");
    }
    //eslint-disable-next-line
  }, []);
```

Next is the loginUser function (which will be execute onClick of Login button later). The setLoading was set to true to allow the loading animation to occur (need explicit statement of isLoading to run) and the useRef.current.value was assigned a unique variable to each of email and password. If email and password is empty, a customized error pop-up will appear while setLoading is prevented from running.



```
const loginUser = () => {
    setLoading(true);
    const email = emailRef.current.value;
    const password = passwordRef.current.value;
    if (!emailRef.current.value) {
      makeToast("error", "Email required.");
      setLoading(false);
    }
    if (!passwordRef.current.value) {
      makeToast("error", "Password required.");
      setLoading(false);
    }
```

Next, axios POST method will run on the login URL path (based on the fixed URL by appending the URL of app.use in app.js and router.post in routes/user.js) and passing in the useRef values which are the email and password. The Promise response then check if the “success” is in the response body (from controllers/userController.js). If there is, store the JSON web token response value (originates from controllers/userController.js) with the key “CHAT_TOKEN” in the localStorage. Note that the token is previously stored authorization request header and now, it is also stored in the localStorage. Socket will be created using setupSocket() (originates from src/App.js) and push the page to the dashboard. If there is an error, display the error and setLoading to false to prevent the loading animation from running. The createAccount() function is simply an option for user to click to be redirected to register page.

```
axios.post("http://localhost:4040/user/login", { email, password })
.then((response) => {
if (response.data.icon === "success") {
          makeToast(response.data.icon, response.data.message);
          localStorage.setItem("CHAT_TOKEN", response.data.token);
          props.setupSocket();
          props.history.push("/dashboard");
        } else {makeToast(response.data.icon, response.data.message);
          return;}
      }).catch((err) => {
        if (err.response.status === 400) {
          makeToast("warning", err.response.data.message);
        }
        setLoading(false);
      });
const createAccount = () => {props.history.push("/register");};
```

The code below shows the return() code. The <Header /> renders the navbar based on src/components/Header.js. Notice the {emailRef} and {passwordRef} which allows the input to be referenced to so its useRef.current.value can be stored in a variable and accessed for querying and saving into mongoDB. The first <Button> have its isLoading set to isLoading which causes the loading animation to play onClick while at the same time executing the loginUser() function. The second <Button> re-directs the page to the Register Page onClick.

```
return (
    <div className="container-fluid">
      <Header />
      <div className="container d-flex align-items-center justify-
content-around">
        <div className="row">
          <div className="col-sm d-flex align-items-center justify-
content-center p-3">
```

```
            <img
              className="img-fluid"
              src={logo}
              width={300}
              alt="cartoon lady walking"
            />
          </div>
          <div className="col-sm d-flex align-items-center justify-content-center p-3">
            <div className="d-flex flex-column">
              <h1>Login</h1>
              <label htmlFor="email">Email</label>
              <input
                type="email"
                name="email"
                id="email"
                placeholder="example@gmail.com"
                ref={emailRef}
                required
              />
              <label htmlFor="password">Password</label>
              <input
                type="password"
                name="password"
                id="password"
                placeholder="********"
                ref={passwordRef}
                required
              />
              <Button
                isLoading={isLoading}
                className="text-white my-2"
                onClick={loginUser}
              >
                Login
              </Button>
              <Button btnStyle="outline-
secondary" onClick={createAccount}>
                Register
              </Button>
            </div>
          </div>
        </div>
      </div>
    </div>
  );
};

export default withRouter(LoginPg);
```

6. Register Page

6.1 Backend – controllers/userController.js

Register Page have similar code with Login Page but with some differences. The backend for Register Page require app.js, routes/user.js, server.js, handlers/errorHandlers.js and controller/userController.js. The Register Page can be accessed via the Register Button in Login Page. The code required in Register is exactly identical as the previous section detailing Login Page code for app.js (**5.1**), routes/user.js (**5.2**), server.js (**5.3**) and models/user.js (**5.4**).

Register Page depends on the userController.login function exported from controllers/userController.js. It requires the Mongoose module, mongoose model User defined as “User” and bcrypt module only unlike the Login Page’s code. The register function starts asynchronously by passing the req.body (from the frontend’s user input) to the name, email and password in JSON format. Before registration can proceed, the name, email and password must pass the validation. If the trimmed name’s length is 0, less than 5 or more than 20, it will throw some errors to ensure name is within appropriate length between 5 and 20 characters. If the trimmed email’s length is empty or the email domain does not match the emailRegex test, it will throw some errors to ensure some email domain from trusted source have been entered. If the trimmed password’s length is 0 or less than 8, errors will be thrown to ensure password is above 8 characters for security reasons.

To check if user already exists in mongoDB, the variable userExists .findOne in the database for User mongoose model collection, passing the entered email. If userExists return true, it will prompt the user to try a different name or email. If the validation is passed, the user’s password is hashed using bcrypt (with await) while passing the password and the salting rounds of 14 into the variable salting. The salting variable are then referenced to another variable hashedPass. Lowercase user will then be defined using new User mongoose model while passing the name, email and password as the hashedPass, followed by saving the information into mongoDB (with await). Finally, res.json will send response.message and response.icon to be used with makeToast function of sweetAlert2 module to display a customized pop-up to indicate that the user had successfully registered.

```
exports.register = async (req, res) => {
  const { name, email, password } = req.body;

  if (name.trim().length === 0) throw "Please enter your name.";
  if (name.trim().length < 5) throw "Name must be at least 5 characters or more.";
  if (name.trim().length > 20) throw "Name must not exceed 20 characters.";
  if (email.trim().length === 0) throw "Please enter your email.";
  const emailRegex = /@gmail.com|@outlook.com|@hotmail.com|@live.com|@yahoo.com/;
  if (!emailRegex.test(email)) throw "Email provided is invalid!";
  if (password.trim().length === 0) throw "Please enter your password.";
  if (password.trim().length < 8) throw "Password must be at least 8 characters long.";
  const userExists = await User.findOne({ email,});
  if (userExists) throw "Credentials provided already exists. Please choose a different one.";

  const salting = await bcrypt.hash(req.body.password, 14);
  const hashedPass = salting;

  const user = new User({
    name,
    email,
    password: hashedPass,
  });

  await user.save();
  res.json({
    message: "User [" + name + "] with email [" + email + "] register
ed successfully!",
    icon: "success",
  });
};
```

6.2 Fronted – src/Pages/RegisterPg.js

The frontend of Register Page depends only on src/Pages/RegisterPg.js. React, {useState} of React framework, axios module, makeToast of sweetAlert2 module, Header component (navbar), a .svg file were imported initially in the code. The RegisterPg function passed with props have declared useState(false) for isLoading to allow (true) or to forbid (false) loading animation to occur. Empty React.createRef() were declared as nameRef, emailRef and passwordRef. The registerUser function have setLoading(true) to allow loading animations to occur and the current value of nameRef, emailRef and passwordRef are assigned to the variables name, email and password respectively to be used later. If nameRef, emailRef or passwordRef have no current value (empty) on registration submission, makeToast pop-up will appear to prompt the user to enter something in the empty input field while setLoading(false) prevent loading animation from triggering.

```
import React, { useState } from "react";
import axios from "axios";
import makeToast from "../Toaster";
import Header from "../components/Header";
import logo from "../assets/img/meditate.svg";
import Button from "../components/Button";

const RegisterPg = (props) => {
  const [isLoading, setLoading] = useState(false);
  const nameRef = React.createRef();
  const emailRef = React.createRef();
  const passwordRef = React.createRef();

  const registerUser = () => {
    setLoading(true);
    const name = nameRef.current.value;
    const email = emailRef.current.value;
    const password = passwordRef.current.value;

    if (!nameRef.current.value) {
      makeToast("error", "Name required.");
      setLoading(false);
    }
    if (!emailRef.current.value) {
      makeToast("error", "Email required.");
      setLoading(false);
    }
    if (!passwordRef.current.value) {
      makeToast("error", "Password required.");
      setLoading(false);
    }
```

Within the registerUser function, there is an axios POST request (based on the path URL prefix of app.use in backend app.js, followed by the path URL of router.post in backend routes/user.js), passing in the name, email and password. On successful registration, the response will be based on the response.data.message in makeToast pop-up and the page will be pushed to Login Page. On error, a pop-up showing the message will be displayed with the makeToast pop-up and setLoading(false) will prevent loading animation from playing.

```
axios.post("http://localhost:4040/user/register",
{ name, email, password }).then((response) => {
        makeToast(response.data.icon, response.data.message);
        props.history.push("/login");})
      .catch((err) => {
        if (err.response.status === 400) {
          makeToast("warning", err.response.data.message);}
        setLoading(false);
      });
```

Outside of registerUser function but still within RegisterPg function, there is loginUser function which re-directs to Login Page.

```
const loginUser = () => {
    props.history.push("/login");
  };
```

Here is the return() code of RegisterPg.js. onClick of the first <Button>, the registerUser function will execute which validates first via backend controllers/userController.js before saving in mongoDB and the isLoading={isLoading} will trigger the loading animation. The second <Button> will trigger loginUser which re-direct to Login Page if the user wanted to click on it and go to Login Page.

```
return (
    <div className="container-fluid">
      <Header />
      <div className="container d-flex align-items-center justify-
content-around">
        <div className="row">
          <div className="col-sm d-flex align-items-center justify-
content-center p-3">
            <img
              className="img-fluid"
              src={logo}
              width={300}
              alt="cartoon lady walking"
            />
          </div>
          <div className="col-sm d-flex align-items-center justify-
content-center p-3">
            <div className="d-flex flex-column">
              <h1>Register</h1>
              <label htmlFor="name">Username</label>
              <input
                type="text"
                name="name"
                id="name"
                placeholder="Your Username"
                ref={nameRef}
                required
              />
              <label htmlFor="email">Email</label>
              <input
                type="email"
                name="email"
                id="email"
                placeholder="example@gmail.com"
                ref={emailRef}
                required
              />
              <label htmlFor="password">Password</label>
              <input
                type="password"
                name="password"
                id="password"
                placeholder="********"
                ref={passwordRef}
                required
              />
              <Button
                isLoading={isLoading}
                className="text-white my-2"
                onClick={registerUser}
              >
                Sign Up!
              </Button>
              <a onClick={loginUser} className="text-center text-
secondary">
                I already have an account!
              </a>
            </div>
          </div>
        </div>
      </div>
    </div>
  );
};

export default RegisterPg;
```

7. Dashboard Page

7.1 Backend – server.js, models/Chatroom.js & app.js

The Dashboard Page, in terms of backend, requires server.js, app.js, models/Chatroom.js, handlers/ errorHandlers.js, routes/ chatroom.js, middlewares/ auth.js and controllers/ chatroomController.js.

Dashboard Page can be accessed after successful login from the Login Page or if there is a token in the localStorage from previous login sessions which have not been cleared yet via the Logout button.

In server.js, it requires the Chatroom model.

```
require("./models/Chatroom");
```

The chatroom model handles the chatroom creation with name maxlength of 40 characters.

```
const mongoose = require("mongoose");
const chatroomSchema = new mongoose.Schema({
    name: {type: String,           maxlength: 40,           trim: true,
required: "Name is required!",},
});
module.exports = mongoose.model("Chatroom", chatroomSchema);
```

In app.js, it requires the chatroom.js routes.

```
app.use("/chatroom", require("./routes/chatroom"));
```

7.2 Backend – routes/chatroom.js

Shown below is the chatroom.js code in routes folder. It requires Express.js Router(), errorHandlers and chatroomController. Additionally, it also requires the middlewares/auth.js. The first router.get GET the prefixed URL of "/chatroom/" but have to pass the auth.js middleware before it can run .getAllChatrooms function in chatroomController.js to get the list of all chatrooms. The second router.get is also similar but its function is to get the the chatroom header names (for Chatroom Page). Thr third router.post is also similar but it POST a new information into mongoDB to create a new Chatroom Page.

```
const router = require("express").Router();
const { catchErrors } = require("../handlers/errorHandlers");
const chatroomController = require("../controllers/chatroomController
");
const auth = require("../middlewares/auth");

router.get("/", auth, catchErrors(chatroomController.getAllChatrooms)
);
router.get("/chatroomheader", auth, catchErrors(chatroomController.ge
tChatroomHeader));
router.post("/", auth, catchErrors(chatroomController.createChatroom)
);
```

```
module.exports = router;
```

7.3 Backend – middlewares/auth.js

In the middle of router.get or router.post request, it must pass the middlewares/auth.js to be authenticated for it to go to the next code.

The code require JSON web token to run. The function runs asynchronously by checking if there the authorization request headers containers something. If there is none, it will return a 401 error and it stop the code from continuing. This prevents unauthorized users from accessing URL that only authenticated users can access. If the user does have the authorization request header (JSON web token), it will need to be split between the whitespaces and array [1] since the JSON web token is in the format of "Bearer <JSON web token>" and then, it is assigned to the variable token. With await, the jwt.verify will decode token along with the PRIVATEKEY (can be found in .env file) to give the payload (which have the user id value, corresponding to the unique mongoDB user id). The payload is then re-declared as req.payload and the next() middleware function allow the code to move to the next specified function found in routes/chatroom.js.

```
const jwt = require("jsonwebtoken");
module.exports = async (req, res, next) => {
    try {
        if (!req.headers.authorization)
            return res.status(401).json({
                message: "Missing required token!",
                icon: "error",});
        const token = req.headers.authorization.split(" ")[1];
        const payload = await jwt.verify(token, process.env.PRIVATEKEY);
        req.payload = payload;
        next();
    } catch (err) {
        res.status(403).json({
            message: "Invalid token!",icon: "error",});}};
```

7.4 Backend – src/chatroomController.js

After successfully passing middlewares/auth.js, the exported code from src/chatroomController.js can then run. The code require mongoose module, the Chatroom mongoose model as "Chatroom and bson module.

The createChatroom functions runs asynchronously starting with declaring the {name} as the req.body. If the trimmed name's length is zero or more than 40 characters long, it will thrown an error so that user will enter a chatroom name only in an appropriate range. The nameRegex test ensures the first character and the last character, that occur 1 or more times, contains alphanumeric, underscore, non-whitespace characters and if it does, it will throw an error that it contain invalid characters. chatroomExists checks the Chatroom collections with .findOne and passing in name. If chatroomExists is true, it will throw an error to prevent duplicate chatroom names. Once all is validated, new Chatroom model is declared, passing in the name and saving it to mongoDB. The res.json will then provide the message to be displayed in the pop-up of frontend.

```
exports.createChatroom = async (req, res) => {
  const { name } = req.body;

  if (name.trim() === "") throw "Please enter a chatroom name.";
  if (name.trim().length > 40) throw "Please enter a chatroom name co
ntaining 40 characters or less.";
  const nameRegex = /^[\W\S_]+$/;
  if (!nameRegex.test(name)) throw "Chatroom name contain invalid cha
racters.";

  const chatroomExists = await Chatroom.findOne({ name });
  if (chatroomExists) throw "Chatroom with that name have been used! 
Please try another name.";

  const chatroom = new Chatroom({
    name,
  });

  await chatroom.save();

  res.json({
    message: "Chatroom created!",
    icon: "success",
  });
};
```

The getAllChatrooms functions run asynchronously to check for all of the documents in Chatroom mongoDB collection. The response are then displayed in JSON format which will then be used in the frontend to be displayed in the frontend.

```
exports.getAllChatrooms = async (req, res) => {
  const chatrooms = await Chatroom.find({});

  res.json(chatrooms);
};
```

7.5 Frontend – src/Pages/DashboardPg.js

On accessing the Dashboard Page, socket from src/App.js will be used here (see **4.2**). It imports React, {useEffect} of React framework, {Link, useHistory} of React-router-dom module, axios module, makeToast from sweetAlert2 module and moment module. Within DashboardPg function, useHistory(), React.createRef() and [chatrooms, setChatrooms] for React.useState([]) are declared. The following

React.useEffect() simply checks for JSON web token stored in the localStorage. If there is none, push the page to Login Page but if there is a token, push to Dashboard Page (no noticeable change).

```
const DashboardPg = () => {
  const history = useHistory();
  const nameRef = React.createRef();
  const [chatrooms, setChatrooms] = React.useState([]);
  React.useEffect(() => {
    const token = localStorage.getItem("CHAT_TOKEN");
    if (!token) {
      history.push("/login");
    } else {
      history.push("/dashboard");
    }
    //eslint-disable-next-line
  }, []);
```

The getChatrooms function get the list of all chatroom name via axios with “/chatroom” , and headers.authorization of “Bearer ” concatenated with the token in localStorage. Then, the response.data will be set with setChatrooms(response.data), which effectively set the chatrooms value to that within response.data. The setTimeout is for automatic refresh of getting the chatrooms list every 30 seconds. On catching the error, setTimeout for running getChatrooms after 10 seconds. The following useEffect() causes the getChatrooms() to be ran every time there is a re-rendering of elements in Dashboard Page.

```
const getChatrooms = () => {
    axios
      .get("http://localhost:4040/chatroom", {
        headers: {
          Authorization: "Bearer " + localStorage.getItem("CHAT_TOKEN"),
        },
      })
      .then((response) => {
        setChatrooms(response.data);
        setTimeout(getChatrooms, 30000);
      })
      .catch((err) => {
        setTimeout(getChatrooms, 10000);
        console.log(err);
      });
  };
  useEffect(() => {
    getChatrooms();
    //eslint-disable-next-line
  }, []);
```

Another function within DashboardPg function is the createRoom function which create another chatroom and saving it to mongoDB in the backend. It utilizes the useRef.current.value associated with the name (linked to the input box for entering the chatroom name on the frontend). The axios POST will run on the URL path based on the routes set, passing in the name and config (contains the authorization request header for validating the middlewares/auth.js). On receiving the response, run the getChatrooms function, clear out the nameRef input (the input box on the frontend) and display a pop-up to indicate successful creation of chatroom. On error, do the same pop-up to indicate what is wrong.

```
const createRoom = () => {
    const name = nameRef.current.value;
    const config = {
      headers: {
        Authorization: `Bearer ${localStorage.getItem("CHAT_TOKEN")}`,
      },
    };

    axios.post("http://localhost:4040/chatroom/",{name,},config).then((response) => {
        getChatrooms();
        nameRef.current.value = "";
        makeToast(response.data.icon, response.data.message);
      })
      .catch((err) => {
        makeToast(err.response.data.icon, err.response.data.message);
      });
  };
```

Here is the return() code for src/Pages/DashboardPg.js. Here the moment module is used to easily format the Date constructor with very little code. Clicking the <ActionCard> with the label=”Talk” will toggle a modal box with the input field with {nameRef} attribute for chatroom creation and clicking “Create Room” button here will trigger createRoom() function. The chatrooms list (from response.data that is set with setChatrooms) are mapped while passing in the chatroom._id and chatroom.name which can be found in the response.data or chatroom key value pairs. Clicking on the “Join” button will re-direct the page to that specific chatroom.

```
return (
    <div className="container-fluid">
      <Header loggedIn />
      <div className="container">
        <div className="mb-3">
          <h1>
            Hello! Today is{" "}
            <span className="text-
primary">{moment().format("dddd")}! </span>
            The time is{" "}
            <span className="text-
primary">{moment().format("h:mma")}!</span>
          </h1>
          <h2 className="border-primary border-bottom pb-3">
            What would you like to do{" "}
            <span className="text-secondary">today?</span>
          </h2>
        </div>
        <div className="d-flex align-items-center justify-content-
center flex-grow-0 flex-shrink-0">
          <ActionCard
            img={talkingImg}
            label="Talk"
            data-bs-toggle="modal"
            data-bs-target="#exampleModal"
            onClick={() => console.log("You wanted to create a new ch
atroom!")}
          />
          <ActionCard
            img={trackImg}
            label="Track Mood"
            onClick={() =>
              alert("This feature is currently under development.")
            }
          />
        </div>

        <div
          className="modal fade"
          id="exampleModal"
          tabIndex="-1"
          aria-labelledby="exampleModalLabel"
          aria-hidden="true"
        >
          <div className="modal-dialog">
            <div className="modal-content">
              <div className="modal-header">
                <h3 className="modal-title" id="exampleModalLabel">
                  Create new chatroom
                </h3>
              </div>
              <div className="modal-body">
                <input
                  className="nonChat"
                  type="text"
                  name="name"
                  id="name"
                  placeholder="Chatroom name"
                  ref={nameRef}
                />
              </div>
              <div className="modal-footer">
                <Button
                  className="text-white fw-bold"
                  btnStyle="secondary"
                  data-bs-dismiss="modal"
                >
                  Close
                </Button>
                <Button className="text-white fw-
bold" onClick={createRoom}>
                  Create room
                </Button>
              </div>
            </div>
          </div>
        </div>

        <div className="chatroomList_container">
          {chatrooms.map((chatroom) => (
            <div key={chatroom._id} className="chatroomName_Join">
              <div className="chatroom_name">{chatroom.name}</div>
              <Link to={"/chatroom/" + chatroom._id} className="joinL
ink">
                <div className="join">Join</div>
              </Link>
            </div>
          ))}
        </div>
      </div>
    </div>
```

```
  );
};

export default DashboardPg;
```

8. Chatroom Page

8.1 Backend – app.js & controllers/chatroomController.js

The backend required for Chatroom Page are app.js, routes/chatroom.js, middlewares/auth.js, controllers/chatroomController.js, models/Message.js and server.js,

The Chatroom Page can be accessed mainly by clicking the "Join" button in Dashboard Page or alternatively, copy pasting the Chatroom Page browser URL (provided that the localStorage still have the token).

In app.js, the following code is needed for the routing to get the chatroom header name.

```
app.use("/chatroomheader", require("./routes/chatroom"));
```

The code for routes/chatroom.js and middlewares/auth.js are the same. The main differences are the usage for this route shown below.

```
router.get("/chatroomheader", auth, catchErrors(chatroomController.getChatroomHeader));
```

After passing the authentication via middlewares/auth.js, the getChatroomHeader function from controllers/chatroomController.js can then execute. The code are shown below.

```
exports.getChatroomHeader = async (req, res) => {
  const bsonObjectId = new bson.ObjectId(req);
  const chatroomHeader = await Chatroom.findOne({ _id: bsonObjectId });
  res.json(chatroomHeader);
};
```

The code runs asynchronously and passing the req (which is the chatroomId from the frontend) to the bson.ObjectID in order to change the chatroomId to BSON format. This BSON id format will then be used with await in Chatroom.findOne to get the chatroom information from mongoDB. The response will then contain the chatroom name which can then be used on the frontend to be rendered.

8.2 Backend – models/Message.js

This is the code for the message Schema which allows the message to be saved to mongoDB. ChatroomId is the id of the chatroom in mongoDB, chatroomName is the name of the chatroom, userId is the user id in mongoDB, username is the display name of the user, message is the message itself entered by the user, date is the calendar date when message is submitted, time is the time when the message is submitted and timestamps is mongoDB-generated createdAt and updatedAt of the message document itself in mongoDB.

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

const messageSchema = new mongoose.Schema(
  {
    chatroomId: {
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      ref: "Chatroom",
    },
    chatroomName: {
      type: String,
      required: true,
      ref: "Chatroom",
    },
    userId: {
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      ref: "User",
    },
    userName: {
      type: String,
      required: true,
      ref: "User",
    },
    message: {
      type: String,
      required: "Message is required!",
    },
    date: {
      type: String,
      required: true,
    },
    time: {
      type: String,
      required: true,
    },
  },
  {
    timestamps: true,
  }
);

module.exports = mongoose.model("Message", messageSchema);
```

8.3 Backend – Server.js

The server.js is where most of the backend code for handling messages are handled via socket.io module. It requires socket.io module and passing the app.listen(PORT) with CORS passed to it. The CORS is required due to different origin of resource-sharing since backend port origin is 4040 while React's default port is 3000. JSON web token and moment model is required. All of the models are also required.

```
const PORT = 4040;
const server = app.listen(PORT, () => {
  console.log(`Backend Server.js listening on port: ${PORT}`);
});

const io = require("socket.io")(server, {
  cors: {
    origin: "http://localhost:3000",
    methods: ["GET", "POST", "PUT", "DELETE"],
  },
});
const jwt = require("jsonwebtoken");
const moment = require("moment");

const Message = mongoose.model("Message");
const User = mongoose.model("User");
const Chatroom = mongoose.model("Chatroom");
```

The server ("io") will use the function asynchronously to get the token the query params of the socket and then, decoding the id found within using jwt.verify and the PRIVATEKEY found in .env file as the payload. The payload contains the id and are assigned to the socket.userId. Run next() to ensure the next middleware function can run and does not hang. If there is an error, catch the error so the error can be debugged.

```
io.use(async (socket, next) => {
  try {
    const token = socket.handshake.query.token;
    const payload = await jwt.verify(token, process.env.PRIVATEKEY);
    socket.userId = payload.id;

    next();
  } catch (err) {
    console.log(err);
  }
});
```

On establishing connection with the server, asynchronously run the function while passing the socket (socket originates from src/Pages/indexPg.js) to find the user in mongoDB with User model and mongoDB's .findOne method and passing the userId found within the socket. On socket disconnect, find the reason why that particular user disconnect and at what time, formatted with moment. On joining chatroom, find the chatroom id by using Chatroom model and mongoDB's .findOne and passing in the _id as the chatroomId. Socket.join(chatroomId) to establish a socket connection with that particular chatroom. Console.log who join what chatroom and its chatroomId and on what time, formatted with moment. On leaving chatroom, do the same code as on joining chatroom.

```
io.on("connection", async (socket) => {
  const user = await User.findOne({ _id: socket.userId });
  socket.on("disconnect", (reason) => {
    console.log("[server.js]{" + user.name + "} disconnected at " + m
oment().format("h:mm:ssA") + ": " + reason);
  });

  socket.on("joinRoom", async ({ chatroomId }) => {
    const chatroom = await Chatroom.findOne({ _id: chatroomId });
    socket.join(chatroomId);
    console.log("[server.js] " + user.name + " joined chatroom [" + c
hatroom.name + " (id:" + chatroomId + ")] " + moment().format("h:mm:s
sA"));
  });

  socket.on("leaveRoom", async ({ chatroomId }) => {
    const chatroom = await Chatroom.findOne({ _id: chatroomId });
    socket.leave(chatroomId);
    console.log("[server.js] " + user.name + " left chatroom [" + cha
troom.name + " (id:" + chatroomId + ")] " + moment().format("h:mm:ssA
"));
  });
```

On “chatroomMessage” event, pass the chatroomId and message and run the function. If the trimmed message is not empty, find in mongoDB the user Id by querying the _id key in mongoDB with the socket.userId and also, do the same for chatroom. Get the current calendar date and current time, stringifying it and removing their quotes. Declare a new Message model and pass in the required values following Message Schema key value pairs. To the chatroomId, emit the “newMessage” event via the “io” server, passing in the message, name, userId, date and time. Finally, save the newMessage in mongoDB.

```
socket.on("chatroomMessage", async ({ chatroomId, message }) => {

    if (message.trim().length > 0) {
      const user = await User.findOne({ _id: socket.userId });
      const chatroom = await Chatroom.findOne({ _id: chatroomId });

      const currentDate = JSON.stringify(moment().format("DoMMMYYYY")
).replace(/"/g, "");
      const currentTime = JSON.stringify(moment().format("h:mmA")).re
place(/"/g, "");
      const newMessage = new Message({
        chatroomId,
        chatroomName: chatroom.name,
        userId: socket.userId,
        userName: user.name,
        message,
        date: currentDate,
        time: currentTime,
      });

      io.to(chatroomId).emit("newMessage", {
        message,
        name: user.name,
        userId: socket.userId,
        date: currentDate,
        time: currentTime,
      });

      //save the message information to mongoDB
      await newMessage.save();
    }
  });
});
```

8.4 Frontend – src/Pages/ChatroomPg.js

On accessing the ChatroomPage, the socket from src/Pages/indexPg.js will be used here. The imports required are React framework, {withRouter} from React-router module, {useHistory} from React-router-dom module and axios module. Here is the useState array for messages, useState for empty string userId, useState array for chatroom header name, useRef for message input field and useHistory for re-directing the browser page.

```
const [messages, setMessages] = React.useState([]);
const [userId, setUserId] = React.useState("");
const [chatroomHeader, setChatroomHeader] = React.useState([]);
const messageRef = React.useRef();
const history = useHistory();
```

The ChatroomPg function uses match and socket as its props. Match is an object from React-router-dom's <Route> and its params is the key/value pairs parsed from the URL corresponding to the dynamic segments of the path (see the <Route> URL path in section **4.2**). The id within the params query contains the chatroomId, therefore, chatroomId can be declared as match.params.id to obtain the chatroomId. The following React.useEffect() is just to re-direct to Login Page if there is no token (unauthorized access).

```
const ChatroomPg = ({ match, socket }) => {
  const chatroomId = match.params.id;

  const [messages, setMessages] = React.useState([]);
  const [userId, setUserId] = React.useState("");
  const [chatroomHeader, setChatroomHeader] = React.useState([]);
  const messageRef = React.useRef();
  const history = useHistory();
  React.useEffect(() => {
    const token = localStorage.getItem("CHAT_TOKEN");
    if (!token) {
      history.push("/login");
    }
    //eslint-disable-next-line
  }, []);
```

The sendMessage function below passes e (abbreviation for event) which executes e.preventDefault(). This is used to prevent the default refreshing on form submit (this sendMessage is executed on clicking the submit button or on clicking Enter keyboard button). If socket exists and the trimmed messageRef (referenced to the message input field) is empty, set it to empty and return to stop running the rest of the code. If there is socket but messageRef is not empty, then emit the "chatroomMessage" event the chatroomId and the message value inside the message input field. Then, empty the message input box.

```
const sendMessage = (e) => {
    e.preventDefault();

    if (socket) {
      if (messageRef.current.value.trim() === "") {
        messageRef.current.value = "";
        return;
      } else {
        socket.emit("chatroomMessage", {
          chatroomId,
          message: messageRef.current.value,
        });
        console.log(userId + " texted: " + messageRef.current.value);
        messageRef.current.value = "";
      }
    }
  };
```

The following useEffect() function will run whenever there is a re-render in the Chatroom Page. If token is found, split the token by a period "." at array 1, followed by decoding the base64 algorithm with atob function and lastly, using JSON.parse method to convert the JSON object to JSON string to get the payload data. After that, set the userId as the id decoded from the payload. If socket exists, on "newMessage" event (see **8.3**), get the data and pass the data into receiveMessage() function. The receiveMessage() function will then execute, passing in message as argument. Inside the receiveMessage() function, the setMessages useState will run another anonymous function, passing in oldMsgs and using the spread operator to append the oldMsgs, with message (which is the "data" previously). Get the chat container and use .scrollTop based on the changes in scrollHeight (overflow due to additional messages coming in which changes the height of the chat container). This will automatically scroll to bottom to display explicitly the new messages.

```
React.useEffect(() => {
    const token = localStorage.getItem("CHAT_TOKEN");
    if (token) {
      const payload = JSON.parse(atob(token.split(".")[1]));
      setUserId(payload.id);
    if (socket) {
      socket.on("newMessage", (data) => {
        receivedMessage(data);
      });
    }
  }, []);
function receivedMessage(message) {
    setMessages((oldMsgs) => [...oldMsgs, message]);
    var Chat_History = document.getElementsByClassName("Chat_History"
)[0];
    Chat_History.scrollTop = Chat_History.scrollHeight;
  }
```

This useEffect() will render every time there is a change in the Chatroom Page. If there is socket, emit the “joinRoom” event chatroomId. The return function unmounts and check if there is socket, emit the “leaveRoom” event on who leave what room (refer to **8.3**).

```
React.useEffect(() => {
    if (socket) {
      socket.emit("joinRoom", { chatroomId });
    }
    return () => {
      if (socket) {
        socket.emit("leaveRoom", { chatroomId });
      }
    };
    //eslint-disable-next-line
  }, []);
```

The getChatroomHeader function runs via axios GET request, passing in the chatroomId and the authorization request headers. On getting response, declare an empty string name, execute a for loop and check if the response.data[i]._id from the array is the same as the chatroomId available in the frontpage. Afterwards, setChatroomHeaders the empty string name in order for it to be available to be used in the rendering of the chatroom header name. On catching the error, setTimeout to 20 seconds before running the getChatroomHeader function again and also, console.log for debug purposes. Additionally, every time there is a re-render in ChatroomPage, the run the getChatroomHeader() function again with React.useEffect().

```
const getChatroomHeader = async () => {
    await axios
      .get("http://localhost:4040/chatroomHeader", {
        chatroomId,
        headers: {
          authorization: `Bearer ${localStorage.getItem("CHAT_TOKEN")
}`,
        },
      })
      .then((response) => {
        var chatroomHeaderName = "";
        for (let i = 0; i < response.data.length; i++) {
          if (response.data[i]._id === chatroomId) {
            chatroomHeaderName = response.data[i].name;
            console.log(response.data[i].name);
          }
        }
        setChatroomHeader(chatroomHeaderName);
      })
      .catch((err) => {
        setTimeout(getChatroomHeader, 20000);
        console.log(err);
      });
  };
 React.useEffect(() => {
    getChatroomHeader();
    //eslint-disable-next-line
  }, []);
```

Additionally, there is also the leaveChatroom function which will re-redirect the Chatroom Page to the Dashboard Page. This is linked to a Leave Button in the return() code.

```
const leaveChatroom = () => {
    history.push("/dashboard");
  };
```

Here is the return() code for Chatroom Page. The {chatroomHeader} is the results from the for loop mentioned previously. There is also the Leave button will trigger the leaveChatroom() to exit the Chatroom Page of that particular chatroom. Messages (originates from setMessage((oldMsgs) => […oldMsgs, message]) are mapped with another return() code. The "My" and "Partner" in the className allow CSS styling to clearly show which comes from the user and which comes from other people. The message.message, message.time and message.date can be traced back to the each of the message in the messages array where each contains their own message, time and date (see **8.3**). The input field have the ref of {messageRef} which allows its current.value to be accessed and manipulated to be sent to the backend.

```
return (
    <div className="Page">
      <div className="Header">
        <div className="chatroomHeader">{chatroomHeader}</div>
        <button className="leave" onClick={leaveChatroom}>
          Leave
        </button>
      </div>
      <div className="Chat_History ">
        {messages.map((message, index) => {
          return (
            <div
              className={userId === message.userId ? "MyRow" : "PartnerRow"}
              key={index}>
              <div
                className={
                  userId === message.userId ? "MyMessage" : "PartnerMessage"
                }>
                {userId === message.userId && (
                  <span className="MyTimestamp">
                    <span className="My_Date">{message.date}</span>
                    <span className="My_Time">{message.time}</span>
                  </span>
                )}
                <span
                  className={
                    userId === message.userId ? "MyNameTag" : "PartnerNameTag"
                  }>
                  {userId === message.userId ? "you" : message.name}
                </span>
                {message.message}
              </div>
              {userId !== message.userId && (
                <div className="PartnerTimestamp">
                  <span className="Partner_Date">{message.date}</span>
                  <span className="Partner_Time">{message.time}</span>
                </div>
              )}
            </div>
          );
        })}
      </div>
      <form className="Form" onSubmit={sendMessage}>
        <input
          className="ChatBox"
          type="text"
          name="message"
          placeholder="Type your message"
          ref={messageRef} />
        <button className="Button" onClick={sendMessage}>
          Send
        </button>
      </form>
    </div>
  );
};
export default withRouter(ChatroomPg);
```