Metropolia University of Applied Sciences

Social Media Web App REST APIs Implementation

Rana Rayhan

Bachelor's Thesis 2024

Bachelor of Engineering, Information Technology

Major:	Software engineering
Supervisors:	Janne Salonen, Project Manage.

Abstract

Author

Rana Rayhan

Degree Program

Bachelor of Engineering, Information Technology

Thesis Title

Social Media Web App REST APIs Implementation

Number of pages: 47

This thesis delves into a comprehensive examination of the intricate processes involved in developing a Social Media Web App, with a primary emphasis on the backend and REST APIs. The exploration begins with the establishment of a robust backend infrastructure, elucidating essential steps such as Node.js initialization, dependency installation, and Express server configuration. The narrative seamlessly progresses through vital aspects like database connections and the creation of schemas and controllers dedicated to user-centric entities, resulting in a dynamic framework proficient in facilitating CRUD operations.

The central focus of this exploration remains on REST APIs, underscoring their pivotal role in web application development. The integration of WebSocket technology is examined, introducing real-time communication features within the Social Media Web App. The section meticulously details the initialization of Node.js, the setup of a chat system, and the installation of socket.io. Throughout this journey, emphasis is consistently placed on the significance of RESTful API endpoints in facilitating efficient communication between the frontend and backend components.

Collectively, this thesis provides a detailed and organized roadmap, offering profound insights into the practical and theoretical dimensions of constructing a Social Media Web App, with a specific focus on REST APIs and backend intricacies. By navigating through the complexities of backend and real-time communication components, it aims to cultivate a holistic understanding of the nuanced web application development process, all within the realm of RESTful architectural principles.

Keyword: NodeJS, ExpressJS, ReactJS, WebSocket, User Authentication, MongoDB Integration

Table of Contents

1. List of Abbreviations

• **API:**	Application Programming Interface
• **CORS:**	Cross-Origin Resource Sharing (Troy, 2018).
• **JWT:**	JSON Web Token (JWT, 2023).
• **REST:**	Representational State Transfer (Fielding, 2000).
• **npm:**	Node.js Package Manager (npm, 2023).
• **React:**	A JavaScript library for building user interfaces (React, 2022).
• **Node:**	A JavaScript runtime for executing server-side code (Node.js, 2023).
• **MongoDB:**	MongoDB A Non-Relational Database (MongoDB, 2023).
• **Mongoose:**	Mongoose is Object Modelling for Node.js (Mongoose, 2023).
• **bcryptjs:**	A password hashing library for secure storage (bcryptjs, 2017).
• **Schema:**	A blueprint or structure defining the organization of data in a database.

1. Introduction

This thesis embarks on a comprehensive exploration of the intricacies involved in developing a sophisticated Social Media Web App, delving into both backend and frontend realms. The evolving landscape of web application development necessitates a nuanced understanding of fundamental technologies, and this work aims to dissect the process, providing a roadmap for readers to navigate the multifaceted journey.

The journey commences with the foundational steps in backend development, where the initiation of Node.js sets the stage for subsequent advancements **(Node.js, 2023).** The exploration extends to npm **(npm, 2023),** showcasing its primary functions and elucidating the meticulous steps for initializing a Node.js project. Essential aspects like dependency installation are underscored, emphasizing a streamlined approach through a single command.

The thesis meticulously details the creation of a RESTful architecture, offering a visual representation of the file structure designed for REST, (representational state transfer) was introduced by Roy Fielding **(Fielding, 2000).** The intricacies of application initialization and server start, coupled with database configuration and Express.js application setup, lay the groundwork for robust backend infrastructure. Security measures, rate limiting, and error handling mechanisms are meticulously explored, ensuring a resilient and functional backend. Moving into user-centric functionalities, the User Managing Controller section unfolds, covering a spectrum of operations from data retrieval and search functionality to more advanced features like user following and unfollowing. Authentication mechanisms take centre stage, with a thorough examination of user authentication, schema and model definition using Mongoose **(Mongoose, 2023),** and the implementation of JSON Web Token **(JWT, 2023)** for secure communication**.**

The exploration continues into managing social media posts, providing insights into the intricacies of post creation, likes management, and timeline organization. Real-time messaging operations form a pivotal component, unveiling the orchestration of chat systems and messaging functionalities. The final chapters extend into frontend development, where React, Redux Toolkit, and Socket.IO are harnessed to elevate user interaction through seamless state management and real-time communication features.

This thesis endeavours to provide readers with a holistic understanding of the web application development process, bridging the gap between theoretical underpinnings and practical implementation. By dissecting each component and showcasing their interconnectedness, this work aims to empower developers and enthusiasts alike in their quest to construct sophisticated and dynamic Social Media Web Apps.

2. Thesis Structure

The thesis follows a well-organized structure that seamlessly navigates through various components of developing a comprehensive Social Media Web App. Commencing with an overview, the List of Abbreviations sets the stage for a reader-friendly experience. The Introduction provides a glimpse into the multifaceted process of web application development, highlighting the intricate interplay between backend and frontend elements. Thesis Structure further delineates the roadmap, outlining key chapters and their corresponding topics.

The foundational chapters commence with "Initialize Node Package Manager" **(npm, 2023)** shedding light on npm's primary functions and the steps for initiating a Node.js project. Moving forward, "File Structure Diagram for REST APIs" offers a visual representation, enhancing the reader's understanding of the project's organization. "Application Initialization and Server Start" delves into the entry point exploration, emphasizing the critical role of this phase in project initiation.

Chapters 7 to 15 constitute the core of the thesis, focusing on various aspects of backend development, including database configuration, Express.js application setup, user management controllers, authentication mechanisms, social media post management, and real-time messaging operations with Socket.IO **(Chacon, S. & Straub, B., 2014).** Each section is meticulously structured, covering essential topics such as middleware configurations, error handling, and API routing.

The React App creation and integration of Redux Toolkit for state management mark the foray into frontend development. Chapter 15 introduces the Socket.IO server for real-time communication, accentuating the importance of dynamic user engagement through instantaneous updates.

In summary, the Thesis Structure guides the reader through a comprehensive exploration of backend and frontend development, culminating in the integration of real-time communication features. This structured approach ensures a coherent and detailed understanding of the Social Media Web App development process, allowing readers to navigate seamlessly through various dimensions of the project.

4. Initialize Node Package Manager

npm stands for Node Package Manager, and it is the default package manager for the Node.js runtime environment. npm is a command-line tool that facilitates the installation, management, and sharing of code packages, which are essentially reusable pieces of code or libraries. These packages can include everything from libraries, frameworks, and tools to complete applications. **(npm, 2023).**

4.1 The primary functions of npm

Package Installation: npm simplifies the process of installing external libraries or tools needed for a project. It automatically fetches and installs the specified packages from the npm registry.

Dependency Management: npm helps manage project dependencies by keeping track of which packages and versions are required. It generates a package.json file, which serves as a manifest for the project and includes information about dependencies.

Script Execution: npm allows developers to define and execute scripts associated with a project. Common scripts include tasks like testing, building, or starting the application, streamlining the development workflow.

Version Control: npm enables developers to specify and manage the versions of packages used in a project. This helps maintain consistency across different environments and team members.

Publishing Packages: Developers can use npm to publish their own packages, contributing to the vast ecosystem of open-source libraries and tools **(npm, 2023).**

4.2 Steps for Initializing a Node.js Project with npm

Create a Project Folder: Begin by creating a folder for project. For example, let's call it "social-media" **(Node.js, 2023).**

```
mkdir social-media
```

Navigate to the Backend Folder: Enter the newly created project folder and create another folder specifically for the backend. You can name it "backend."

```
cd social-media
mkdir backend
```

Create an index.js File: Within the **"backend"** folder, create an index.js file using your preferred text editor. This file will serve as the entry point for your Node.js application.

```
touch index.js
```

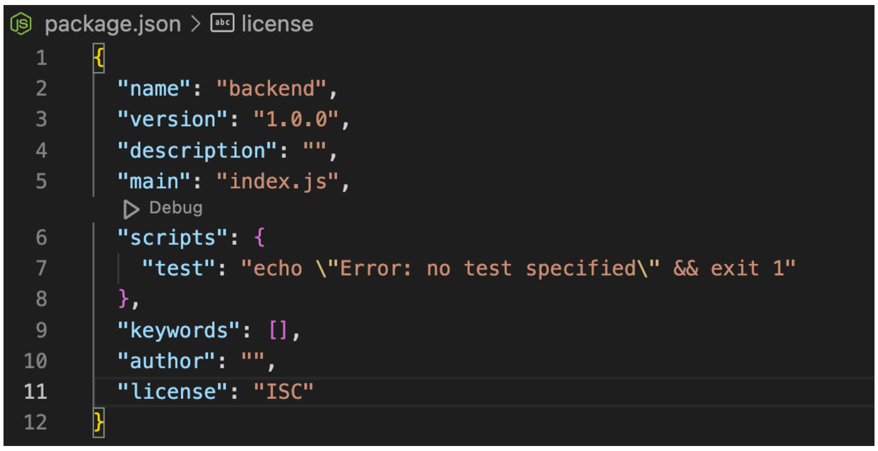
Initialize a Node.js Project: Navigate into the "backend" folder and open a terminal. Use the following command to initialize a new Node.js project. The -y flag auto-generates a package.json file with default values.

```
npm init -y
```

Figure 1: Initialize Node Package Manager (Source Code, 2023)

This command initiates the Node.js project, creating the necessary configuration files and allowing you to manage project dependencies and settings.

At this point, you've set up the foundation for your backend development, complete with an index.js file as the starting point for your server logic and a **package.json** file to manage project dependencies. These initial steps pave the way for building the backend functionalities of your Social Media Web App.



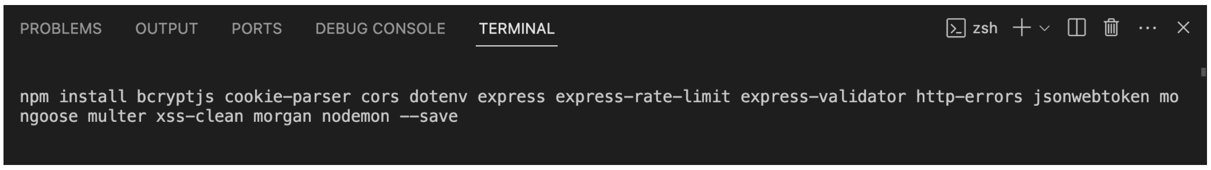
```json
{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

4.3 Installing Dependencies

- **bcryptjs:** Used for password hashing, enhancing security in user authentication.
- **cookie-parser:** Middleware for parsing cookies in HTTP requests.
- **cors:** Cross-Origin Resource Sharing middleware, allowing secure cross-origin requests.
- **dotenv:** Facilitates environment variable management for configuration.
- **express:** Main web application framework for building the backend.
- **express-rate-limit:** Implements request rate limiting for security.
- **express validator:** Middleware for input validation in Express.js.
- **http-errors:** Handles HTTP errors in a structured way.
- **jsonwebtoken:** Implements JSON Web Tokens for user authentication.
- **mongoose:** MongoDB object modelling tool, simplifying interactions with MongoDB.
- **multer:** Middleware for handling multipart/form-data, useful for file uploads.
- **xss-clean:** Middleware for protecting against Cross-Site Scripting (XSS) attacks.
- **morgan:** Development-only HTTP request logger middleware.
- **nodemon:** Development tool for auto-reloading the server during development.

4.3.1 Single command combining all the dependency installations:

npm install bcryptjs cookie-parser cors dotenv express express-rate-limit express-validator http-errors jsonwebtoken mongoose multer xss-clean morgan nodemon --save



```
npm install bcryptjs cookie-parser cors dotenv express express-rate-limit express-validator http-errors jsonwebtoken mo
ngoose multer xss-clean morgan nodemon --save
```

Figure 2: Single command combining all the dependency installations.

5. File Structure Diagram for REST APIs

REST APIs (Representational State Transfer Application Programming Interfaces) adhere to a set of architectural principles designed for networked applications. Emphasizing statelessness, REST APIs require each client-server interaction to be self-contained, with no dependency on prior communication. Following a client-server architecture, these APIs separate concerns, enabling independent evolution of the user interface on the client side and the server-side resource processing. The uniform interface principle simplifies interactions, utilizing unique resource identification through URIs and allowing varied representations, such as JSON or XML. Stateless communication ensures that each request carries sufficient information, fostering a scalable and decoupled system. REST APIs provide a versatile and standardized approach to web service design, facilitating interoperability and scalability in distributed systems. **(Fielding, 2000).**

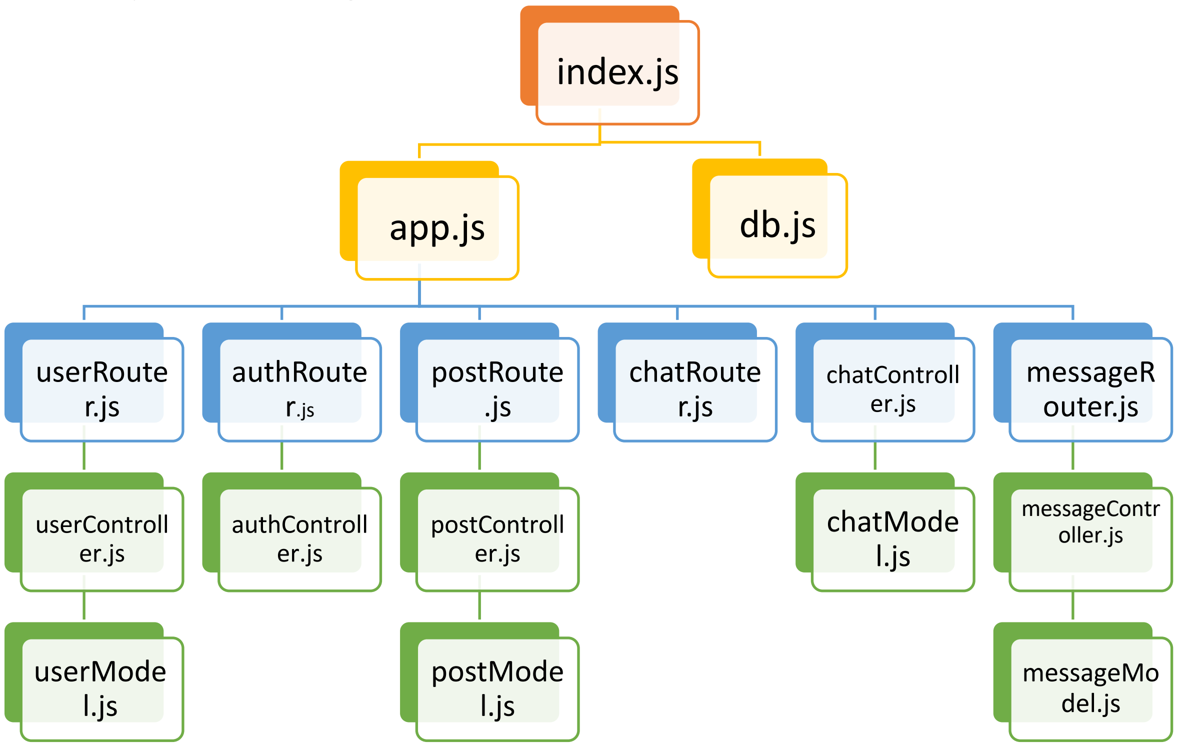


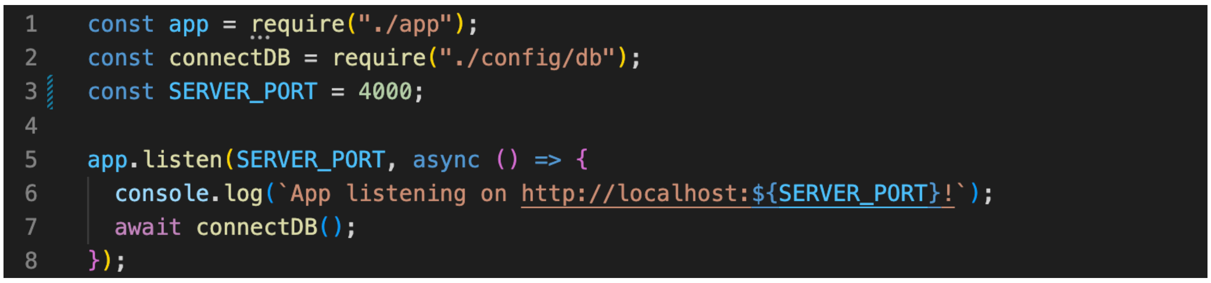
Figure 3: File Structure Diagram for REST APIs

In this project's directory structure, the main entry point is the '**index.js**' file, which likely serves as the starting point for the application. The core logic of the application is organized into modular components, with the **'app.js'** file configuring the overall application settings. The user-related functionality is encapsulated within the **'userRouter.js',** further divided into **'userController.js'** handling user-related business logic and **'userModel.js'** managing the user data. Similarly, authentication features are structured in the **'authRouter.js'** and its associated controllers and models. The application extends its functionality to handle posts through the **'postRouter.js',** with controllers and models ensuring the proper management of post-related data. Additionally, real-time communication functionalities such as chat and messages are handled in a modular fashion through the **'chatRouter.js'** and **'messageRouter.js'** respectively, with corresponding controllers and models for each. This organized directory structure promotes maintainability and scalability by encapsulating related functionality within well-defined modules. **(Express.js, 2022).**

6. Application Initialization and Server Start: Exploring the Entry Point

The **'index.js'** file serves as the entry point for application, initiating its execution. Here, it begins by importing the **'app.js'** module, presumably containing the main configuration and setup for your web server. The **'connectDB'** function from the **'./config/db'** file is then invoked to establish a connection to the database. The application is set to listen on port 4000 with the **'app.listen'** method, and once the server is successfully started, a message is logged to the console, indicating that the application is now listening **on http://localhost:4000.** This file essentially orchestrates the start-up process, ensuring that the server is initialized, the database connection is established, and the application is ready to handle incoming requests. **(Express.js, 2022).** The source code for the application can be found here:

https://github.com/rana-rayhan/socialMedia-mern-p9/tree/main/server

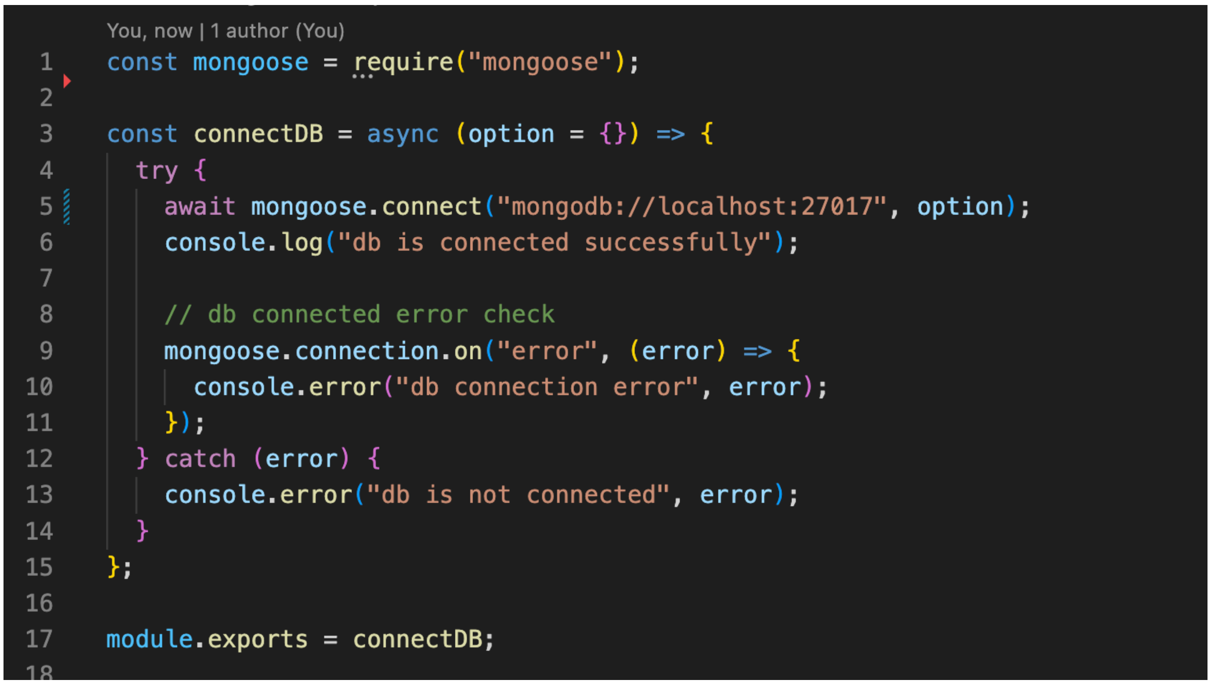


```
const app = require("./app");
const connectDB = require("./config/db");
const SERVER_PORT = 4000;

app.listen(SERVER_PORT, async () => {
  console.log(`App listening on http://localhost:${SERVER_PORT}!`);
  await connectDB();
});
```

Figure 4: An example of index.js file can be seen in this picture (Source Code, 2023)

7. Database Configuration

The **'db.js'** file encapsulates the database connection logic for Node.js application using the Mongoose library. It exports a function named **'connectDB',** which establishes a connection to the MongoDB database specified by the local URL provided as an argument. The function includes error handling to log successful connections and, in case of errors, prints messages to the console indicating whether the connection was successful or unsuccessful. This modular approach allows for easy integration of the database connection functionality across various parts of your application, promoting maintainability and reusability of the database-related code. **(Mongoose, 2023).**



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

const connectDB = async (option = {}) => {
  try {
    await mongoose.connect("mongodb://localhost:27017", option);
    console.log("db is connected successfully");

    // db connected error check
    mongoose.connection.on("error", (error) => {
      console.error("db connection error", error);
    });
  } catch (error) {
    console.error("db is not connected", error);
  }
};

module.exports = connectDB;
```

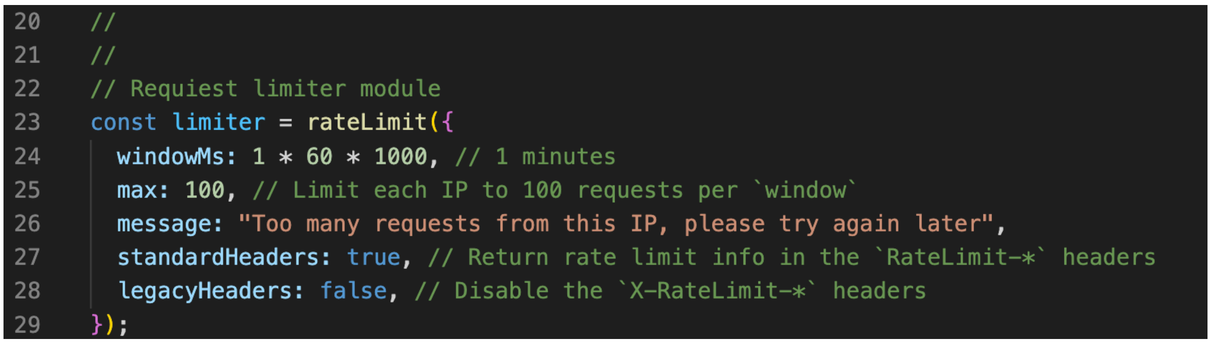
Figure 5: An example of db.js file can be seen in this picture (Source Code, 2023)

8. Express.js Application Configuration:

The **'app.js'** file orchestrates the configuration and setup of your Express.js application **(Express.js, 2022),** serving as the central hub for middleware integration, route handling, and error management. It initializes key middleware components, including cookie parsing, CORS support, request rate limiting, protection against cross-site scripting (XSS) attacks, and logging using the Morgan module. The file then defines and mounts routers for various application features, such as user authentication, post management, and real-time chat functionalities. Additionally, it includes error handling mechanisms for both client-side (404 Not Found) and server-side errors, ensuring a robust and standardized response format. This modular and organized approach in 'app.js' enhances the maintainability and extensibility of your application by clearly delineating middleware, routes, and error-handling components**.**

8.1 Implementing IP-Based Rate Limiting for Request Throttling using.

The provided code configures a rate limiter middleware in a Node.js application using the 'express-rate-limit' package **(Express rate limit, 2023)**. This middleware limits the number of requests a client can make within a one-minute window to 100, based on their IP address. If a client exceeds this limit, the server responds with a message stating, "Too many requests from this IP, please try again later." Additionally, the configuration includes options to return rate limit information in the headers ('standardHeaders: true') and to disable legacy headers ('legacyHeaders: false'). This implementation helps prevent abuse or excessive usage by imposing a reasonable constraint on request frequency from a single IP address, promoting fair and controlled access to the server resources.

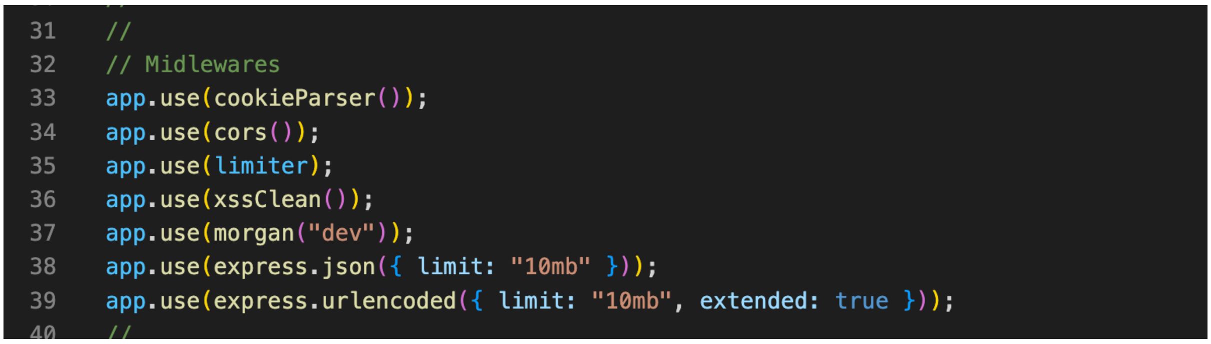


```
//
//
// Requiest limiter module
const limiter = rateLimit({
  windowMs: 1 * 60 * 1000, // 1 minutes
  max: 100, // Limit each IP to 100 requests per `window`
  message: "Too many requests from this IP, please try again later",
  standardHeaders: true, // Return rate limit info in the `RateLimit-*` headers
  legacyHeaders: false, // Disable the `X-RateLimit-*` headers
});
```

Figure 6: An example of express-rate-limit can be seen in this picture (Source Code, 2023)

8.2 Application Security and Functionality with Essential Middleware Configurations

The provided code snippet configures several middleware functions in a Node.js application using Express. Each middleware serves a specific purpose in enhancing the functionality, security, or logging of the application.



```
//
// Midlewares
app.use(cookieParser());
app.use(cors());
app.use(limiter);
app.use(xssClean());
app.use(morgan("dev"));
app.use(express.json({ limit: "10mb" }));
app.use(express.urlencoded({ limit: "10mb", extended: true }));
//
```

Figure 7: An example of Application Security, Functionality and Middleware (Source Code, 2023)

Cookie Parser: This middleware parses incoming cookies and makes them available in the 'req.cookies' object. It is essential for handling and managing data stored in cookies, providing session management and user authentication. **(Express.js, 2022).**

CORS: Cross-Origin Resource Sharing (CORS) middleware. It enables secure cross-origin data requests between the client and the server, ensuring that the server explicitly allows such requests, preventing potential security issues related to cross-origin requests. **(Troy, 2018).**

Limiter: This middleware, as explained in a previous response, implements rate limiting to control the number of requests from a single IP address within a specified time window. It helps prevent abuse, ensures fair usage, and protects the server from potential denial-of-service (DoS) attacks. **(Express rate limit, 2023).**

Xss Clean: This middleware helps protect against Cross-Site Scripting (XSS) attacks.It sanitizes user input by escaping or removing potentially malicious characters, reducing the risk of injecting malicious scripts into the application. **(xss clean, 2023).**

Morgan: Morgan is a popular HTTP request logger middleware. Setting it to "dev" format logs concise, development-friendly information about each incoming request, including request method, status code, and response time. This aids in debugging and monitoring during development. **(Morgan, 2020).**

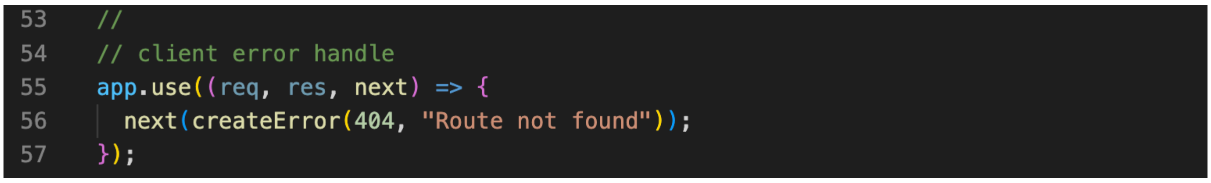
express.json({ limit: "10mb" }): This middleware parses incoming JSON payloads in the request body. The 'limit' option sets the maximum size of the JSON payload, helping to prevent potential abuse or errors related to excessively large request bodies. **(Express.js, 2022).**

express.urlencoded({ limit: "10mb", extended: true }): Similar to 'express.json()', this middleware parses incoming URL-encoded form data. The 'limit' option sets the maximum size of the URL-encoded form data, and 'extended: true' allows parsing of rich objects and arrays in the form data. **(Express.js, 2022).**

In summary, these middleware functions collectively contribute to enhancing the security, functionality, and logging capabilities of the Node.js application **(Node.js, 2023)**, addressing common concerns such as request size limits, cross-origin resource sharing, security vulnerabilities, and request logging during development.

8.3 Client Error Handling Middleware

This middleware handles client-side errors by using createError to generate a 404 (Not Found) error with a custom message, indicating that the requested route is not found.



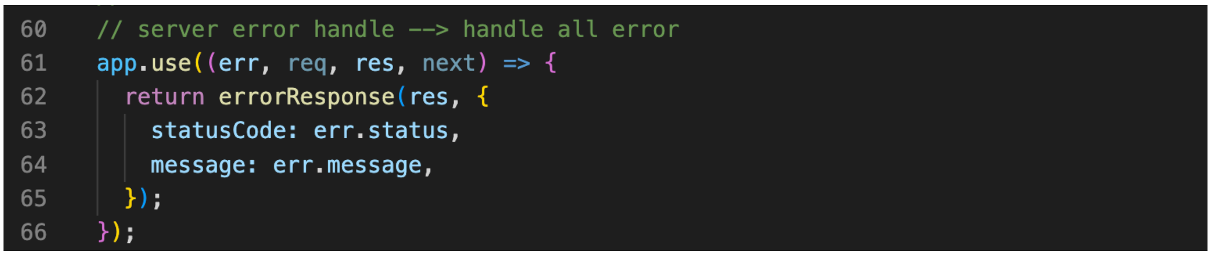
```javascript
//
// client error handle
app.use((req, res, next) => {
  next(createError(404, "Route not found"));
});
```

Figure 8: An example of client error handling middleware (Source Code, 2023)

The next function is then called to pass the error to the next middleware in the stack. This ensures that the error is propagated to the subsequent error-handling middleware. **(Express.js, 2022).**

8.4 Server Error Handling Middleware

This middleware is designed to handle server-side errors. It is executed when an error is passed to it, either from the client error handling middleware or from any other part of the application. **(Express.js, 2022).**



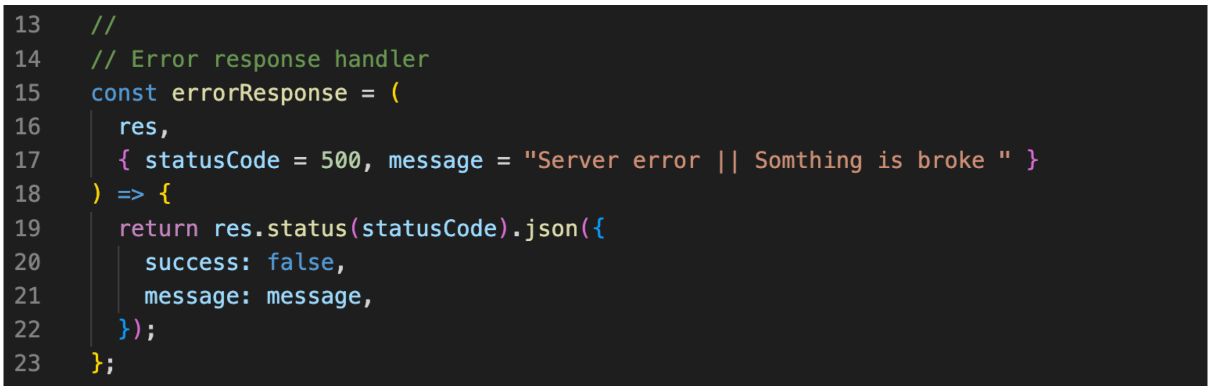
```javascript
// server error handle --> handle all error
app.use((err, req, res, next) => {
  return errorResponse(res, {
    statusCode: err.status,
    message: err.message,
  });
});
```

Figure 9: An example of server error handling middleware (Source Code, 2023)

The error object (err) contains information such as the status code and message. In this case, it uses the errorResponse function to send a structured error response to the client, including the status code and a descriptive message.

8.5 Error response Handling controller

This middleware handles client-side errors by using **'createError'** to generate a 404 (Not Found) error with a custom message, indicating that the requested route is not found. The 'next' function is then called to pass the error to the next middleware in the stack. This ensures that the error is propagated to the subsequent error-handling middleware.



```
//
// Error response handler
const errorResponse = (
  res,
  { statusCode = 500, message = "Server error || Somthing is broke " }
) => {
  return res.status(statusCode).json({
    success: false,
    message: message,
  });
};
```

Figure 10: An example of error response controller (Source Code, 2023)

This middleware is designed to handle server-side errors. It is executed when an error is passed to it, either from the client error handling middleware or from any other part of the application.

The error object ('err') contains information such as the status code and message. In this case, it uses the 'errorResponse' function to send a structured error response to the client, including the status code and a descriptive message.

Error handling middleware is crucial in an application to gracefully manage and respond to errors, enhancing user experience and aiding in debugging during development.

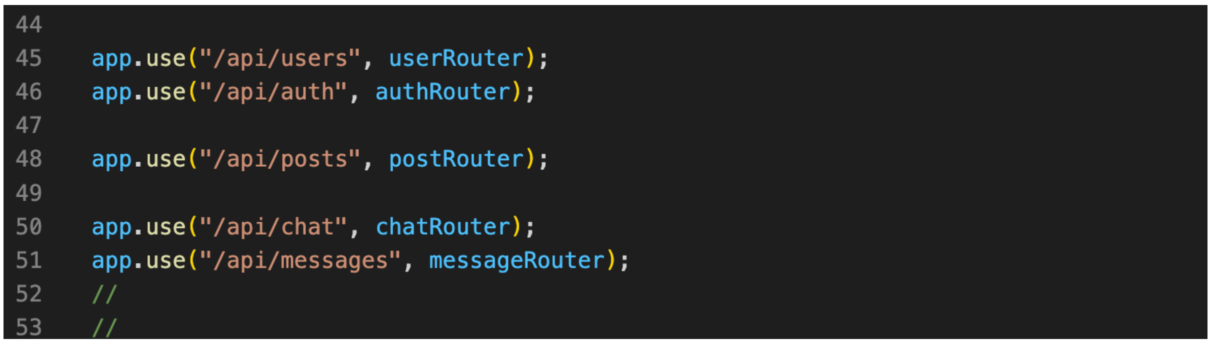
The client error handling middleware specifically addresses cases where a requested route is not found, returning a 404-status and a meaningful message to the client.

The server error handling middleware is a catch-all for handling any other errors that may occur during the application's execution. It ensures that all errors are consistently formatted and communicated to the client, providing valuable information for debugging while also maintaining a professional and user-friendly error response.

Overall, these error-handling middleware components contribute to the robustness and reliability of the application by ensuring that both client and server errors are appropriately handled and communicated to the users or developers.

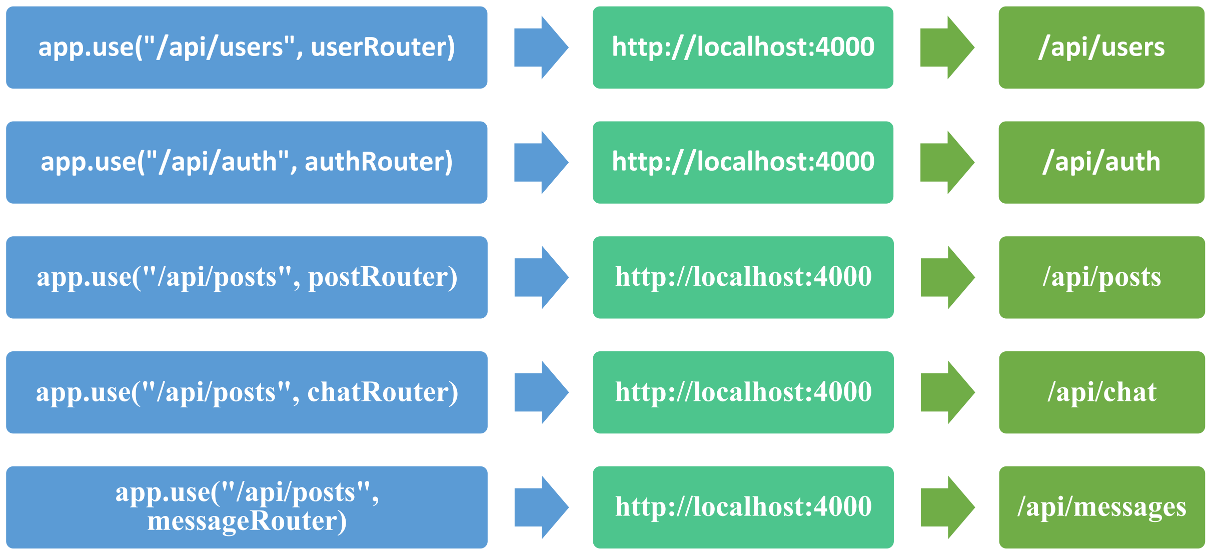
8.6 Structured API Routing for Application

The provided code sets up routing for different features in a Node.js application using Express **(Express.js, 2022).** Each app.use statement associates a specific route prefix with its corresponding router.



```

app.use("/api/users", userRouter);
app.use("/api/auth", authRouter);

app.use("/api/posts", postRouter);

app.use("/api/chat", chatRouter);
app.use("/api/messages", messageRouter);
//
//
```

Figure 11: An example of Structured API Routing (Source Code, 2023)


app.use("/api/users", userRouter)
http://localhost:4000
/api/users
app.use("/api/auth", authRouter)
http://localhost:4000
/api/auth
app.use("/api/posts", postRouter)
http://localhost:4000
/api/posts
app.use("/api/posts", chatRouter)
http://localhost:4000
/api/chat
app.use("/api/posts", messageRouter)
http://localhost:4000
/api/messages

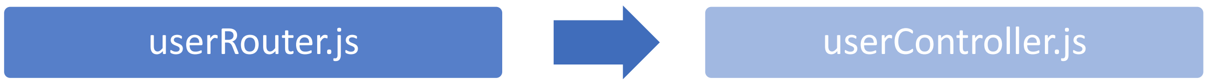
The use of route prefixes is beneficial for organizing and structuring an API. It allows different parts of the application to have their dedicated routers and handlers, making the codebase more modular and easier to maintain. Additionally, it provides a clear and intuitive structure for developers and users interacting with the API.

9. User Managing Controller

The userController.js file encapsulates various handlers that perform CRUD operations on user data within a Node.js application **(Node.js, 2023).** These operations include retrieving users, searching for users, updating user information, and managing user relationships such as following and unfollowing. The implementation follows a structured and modular approach, enhancing code maintainability and readability. The error handling approach, which delegates errors to the next middleware, contributes to a robust and user-friendly application. Overall, the user controller plays a crucial role in managing user-related functionalities, adhering to principles of RESTful API design, and providing a foundation for scalable and extensible user management in the application. **(Fielding, 2000).**

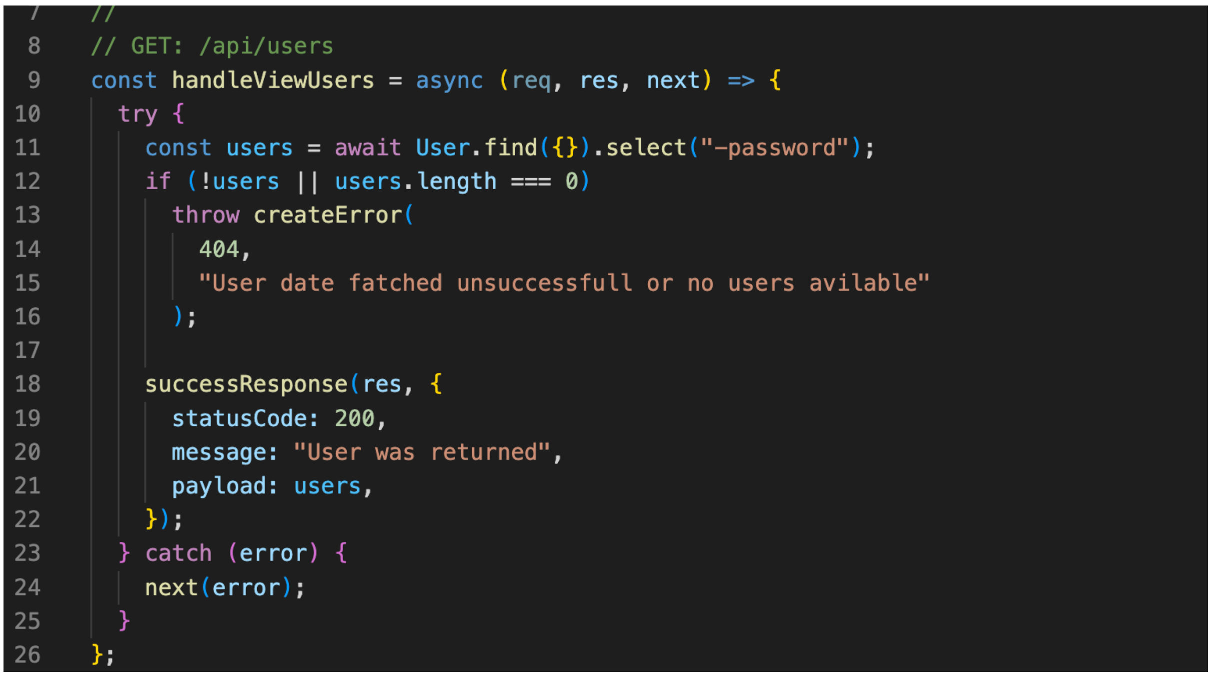
9.1 User Router and Controllers

This line associates the route with the **userRouter**. It means that any request to a URL that starts with **"http://localhost/api/users"** will be directed to the routes defined in userRouter.js file. This is a common approach to organize and modularize routes based on their functionality. **(Express.js, 2022).**



9.2 User Data Retrieval

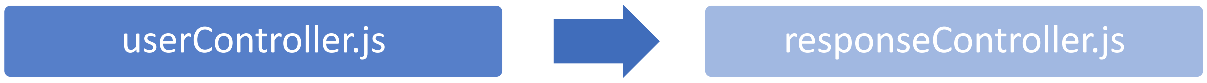
The **handleViewUsers** function in the userController.js file is designed to retrieve a list of users from the database and send a response to the client.



```
//
// GET: /api/users
const handleViewUsers = async (req, res, next) => {
  try {
    const users = await User.find({}).select("-password");
    if (!users || users.length === 0)
      throw createError(
        404,
        "User date fatched unsuccessfull or no users avilable"
      );

    successResponse(res, {
      statusCode: 200,
      message: "User was returned",
      payload: users,
    });
  } catch (error) {
    next(error);
  }
};
```

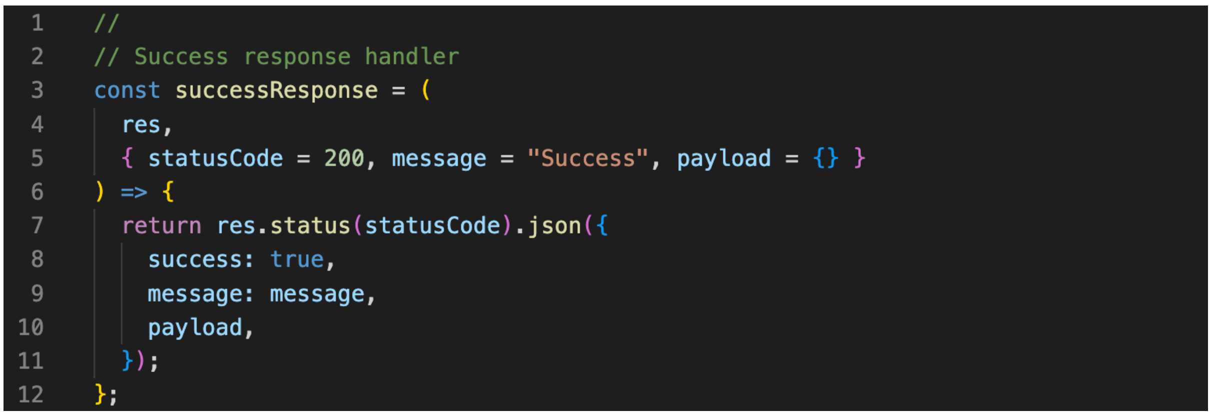
Figure 12: An example of user data retrieval (Source Code, 2023)

The handleViewUsers function serves the purpose of retrieving user data from the database and providing it as a response to the client. It utilizes the User model to perform a query to find all users, excluding their passwords for security reasons. The asynchronous nature of the function is managed through the use of await. After fetching the user data, it checks if any users were returned and, if not, throws a 404-error using the createError function, indicating that the user data retrieval was unsuccessful or that no users are available. In the case of a successful data retrieval, it sends a well-structured response using the **successResponse** function, communicating a 200-status code, a message stating that the users were returned, and the actual user data as the payload. Any encountered errors during this process are forwarded to the error-handling middleware using next(error). Overall, the handleViewUsers function ensures a resilient and user-friendly response in scenarios where user data is successfully retrieved or when an error occurs in the process. **(Express.js, 2022).**



9.3 Success Response Controller

The **successResponse** function in the provided code snippet is a utility function designed to streamline the generation of successful responses in a Node.js application **(Node.js, 2023)**. In a paragraph, we can describe its functionality:



```
//
// Success response handler
const successResponse = (
  res,
  { statusCode = 200, message = "Success", payload = {} }
) => {
  return res.status(statusCode).json({
    success: true,
    message: message,
    payload,
  });
};
```

Figure 13: An example of success response function (Source Code, 2023)

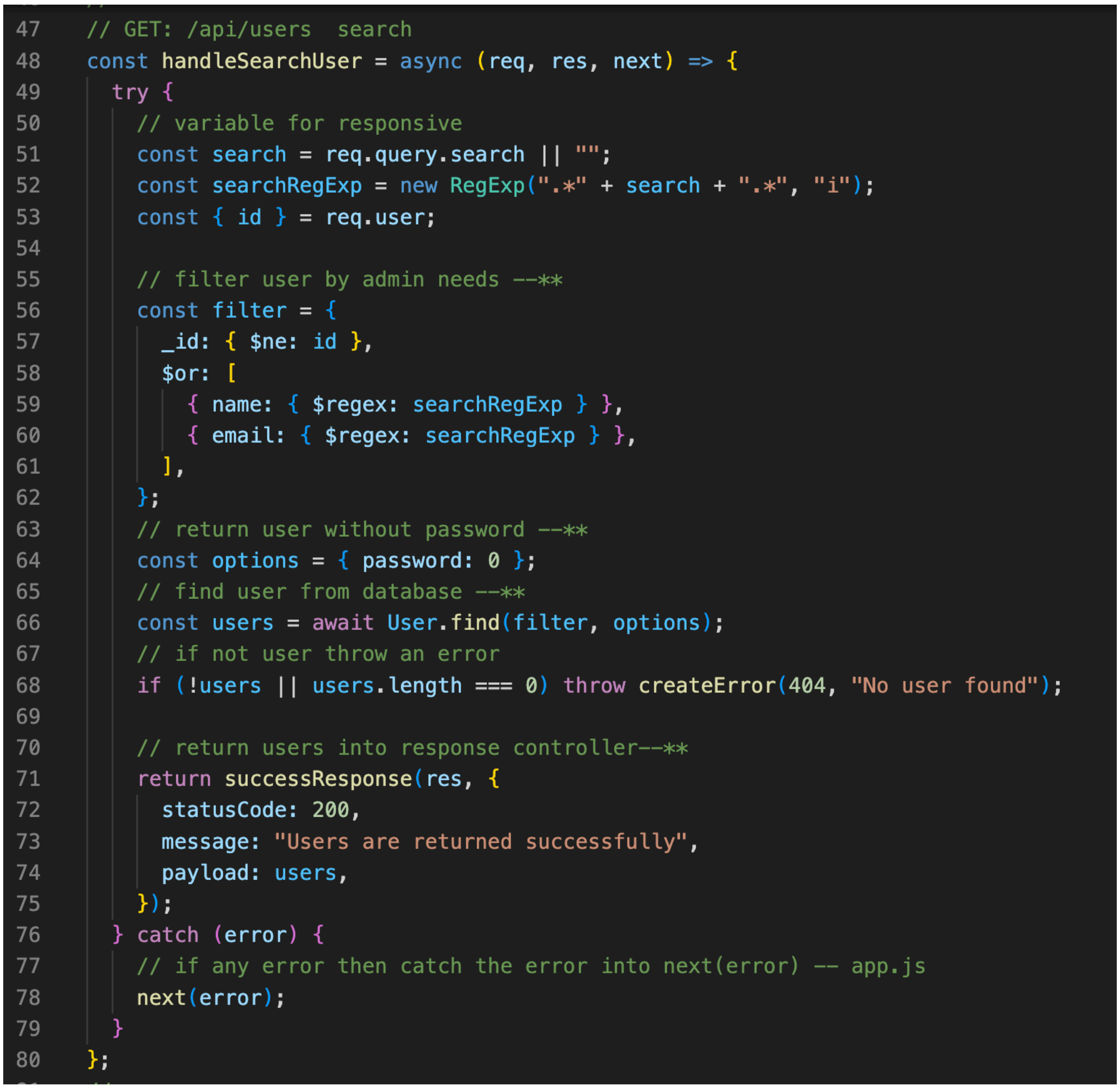
The successResponse function simplifies the process of sending successful responses to clients in application. It takes as parameters the Express response object (res) and an object containing optional values for the HTTP status code (statusCode), a success message (message), and a payload of data (payload). By default, it sets the status code to 200 (OK) and provides a generic success message. The function then constructs a JSON response with a consistent structure: a boolean success indicating the operation's success, a descriptive message, and an optional payload containing relevant data. This standardized structure facilitates clarity and consistency in the communication between the server and client. The function concludes by sending the JSON response using the specified status code and returning the response object. In summary, the successResponse function contributes to maintainable and readable code by providing a unified approach to crafting successful responses in various parts of the application.

9.3 User Search Functionality

The **handleSearchUser** function in the provided code snippet serves the purpose of searching for users based on a given query and returning the matched user data.

The handleSearchUser function is designed to handle user searches within the application. It begins by extracting the search query from the request parameters, defaulting to an empty string if not provided. The function then creates a case-insensitive regular expression (searchRegExp) using the search query, allowing for flexible and inclusive search patterns. The user's own ID is retrieved from the request object, presumably to exclude the current user from the search results. **(RegExp, 2023).**

To filter users based on the search query and admin requirements, the function constructs a filter object. This object specifies criteria for the search, excluding the current user's ID ($ne: id) and checking for matches in the user's name and email using the regular expression.



```
// GET: /api/users  search
const handleSearchUser = async (req, res, next) => {
  try {
    // variable for responsive
    const search = req.query.search || "";
    const searchRegExp = new RegExp(".*" + search + ".*", "i");
    const { id } = req.user;

    // filter user by admin needs --**
    const filter = {
      _id: { $ne: id },
      $or: [
        { name: { $regex: searchRegExp } },
        { email: { $regex: searchRegExp } },
      ],
    };
    // return user without password --**
    const options = { password: 0 };
    // find user from database --**
    const users = await User.find(filter, options);
    // if not user throw an error
    if (!users || users.length === 0) throw createError(404, "No user found");

    // return users into response controller--**
    return successResponse(res, {
      statusCode: 200,
      message: "Users are returned successfully",
      payload: users,
    });
  } catch (error) {
    // if any error then catch the error into next(error) -- app.js
    next(error);
  }
};
```

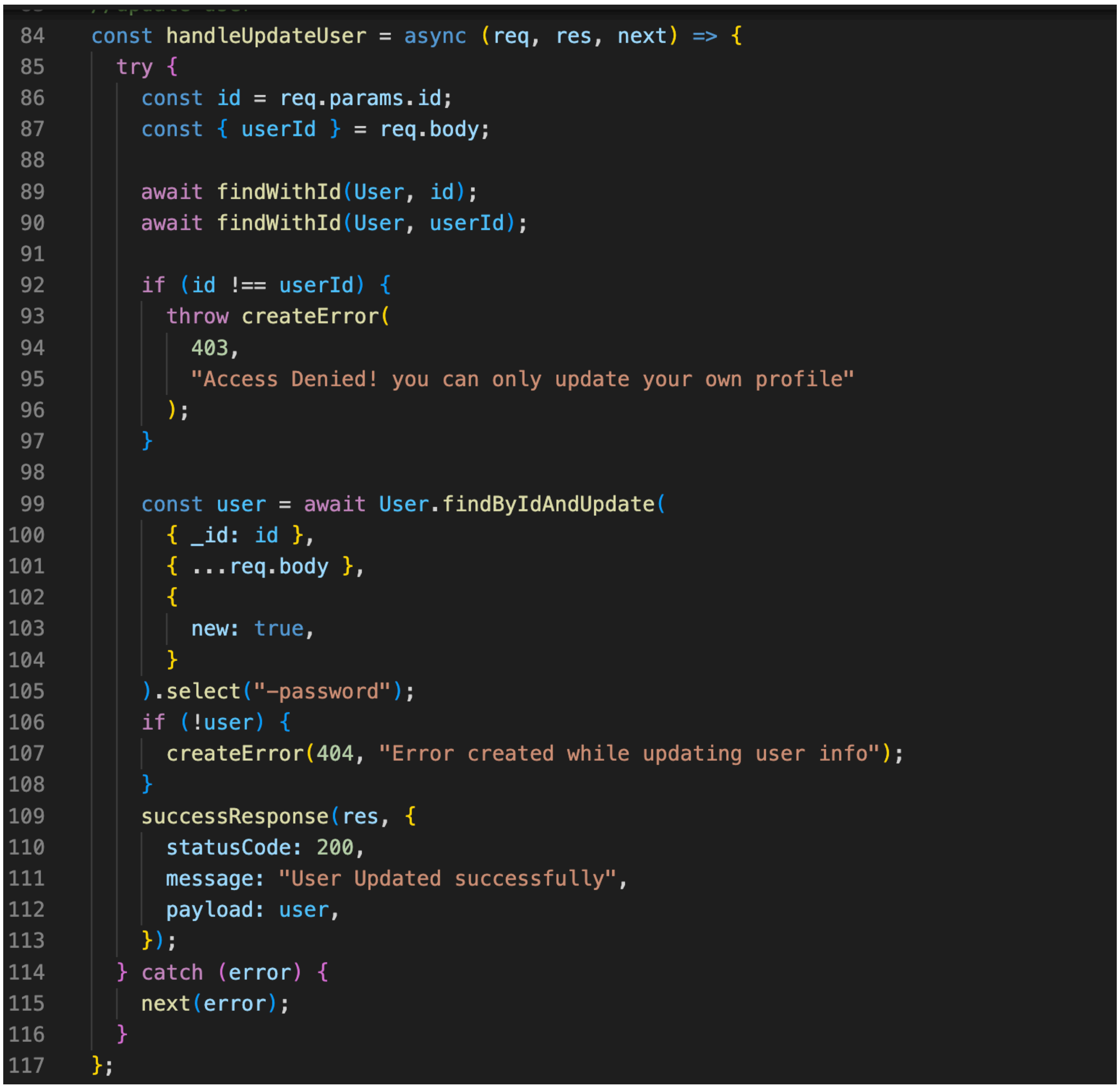
Figure 14: An example of user search functionality (Source Code, 2023)

The function then defines options to exclude the password from the returned user data. It performs a database query using the User model, applying the filter and options. If no users are found or the result is an empty array, it throws a 404-error using createError, indicating that no users were found. **(Express.js, 2022).**

In the case of a successful search, the function returns a well-structured response using the successResponse function. This response includes a 200-status code, a success message, and the retrieved users as the payload. Any errors encountered during the process are caught and forwarded to the error-handling middleware (next(error)). In summary, the handleSearchUser function provides a robust and organized approach to user searches, offering clear and informative responses to clients while handling potential errors effectively.

9.4 Update User Profile Functionality

The **handleUpdateUser** function in the provided code snippet manages the process of updating a user's information in the application. **(Express.js, 2022).**



```
const handleUpdateUser = async (req, res, next) => {
  try {
    const id = req.params.id;
    const { userId } = req.body;

    await findWithId(User, id);
    await findWithId(User, userId);

    if (id !== userId) {
      throw createError(
        403,
        "Access Denied! you can only update your own profile"
      );
    }

    const user = await User.findByIdAndUpdate(
      { _id: id },
      { ...req.body },
      {
        new: true,
      }
    ).select("-password");
    if (!user) {
      createError(404, "Error created while updating user info");
    }
    successResponse(res, {
      statusCode: 200,
      message: "User Updated successfully",
      payload: user,
    });
  } catch (error) {
    next(error);
  }
};
```

Figure 15: An example of update user profile (Source Code, 2023)

The handleUpdateUser function is responsible for updating a user's profile within the application. It begins by extracting the user ID from the request parameters (id) and the updated user ID from the request body (userId). The function then ensures the existence of both the user to be updated and the user making the request by using the **findWithId** function for each. This step adds a layer of security by confirming the validity of the users involved in the update operation.

To prevent unauthorized updates, the function compares the extracted id and userId. If they do not match, the function throws a 403-error using createError, indicating that the update is denied since users can only update their own profiles.

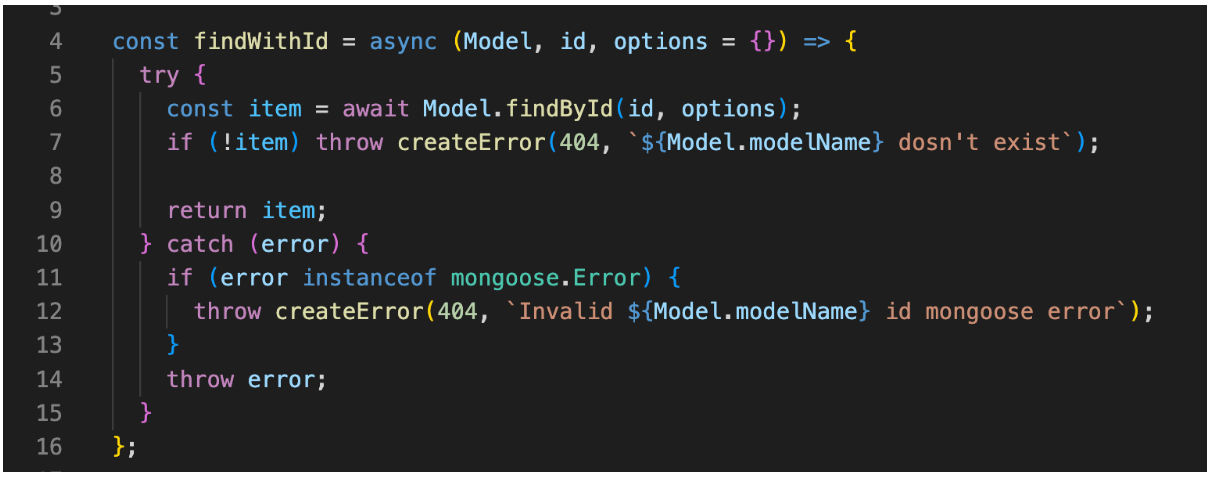
Subsequently, the function utilizes the User model's findByIdAndUpdate method to update the user's information with the provided data in the request body (req.body). The option {new: true} ensures that the function returns the updated user data rather than the original data. The user's password is excluded from the response for security reasons.

If the update operation is successful, the function sends a well-structured response using the successResponse function. This response includes a 200-status code, a success message, and the updated user data as the payload. In case of any errors during the update process, the function forwards the error to the error-handling middleware using next(error).

In summary, the handleUpdateUser function ensures the secure and authorized update of user profiles, providing clear and informative responses to clients while handling potential errors effectively. **(Express.js, 2022).**

9.5 Find User Functionality

The **findWithId** function in the provided code snippet is a utility function designed to retrieve an item from the database based on its ID using the 'Model.findById' method.



```
const findWithId = async (Model, id, options = {}) => {
  try {
    const item = await Model.findById(id, options);
    if (!item) throw createError(404, `${Model.modelName} dosn't exist`);

    return item;
  } catch (error) {
    if (error instanceof mongoose.Error) {
      throw createError(404, `Invalid ${Model.modelName} id mongoose error`);
    }
    throw error;
  }
};
```

Figure 16: An example of find user with id.

The 'findWithId' function takes three parameters: 'Model' (representing the Mongoose model to query), 'id' (representing the ID of the item to retrieve), and 'options' (additional options for the 'findById' method, with a default value of an empty object). **(Mongoose, 2023).**

Inside the function, it attempts to retrieve the item from the database using 'Model.findById(id, options)'. If the item is not found (i.e., if the result is false), the function throws a '404' error using 'createError', indicating that the item with the specified ID does not exist.

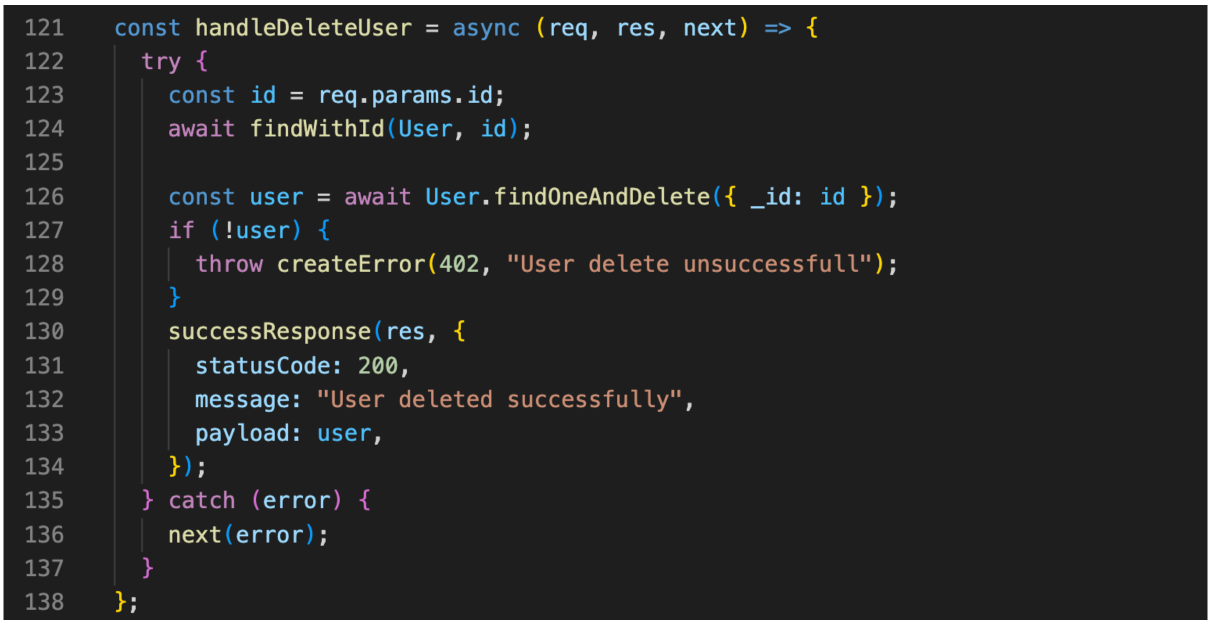
Additionally, the function includes error handling for cases where a mongoose error occurs during the retrieval process. If the error is an instance of 'mongoose.Error'**(Mongoose, 2023),** it throws a '404' error with a message indicating an invalid mongoose ID error. For other types of errors, it rethrows the error.

In summary, the 'findWithId' function provides a reusable and standardized way to retrieve items from the database based on their IDs. It includes error handling to manage cases where the item is not found or if there's a mongoose error during the retrieval process. This function can be employed across various parts of the application to ensure consistency in handling ID-based queries. **(Express.js, 2022).**

9.6 User Deletion Functionality

The **handleDeleteUser** function in the provided code snippet is responsible for deleting a user from the application.

The handleDeleteUser function starts by extracting the user ID from the request parameters (req.params.id). It then uses the findWithId function to ensure the existence of the user with the specified ID. This step is crucial for validating that the user to be deleted actually exists, providing an extra layer of security. **(Express.js, 2022).**



```
const handleDeleteUser = async (req, res, next) => {
  try {
    const id = req.params.id;
    await findWithId(User, id);

    const user = await User.findOneAndDelete({ _id: id });
    if (!user) {
      throw createError(402, "User delete unsuccessfull");
    }
    successResponse(res, {
      statusCode: 200,
      message: "User deleted successfully",
      payload: user,
    });
  } catch (error) {
    next(error);
  }
};
```

Figure 17:An example of delete user (Source Code, 2023)

Following the validation, the function uses the User model's findOneAndDelete method to find and delete the user with the specified ID using mongoose **(Mongoose, 2023)**. If the deletion is successful, the function proceeds to create a response using the successResponse function. This response includes a 200-status code, a success message indicating that the user was deleted successfully, and the deleted user's data as the payload.

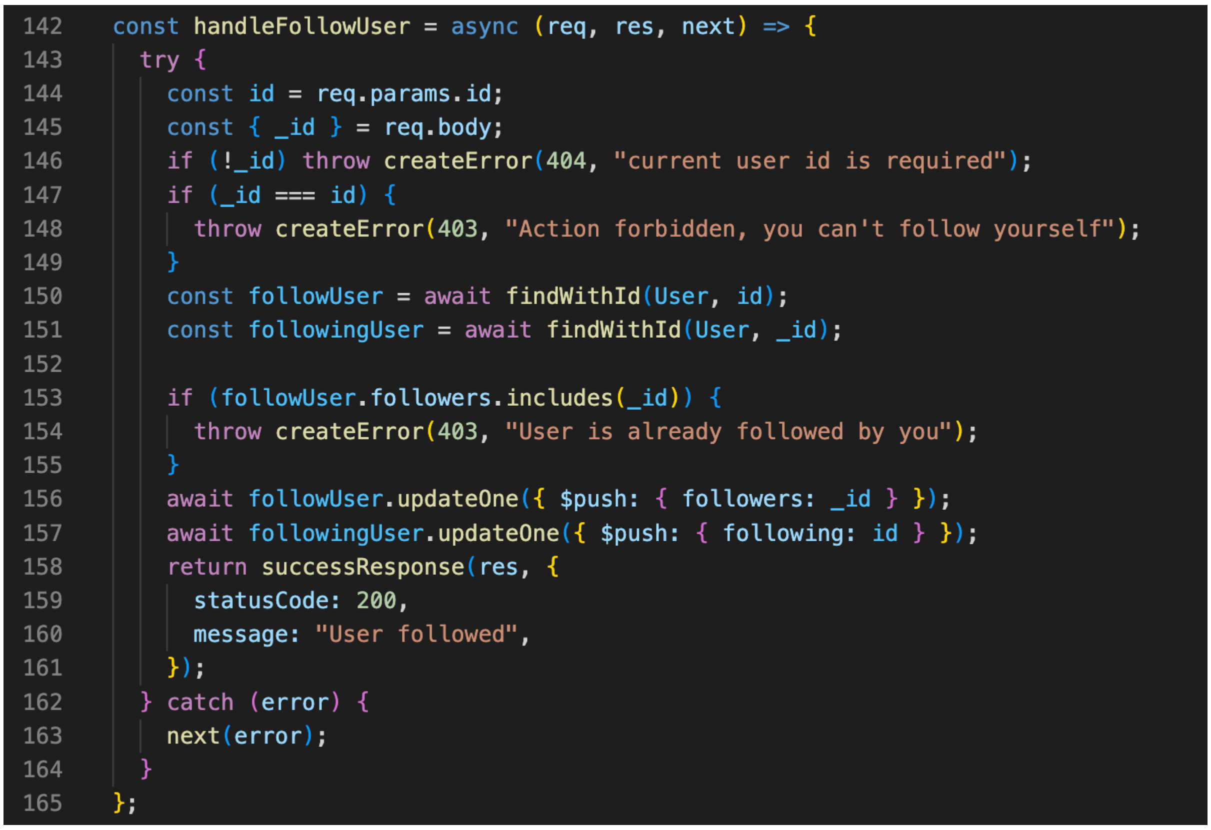
In the event that no user is found for deletion, the function throws a 402-error using createError, indicating that the deletion was unsuccessful. Any errors encountered during the process are caught and forwarded to the error-handling middleware using next(error).

In summary, the handleDeleteUser function provides a secure and systematic approach to deleting users from the application, ensuring that only existing users can be deleted and providing clear and informative responses to clients while handling potential errors effectively. **(Express.js, 2022).**

9.7 User Following Functionality

The **handleFollowUser** function in the provided code snippet manages the process of a user following another user within the application.

The handleFollowUser function begins by extracting the user ID from the request parameters (req.params.id) and the current user's ID from the request body (req.body._id). It then checks if the current user ID is provided; if not, it throws a 404-error using createError, indicating that the current user ID is required.



```javascript
const handleFollowUser = async (req, res, next) => {
  try {
    const id = req.params.id;
    const { _id } = req.body;
    if (!_id) throw createError(404, "current user id is required");
    if (_id === id) {
      throw createError(403, "Action forbidden, you can't follow yourself");
    }
    const followUser = await findWithId(User, id);
    const followingUser = await findWithId(User, _id);

    if (followUser.followers.includes(_id)) {
      throw createError(403, "User is already followed by you");
    }
    await followUser.updateOne({ $push: { followers: _id } });
    await followingUser.updateOne({ $push: { following: id } });
    return successResponse(res, {
      statusCode: 200,
      message: "User followed",
    });
  } catch (error) {
    next(error);
  }
};
```

Figure 18: An example of following user controller (Source Code, 2023)

A subsequent check ensures that a user cannot follow themselves by comparing the extracted user ID and the current user ID. If they match, the function throws a 403-error using createError, indicating that following oneself is forbidden. **(Express.js, 2022).**

The function proceeds to use the findWithId utility function to ensure the existence of both the user to be followed (followUser) and the current user (followingUser). This ensures that both users are valid entities within the application.

Following the validations, the function checks if the current user is already following the user to be followed by examining the follower's array of the followUser. If the current user ID is found in this array, it throws a 403 error, indicating that the user is already being followed.

If all validations pass, the function updates the follower's array of the followUser by pushing the current user ID (_id). Additionally, it updates the following array of the current user by pushing the user ID being followed (id). Both updates are performed using the $push operator.

Finally, the function sends a successful response using the successResponse function. This response includes a 200-status code, a message indicating that the user has been followed, and no payload.

Any errors encountered during this process are caught and forwarded to the error-handling middleware using next(error).

In summary, the handleFollowUser function provides a comprehensive and secure approach to user-following functionality, ensuring proper validations and clear responses to clients while handling potential errors effectively. **(Express.js, 2022).**

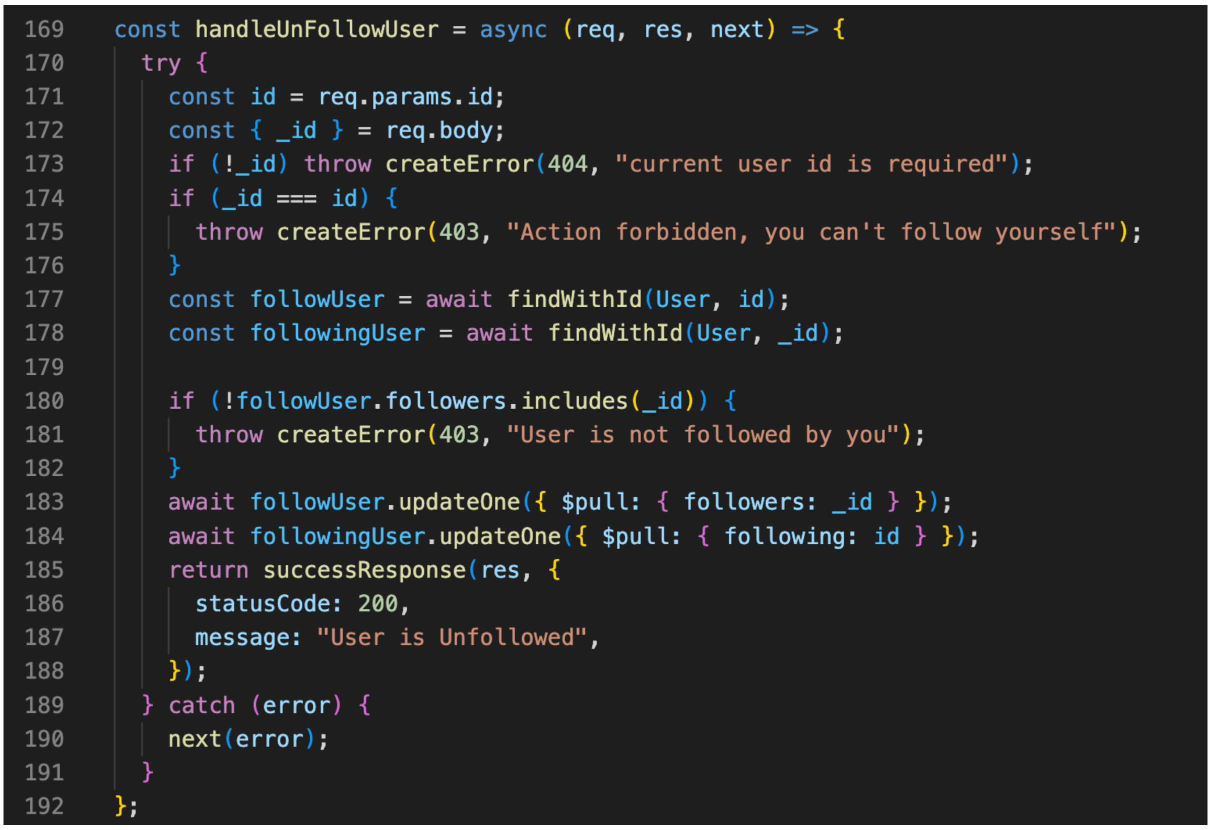
9.8 User Unfollowing Functionality

The **handleUnFollowUser** function begins by extracting the user ID from the request parameters (req.params.id) and the current user's ID from the request body (req.body._id). It then checks if the current user ID is provided; if not, it throws a 404-error using createError, indicating that the current user ID is required.

A subsequent check ensures that a user cannot unfollow themselves by comparing the extracted user ID and the current user ID. If they match, the function throws a 403-error using createError, indicating that unfollowing oneself is forbidden.

The function proceeds to use the findWithId utility function to ensure the existence of both the user to be unfollowed (followUser) and the current user (followingUser). This ensures that both users are valid entities within the application.

Following the validations, the function checks if the current user is already following the user to be unfollowed by examining the follower's array of the followUser. If the current user ID is not found in this array, it throws a 403-error, indicating that the user is not currently being followed.



```
const handleUnFollowUser = async (req, res, next) => {
  try {
    const id = req.params.id;
    const { _id } = req.body;
    if (!_id) throw createError(404, "current user id is required");
    if (_id === id) {
      throw createError(403, "Action forbidden, you can't follow yourself");
    }
    const followUser = await findWithId(User, id);
    const followingUser = await findWithId(User, _id);

    if (!followUser.followers.includes(_id)) {
      throw createError(403, "User is not followed by you");
    }
    await followUser.updateOne({ $pull: { followers: _id } });
    await followingUser.updateOne({ $pull: { following: id } });
    return successResponse(res, {
      statusCode: 200,
      message: "User is Unfollowed",
    });
  } catch (error) {
    next(error);
  }
};
```

Figure 19: An example of unfollow user handler (Source Code, 2023)

If all validations pass, the function updates the follower's array of the followUser by pulling out the current user ID (_id). Additionally, it updates the following array of the current user by pulling out the user ID being unfollowed (id). Both updates are performed using the $pull operator.

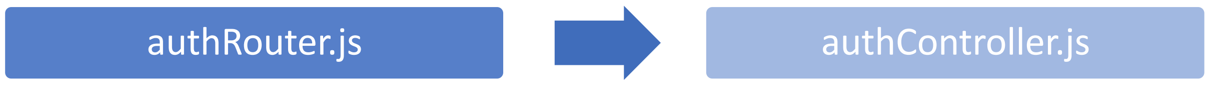
Finally, the function sends a successful response using the successResponse function. This response includes a 200-status code, a message indicating that the user has been unfollowed, and no payload.

Any errors encountered during this process are caught and forwarded to the error-handling middleware using next(error).

In summary, the handleUnFollowUser function provides a comprehensive and secure approach to user-unfollowing functionality, ensuring proper validations and clear responses to clients while handling potential errors effectively.

10. Authentication router and controllers

This line associates the route with the **authRouter**. Requests to **http://localhost:4000/api/auth** will be handled by the routes defined in authRouter.js file which is express router **(Express.js, 2022).** This separation allows for a cleaner and more maintainable code structure, especially as the application grows**.**



10.1 Managing User Authentication

The **authController.js** file encapsulates user authentication functionalities in a Node.js application. It includes two key functions, ‘handleRegisterUser’ and ‘handleLogin’. The former manages the user registration process by extracting relevant data from the request body, passing it to the ‘createUser’ service for processing, and responding with a success message and the newly created user's data upon successful registration. The latter handles user login, extracting login credentials, utilizing the ‘loginUser’ service, generating a JSON Web Token (JWT) through the ‘createJsonWebToken’ **(JWT, 2023)** helper function, and responding with the user's data along with the generated JWT upon successful authentication. Both functions leverage the ‘successResponse’ function for consistent success responses. The module also includes imports for necessary helpers, secrets, and services, making it a central component for managing user authentication in the application. **(Express.js, 2022).**

10.2 User Schema and Model Definition with Mongoose

The provided code defines a MongoDB schema **(MongoDB, 2023)** for a user in a Node.js application using Mongoose **(Mongoose, 2023).** The **userSchema** includes fields such as name, email, password, isAdmin, image, coverImage, about, livesin, worksAt, relationship, followers, and following. Notably, the password field incorporates a set function that utilizes bcrypt **(bcryptjs, 2017)** to hash and salt the password for secure storage**.**

The schema specifies default values for some fields, such as empty strings or default descriptions. The timestamps option is set to true, which automatically adds createdAt and updatedAt fields to track document creation and modification times. The User model is created using mongoose.model and is exported for use in other parts of the application.

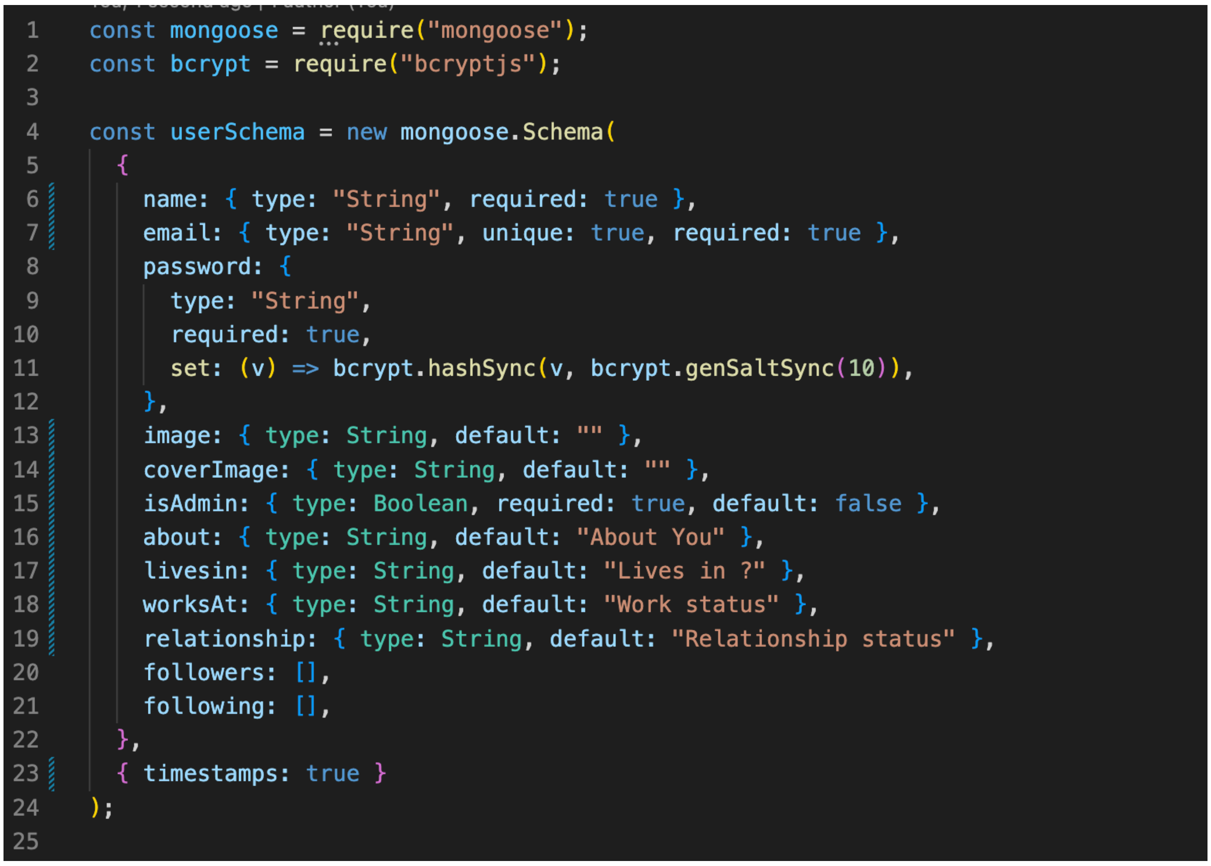

```
const mongoose = require("mongoose");
const bcrypt = require("bcryptjs");

const userSchema = new mongoose.Schema(
  {
    name: { type: "String", required: true },
    email: { type: "String", unique: true, required: true },
    password: {
      type: "String",
      required: true,
      set: (v) => bcrypt.hashSync(v, bcrypt.genSaltSync(10)),
    },
    image: { type: String, default: "" },
    coverImage: { type: String, default: "" },
    isAdmin: { type: Boolean, required: true, default: false },
    about: { type: String, default: "About You" },
    livesin: { type: String, default: "Lives in ?" },
    worksAt: { type: String, default: "Work status" },
    relationship: { type: String, default: "Relationship status" },
    followers: [],
    following: [],
  },
  { timestamps: true }
);

```

Figure 20: An example of user schema (Source Code, 2023)

In summary, this Mongoose **(Mongoose, 2023)** schema represents a user entity with various attributes and incorporates password hashing for enhanced security in a MongoDB database **(MongoDB, 2023)**. Note that the duplicate isAdmin field should be corrected for accurate schema definition**.**

10.3 User Registration Handling

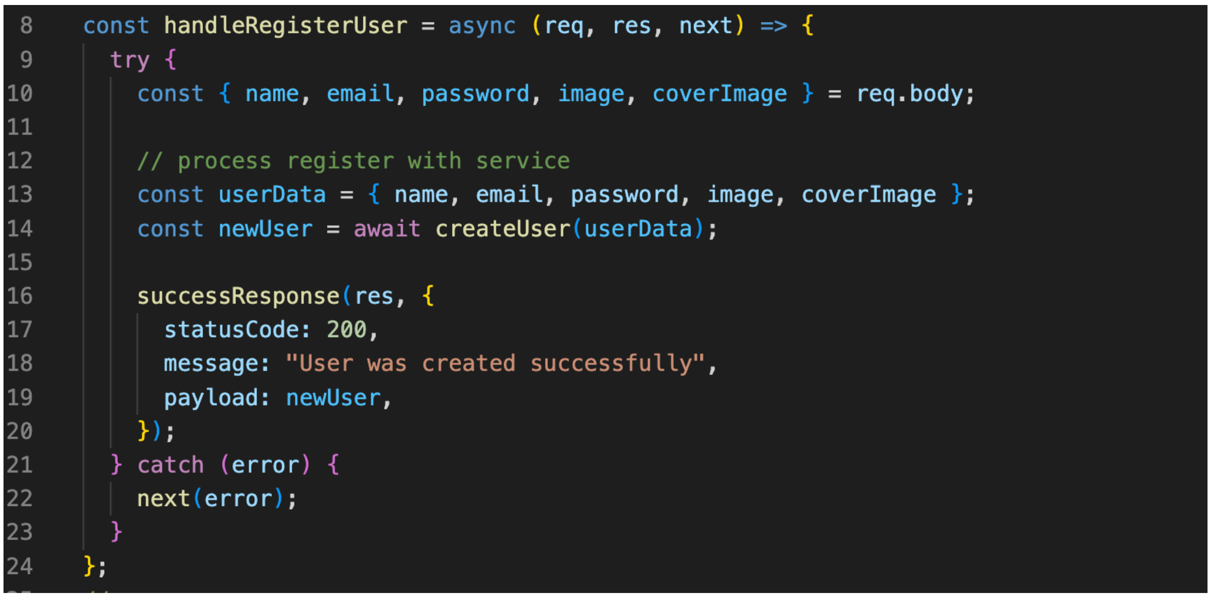
The **handleRegisterUser** function in the provided code snippet manages the process of registering a new user in the application. Let's break down its functionality:

The handleRegisterUser function begins by extracting relevant information from the request body, including the user's name, email, password, image, and coverImage.

Next, it creates a userData object containing the extracted information, and then proceeds to call the createUser function with this data. The purpose of createUser is assumed to be handling the registration process and creating a new user in the application.

If the registration process is successful and a new user is created, the function sends a successful response using the successResponse function. This response includes a 200-status code, a message indicating that the user was created successfully, and the newly created user's data as the payload.

In case of any errors during the registration process, the function catches the error and forwards it to the error-handling middleware using next(error).



```javascript
const handleRegisterUser = async (req, res, next) => {
  try {
    const { name, email, password, image, coverImage } = req.body;

    // process register with service
    const userData = { name, email, password, image, coverImage };
    const newUser = await createUser(userData);

    successResponse(res, {
      statusCode: 200,
      message: "User was created successfully",
      payload: newUser,
    });
  } catch (error) {
    next(error);
  }
};
```

Figure 21: An example of user registration handler (Source Code, 2023)

In summary, the handleRegisterUser function provides a straightforward and organized approach to user registration, creating a new user in the application and responding with clear messages and appropriate status codes. **(Express.js, 2022).**

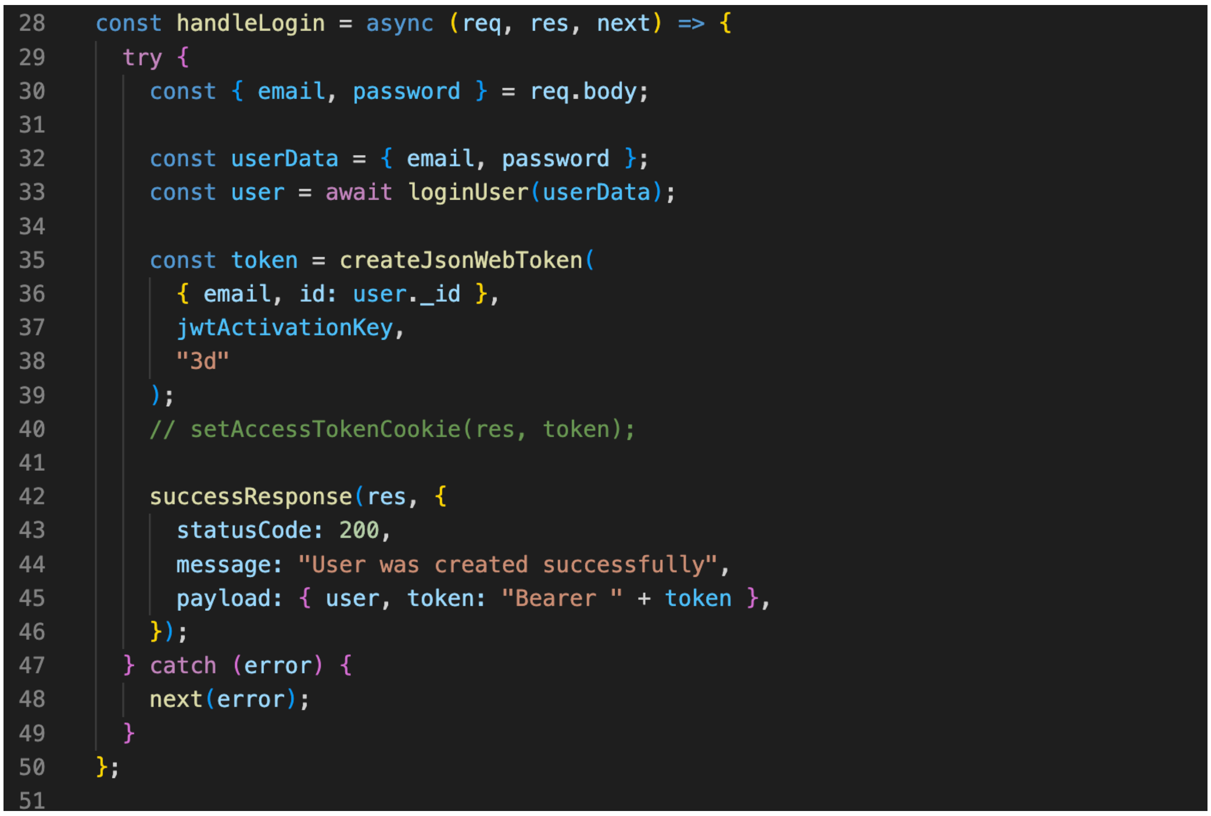
10.4 User Login Handling

The **handleLogin** function in the provided code snippet manages the user login process in a Node.js application (**Node.js, 2023)**. It begins by extracting the email and password from the request body. Subsequently, it creates a userData object with these credentials and calls the loginUser service to authenticate the user.

Upon successful authentication, the function generates a JSON Web Token **(JWT, 2023)** using the createJsonWebToken helper function. The token is constructed with the user's email and id, along with a predefined activation key and an expiration time of 3 days.

The code includes a commented-out line, presumably for setting an access token cookie, but it is currently inactive.

Finally, the function responds with a success message, a 200-status code, and a payload containing the user's data and the generated JWT. In case of any errors during the login process, the function catches the error and forwards it to the error-handling middleware using next(error). **(Express.js, 2022).**



```
const handleLogin = async (req, res, next) => {
  try {
    const { email, password } = req.body;

    const userData = { email, password };
    const user = await loginUser(userData);

    const token = createJsonWebToken(
      { email, id: user._id },
      jwtActivationKey,
      "3d"
    );
    // setAccessTokenCookie(res, token);

    successResponse(res, {
      statusCode: 200,
      message: "User was created successfully",
      payload: { user, token: "Bearer " + token },
    });
  } catch (error) {
    next(error);
  }
};
```

Figure 22: An example of user login handler (Source Code, 2023)

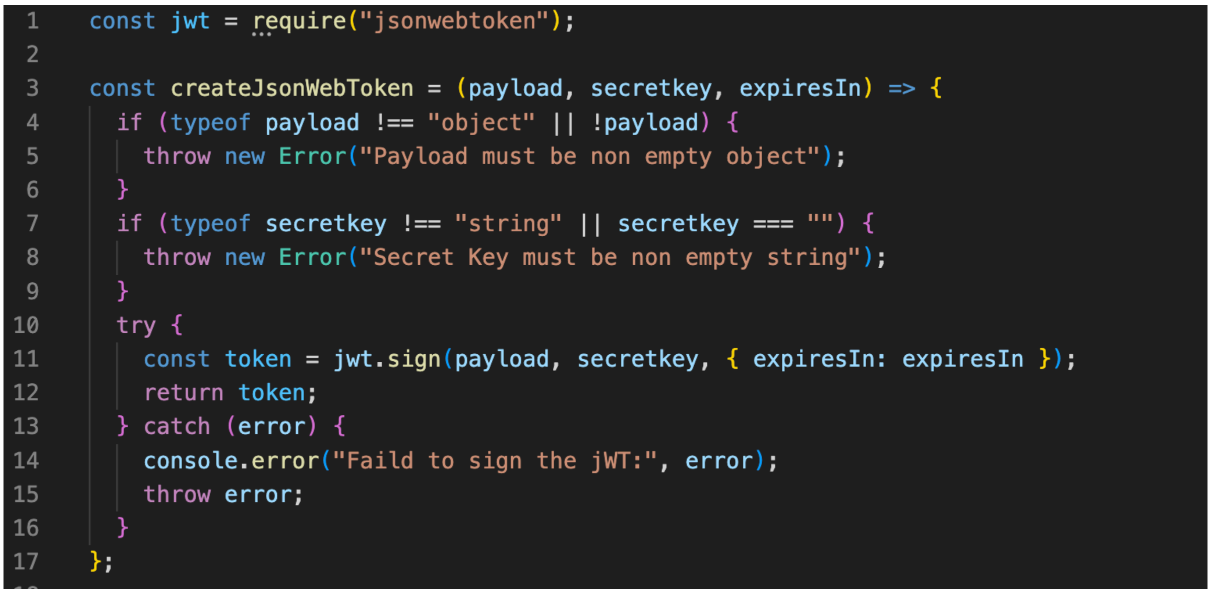
In summary, the handleLogin function orchestrates the user login procedure, incorporating secure token generation and providing a clear and standardized response upon successful authentication.

10.5 JSON Web Token (JWT) Generation Utility

The **createJsonWebToken** function in the provided code is a utility for generating JSON Web Tokens **(JWT, 2023)** in a Node.js application. It takes three parameters: payload (an object containing the data to be included in the token), secretkey (a string serving as the secret key for signing the token), and expiresIn (a string indicating the expiration time for the token).

The function first validates that the payload is a non-empty object and that the secretkey is a non-empty string. If these conditions are not met, the function throws an error**.**

Subsequently, the function uses the jwt.sign method to sign the JWT with the provided payload and secret key. The expiresIn parameter is used to set the expiration time for the token. If the token generation is successful, the function returns the generated token. In case of any errors during the signing process, it catches the error, logs a message to the console, and rethrows the error. **(Node.js, 2023).**



```
const jwt = require("jsonwebtoken");

const createJsonWebToken = (payload, secretkey, expiresIn) => {
  if (typeof payload !== "object" || !payload) {
    throw new Error("Payload must be non empty object");
  }
  if (typeof secretkey !== "string" || secretkey === "") {
    throw new Error("Secret Key must be non empty string");
  }
  try {
    const token = jwt.sign(payload, secretkey, { expiresIn: expiresIn });
    return token;
  } catch (error) {
    console.error("Faild to sign the jWT:", error);
    throw error;
  }
};
```

Figure 23: An example of creating JSON web Token (Source Code, 2023)

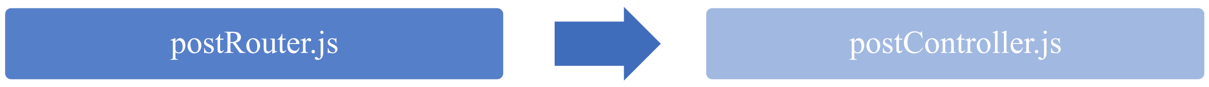
In summary, the createJsonWebToken function provides a robust and secure mechanism for generating JWTs with proper input validation and error handling.

11. Managing Social Media Posts

The **postController.js** file manages various operations related to posts in a social media application. It includes functions to retrieve all posts, get posts created by a specific user, fetch a post by its ID, create a new post, update an existing post, delete a post, like/unlike a post, and retrieve posts for a user's timeline. These functions utilize the PostModel and UserModel from the model's directory, along with helper functions like findWithId and error handling using http-errors. The file follows a modular structure, ensuring that each function handles a specific aspect of post management, promoting code organization and readability. The functions also incorporate validation checks, such as ensuring user login for certain actions and checking image size constraints. Overall, the postController.js file serves as a comprehensive controller for handling post-related operations in the application. **(Express.js, 2022).**

11.1 Post Router and Controllers

Post Router: This line associates the route with the **postRouter**. Requests to **http://localhost:4000/api/posts** will be directed to the routes defined in postRouter.js file. This follows the same pattern of organizing routes based on their specific functionality. **(Express.js, 2022).**



11.2 Post Schema and Model Definition with Mongoose

The **postModel** schema defines the structure of posts in a social media application using MongoDB and Mongoose. Each post is represented by a document with several properties. The "name" field stores the name of the post, the "userId" field corresponds to the user ID of the creator, ensuring a connection to the user who authored the post. The "desc" field allows for an optional description of the post. The "likes" field is an array that will contain user IDs who have liked the post. The "image" field stores the filename of an image associated with the post, with a default value of "default.jpg" if no image is provided. The schema also includes timestamps, automatically generating "createdAt" and "updatedAt" fields to track when the post was created or last updated. Overall, this schema provides a structured representation for posts, capturing essential information for a social media platform. **(Mongoose, 2023).**

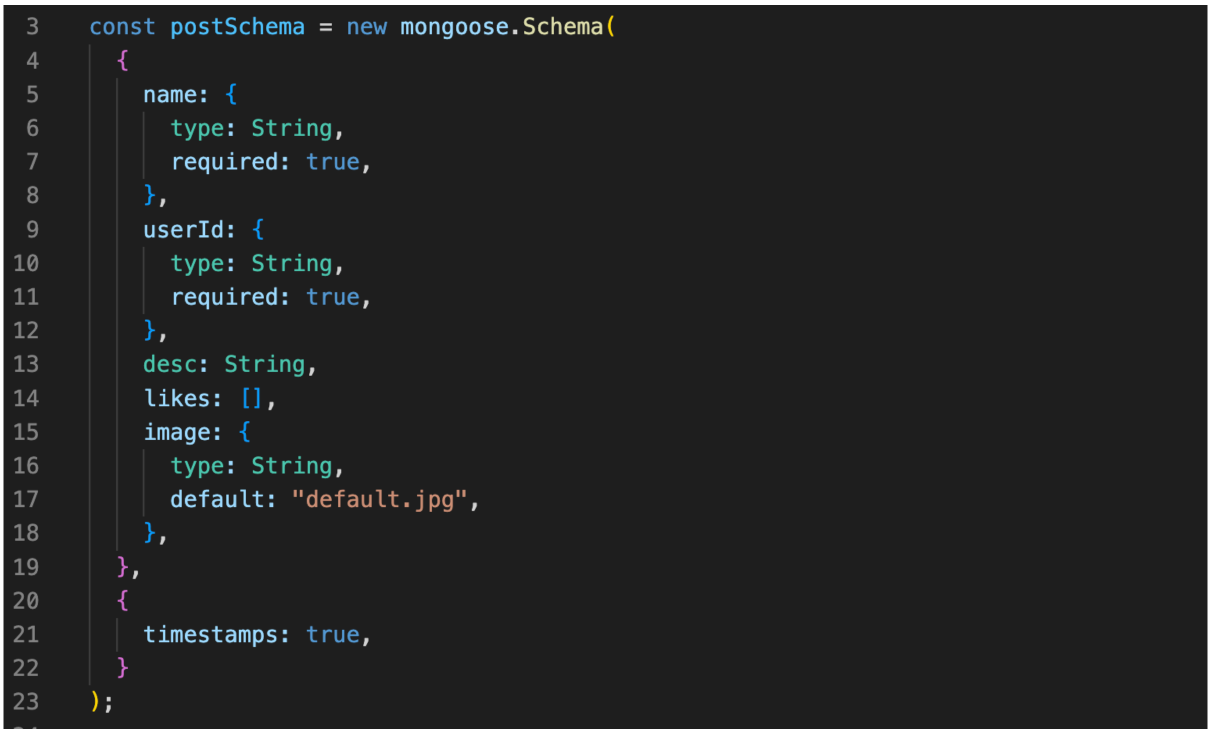
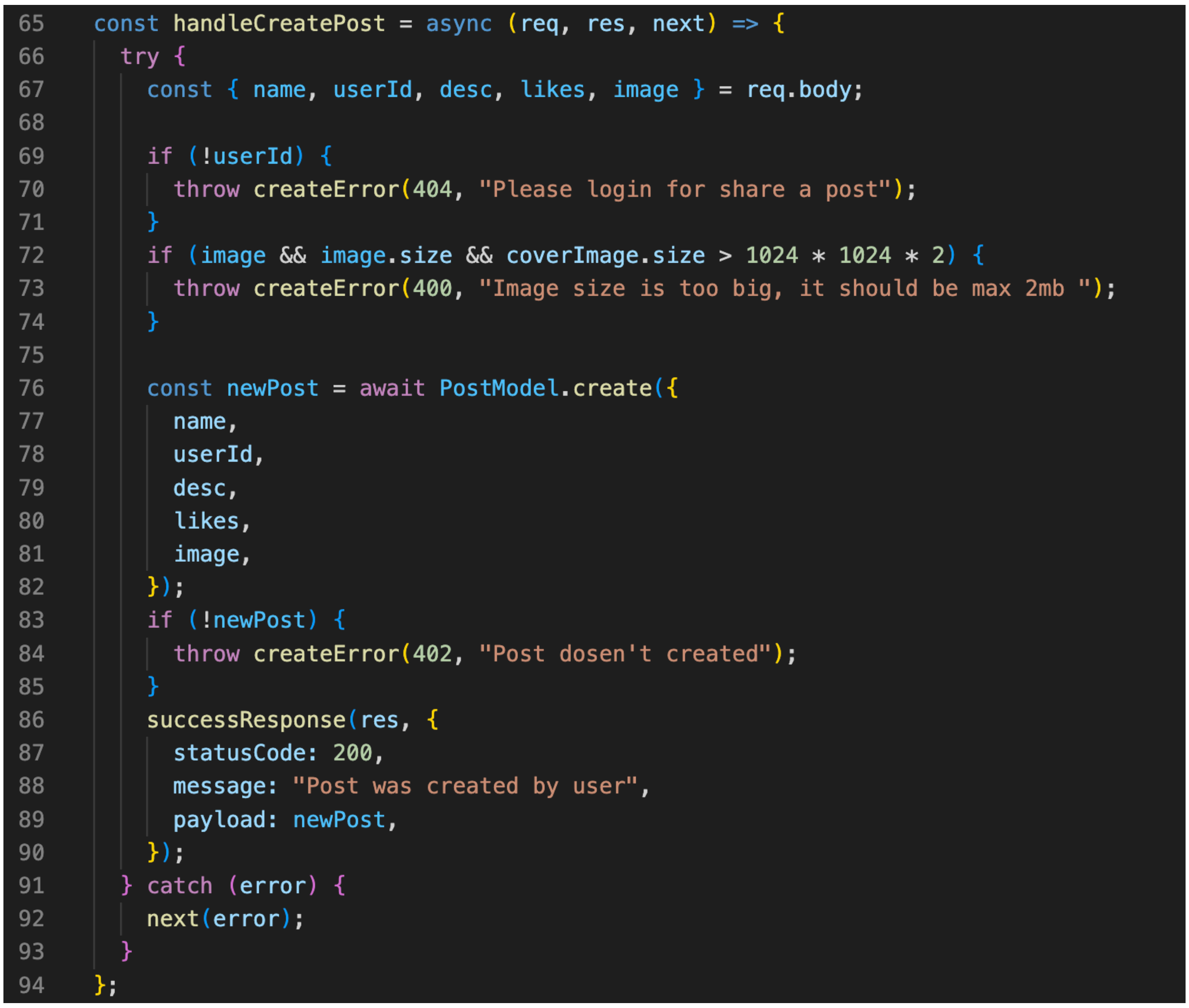

```
const postSchema = new mongoose.Schema(
  {
    name: {
      type: String,
      required: true,
    },
    userId: {
      type: String,
      required: true,
    },
    desc: String,
    likes: [],
    image: {
      type: String,
      default: "default.jpg",
    },
  },
  {
    timestamps: true,
  }
);
```

Figure 24: An example of Post schema (Source Code, 2023)

11.3 Creating New Posts

The **handleCreatePost** function is a controller responsible for creating a new post in a social media application. The function first extracts relevant information such as the post's "name," "userId," "desc," "likes," and "image" from the request body. It then performs validation checks, ensuring that a user is logged in before sharing a post. Additionally, it checks the size of the image, throwing an error if it exceeds the maximum limit of 2 megabytes.

Subsequently, the function attempts to create a new post using the PostModel**, (Mongoose, 2023)** which presumably represents the post schema in the application. If the post creation is successful, it sends a success response with a status code of 200, indicating that the post was created successfully. The response includes a message confirming the successful post creation and the details of the newly created post in the payload.



```javascript
const handleCreatePost = async (req, res, next) => {
  try {
    const { name, userId, desc, likes, image } = req.body;

    if (!userId) {
      throw createError(404, "Please login for share a post");
    }
    if (image && image.size && coverImage.size > 1024 * 1024 * 2) {
      throw createError(400, "Image size is too big, it should be max 2mb ");
    }

    const newPost = await PostModel.create({
      name,
      userId,
      desc,
      likes,
      image,
    });
    if (!newPost) {
      throw createError(402, "Post dosen't created");
    }
    successResponse(res, {
      statusCode: 200,
      message: "Post was created by user",
      payload: newPost,
    });
  } catch (error) {
    next(error);
  }
};
```

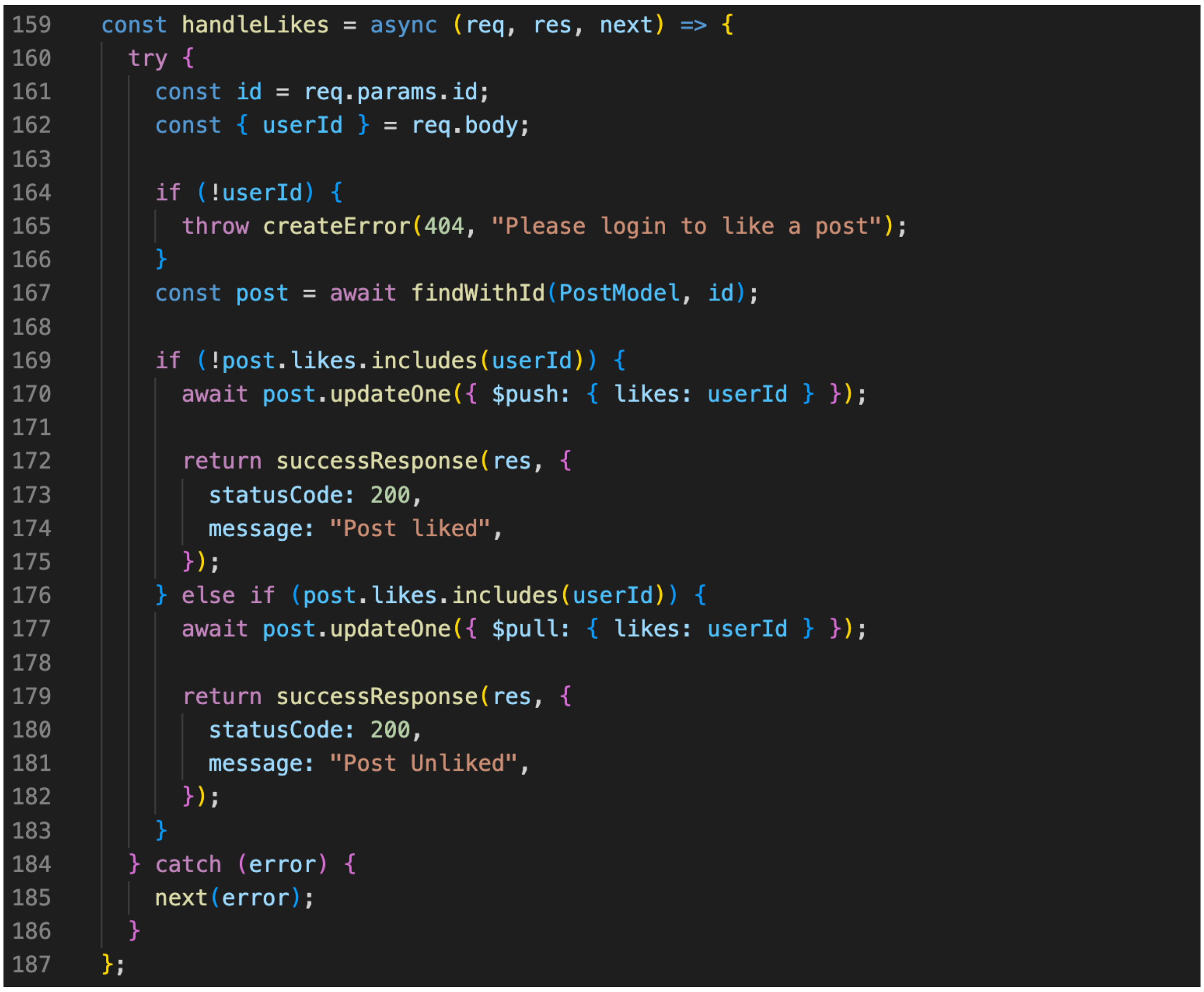
Figure 25: An example of creating post handler (Source Code, 2023)

In case of any errors during the process, the function catches the error and passes it to the next middleware in the chain using the "next" function, ensuring proper error handling in the application. Overall, the "handleCreatePost" function encapsulates the logic for creating new posts, incorporating user authentication and image size validation. **(Express.js, 2022).**

11.4 Managing Post Likes

The **handleLikes** function is designed to manage the likes/unlikes functionality for posts in a social media application. The logic is encapsulated within a try-catch block to handle potential errors during execution. The function extracts the post ID from the request parameters and the user ID from the request body. It then checks if a user is logged in, throwing an error if not, ensuring that only authenticated users can like or unlike a post.

Within the try block, the function fetches the corresponding post using the "findWithId" utility, presumably a function to find an item by ID. The code checks whether the user has already liked the post or not. If the user has not liked the post, the function updates the post document in the database by pushing the user's ID to the "likes" array. Conversely, if the user has already liked the post, the function updates the post by pulling the user's ID from the "likes" array, effectively unlike the post. **(Express.js, 2022)**.



```
const handleLikes = async (req, res, next) => {
  try {
    const id = req.params.id;
    const { userId } = req.body;

    if (!userId) {
      throw createError(404, "Please login to like a post");
    }
    const post = await findWithId(PostModel, id);

    if (!post.likes.includes(userId)) {
      await post.updateOne({ $push: { likes: userId } });

      return successResponse(res, {
        statusCode: 200,
        message: "Post liked",
      });
    } else if (post.likes.includes(userId)) {
      await post.updateOne({ $pull: { likes: userId } });

      return successResponse(res, {
        statusCode: 200,
        message: "Post Unliked",
      });
    }
  } catch (error) {
    next(error);
  }
};
```

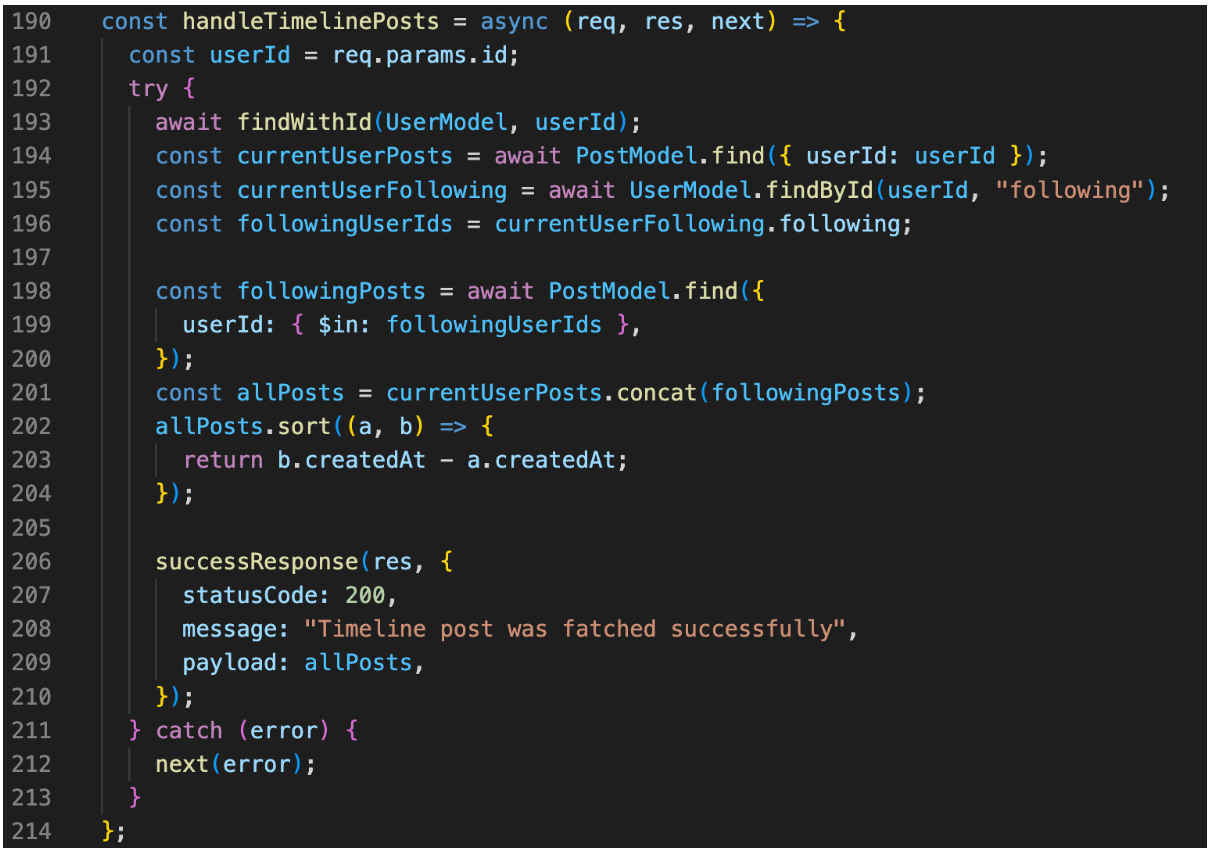
Figure 26: An example of likes handler (Source Code, 2023)

The response to these operations includes a success status code (200) along with a corresponding message indicating whether the post was liked or unliked. The overall purpose of this code is to handle the dynamic interaction of users with posts, allowing them to express their preferences through likes and unlike in a social media context. **(Node.js, 2023).**

11.5 Timeline Post Retrieval and Organization Function

The **handleTimelinePosts** function is designed to retrieve and organize posts for a user's timeline in a social media application. In this implementation, the timeline is constructed by combining posts created by the user (referred to as currentUserPosts) with posts from other users the current user is following (retrieved from followingUserIds). The code ensures that the user's existence is verified using the "findWithId" utility before proceeding. **(Express.js, 2022).**

Within the try block, the function queries the database for posts created by the current user and the users they are following. The result is an array of allPosts, which is a combination of the user's own posts and those from users they follow. To maintain a chronological order, the array is sorted based on the timestamp of the posts, with the most recent posts appearing first.



```
const handleTimelinePosts = async (req, res, next) => {
  const userId = req.params.id;
  try {
    await findWithId(UserModel, userId);
    const currentUserPosts = await PostModel.find({ userId: userId });
    const currentUserFollowing = await UserModel.findById(userId, "following");
    const followingUserIds = currentUserFollowing.following;

    const followingPosts = await PostModel.find({
      userId: { $in: followingUserIds },
    });
    const allPosts = currentUserPosts.concat(followingPosts);
    allPosts.sort((a, b) => {
      return b.createdAt - a.createdAt;
    });

    successResponse(res, {
      statusCode: 200,
      message: "Timeline post was fatched successfully",
      payload: allPosts,
    });
  } catch (error) {
    next(error);
  }
};
```

Figure 27: An example of retrieval timeline post (Source Code, 2023)

The response to this operation includes a success status code (200), a message indicating the successful retrieval of timeline posts, and the payload containing the sorted array of posts. This function serves the purpose of providing a user with a timeline view that aggregates posts from both the user and the accounts they follow, creating a comprehensive and ordered feed of relevant content. **(Mongoose, 2023).**

12 Managing Real-time Messaging Operations

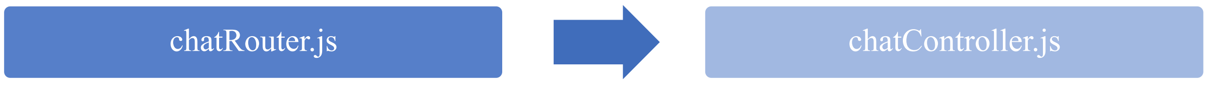
The chatController.js file handles the backend logic for managing user chats in a real-time messaging application. The first endpoint, handleCreateChat, allows users to initiate a chat by providing the sender and receiver IDs. It checks for existing chats between the specified users and creates a new chat if none exists. The second endpoint, handleUserChat, retrieves the list of chats associated with a specific user ID. It ensures the validity of the user ID and returns the chat list.

The third endpoint, handleFindChat, retrieves a specific chat between two users by providing their IDs. It validates the input IDs and fetches the corresponding chat. Lastly, the handleDeleteChat endpoint deletes a chat based on the provided chat ID. It validates the input ID and returns the deleted chat upon successful deletion.

The file employs error handling using the http-errors library and responds with success messages and payloads using the successResponse function from the responseController.js file. Overall, this controller file manages chat-related operations in the application.

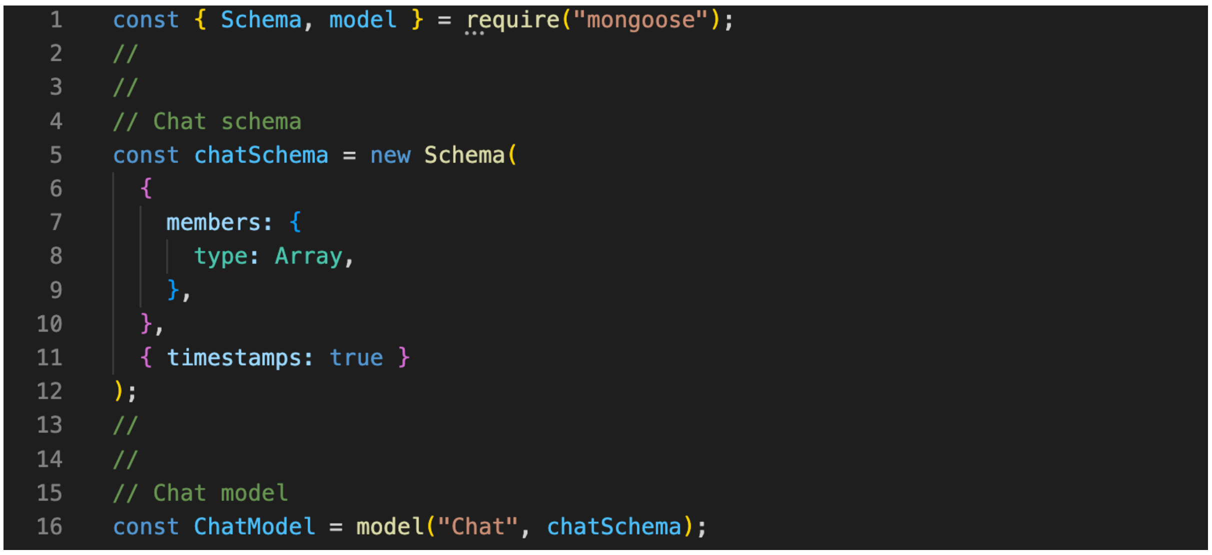
12.1 Chat Router and Controllers

Associates the route with the **chatRouter**. This suggests that the application has functionality related to chat, and requests to **http://localhost:4000/api/chat** will be handled by the routes defined in chatRouter using express-router. **(Express.js, 2022).**



12.2 Chat Schema and Model Definition with Mongoose

The provided code defines a Mongoose schema and model for a chat within a chat application. The '**chatSchema**' specifies a structure with a single field, 'members', which is of type Array. This field is designed to store an array of member IDs representing users participating in the chat. The schema also includes timestamps to automatically record the creation and update times of chat instances. **(Mongoose, 2023).**



```javascript
const { Schema, model } = require("mongoose");
//
//
// Chat schema
const chatSchema = new Schema(
  {
    members: {
      type: Array,
    },
  },
  { timestamps: true }
);
//
//
// Chat model
const ChatModel = model("Chat", chatSchema);
```

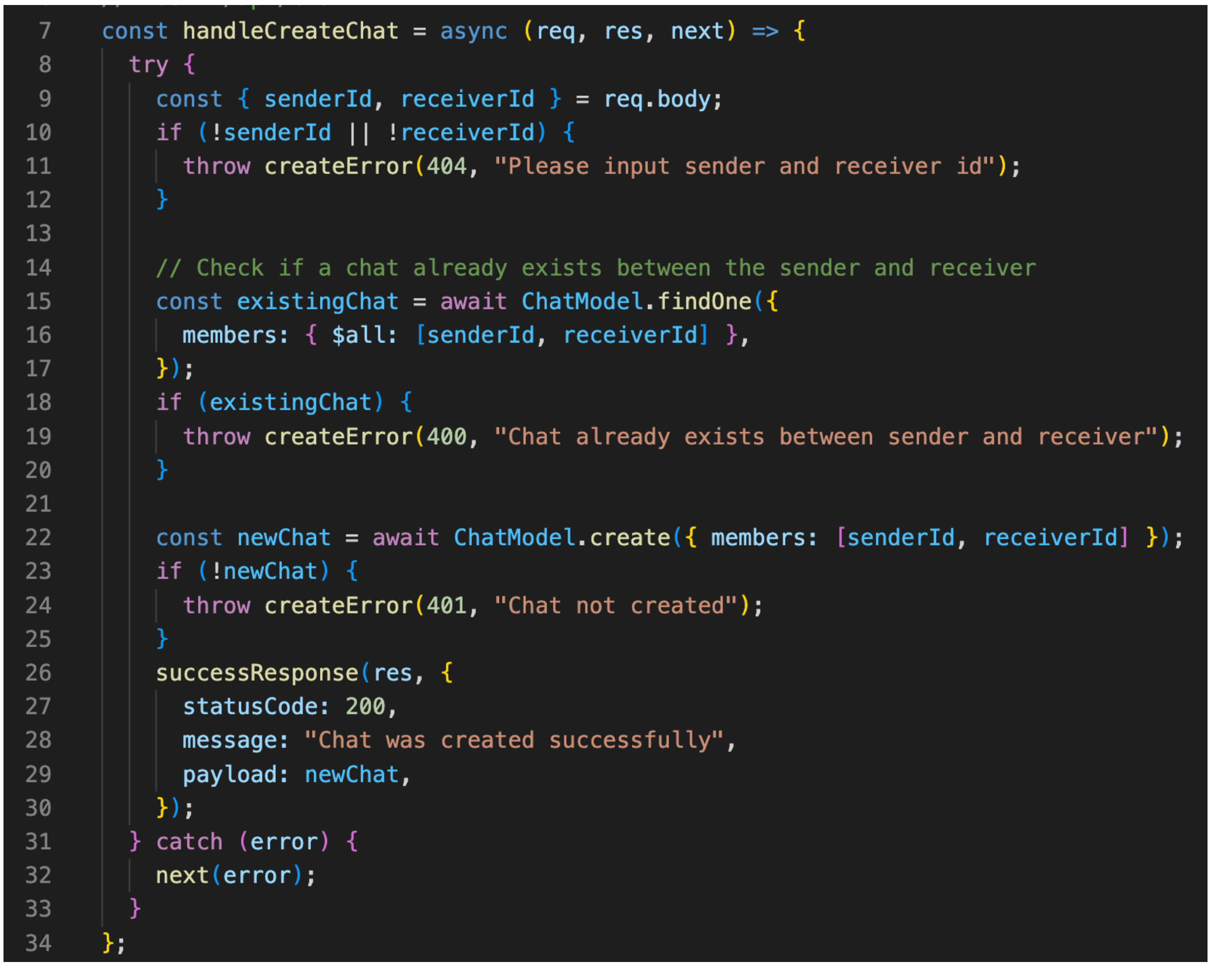
Figure 28: An example of chat schema (Source Code, 2023)

Subsequently, the 'ChatModel' is created using Mongoose's 'model' function, associating it with the "Chat" collection in the MongoDB database **(MongoDB, 2023).** This model is then exported for use in other parts of the application. The structure conforms to the principles of MongoDB, where each chat is uniquely identified by an automatically generated ObjectId and includes the array of member IDs.

This schema and model serve as a foundation for managing and persisting chat-related data, allowing the application to store information about chat memberships and track when these records were created or updated.

12.3 Real-Time Chat Creation with Error Handling

The **handleCreateChat** function in the ‘chatController.js’ file serves as an endpoint for creating new chat instances in a real-time messaging system. It requires the ‘senderId’ and ‘receiverId’ to be provided in the request body. The function begins by checking the existence of a chat between the specified sender and receiver using the ‘ChatModel’. If a chat already exists, it throws a 400-error indicating that a chat is already established between the specified users. If not, a new chat is created with the provided member IDs using the ‘ChatModel.create’ method. Upon successful creation, a 200-status response is sent, confirming the successful creation of the chat, and including the new chat instance in the payload. **(Express.js, 2022).**



```
const handleCreateChat = async (req, res, next) => {
  try {
    const { senderId, receiverId } = req.body;
    if (!senderId || !receiverId) {
      throw createError(404, "Please input sender and receiver id");
    }

    // Check if a chat already exists between the sender and receiver
    const existingChat = await ChatModel.findOne({
      members: { $all: [senderId, receiverId] },
    });
    if (existingChat) {
      throw createError(400, "Chat already exists between sender and receiver");
    }

    const newChat = await ChatModel.create({ members: [senderId, receiverId] });
    if (!newChat) {
      throw createError(401, "Chat not created");
    }
    successResponse(res, {
      statusCode: 200,
      message: "Chat was created successfully",
      payload: newChat,
    });
  } catch (error) {
    next(error);
  }
};
```

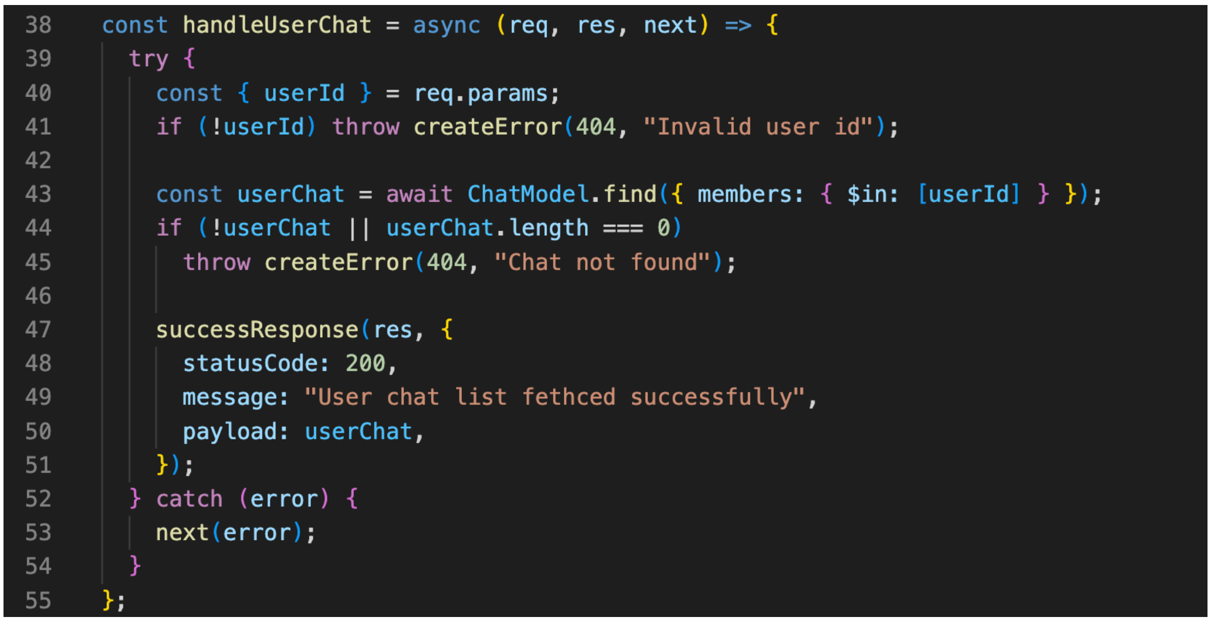
Figure 29: An example of real-time chat creation handler (Source Code, 2023)

This functionality enhances the user experience by preventing the creation of duplicate chats between the same users and ensuring that users can seamlessly initiate new chats. It provides a robust foundation for managing real-time communication within the application.

Additionally, the error handling mechanisms ensure that appropriate error responses are provided in case of invalid inputs or failed chat creation attempts, enhancing the reliability of the messaging feature.

12.4 User-Specific Chat Retrieval

The **handleUserChat** function is designed to retrieve the chat list of a user identified by their ID. It begins by extracting the user ID from the request parameters and checking its validity; if the ID is missing or invalid, the function throws a 404-error. Subsequently, the function queries the ChatModel to find all chat instances where the specified user ID is included in the 'members' array. If no chats are found, it throws a 404-error indicating that the user's chat list is not available **(Express.js, 2022).**



```
const handleUserChat = async (req, res, next) => {
  try {
    const { userId } = req.params;
    if (!userId) throw createError(404, "Invalid user id");

    const userChat = await ChatModel.find({ members: { $in: [userId] } });
    if (!userChat || userChat.length === 0)
      throw createError(404, "Chat not found");

    successResponse(res, {
      statusCode: 200,
      message: "User chat list fethced successfully",
      payload: userChat,
    });
  } catch (error) {
    next(error);
  }
};
```

Figure 30: An example of retrieval chat list (Source Code, 2023)

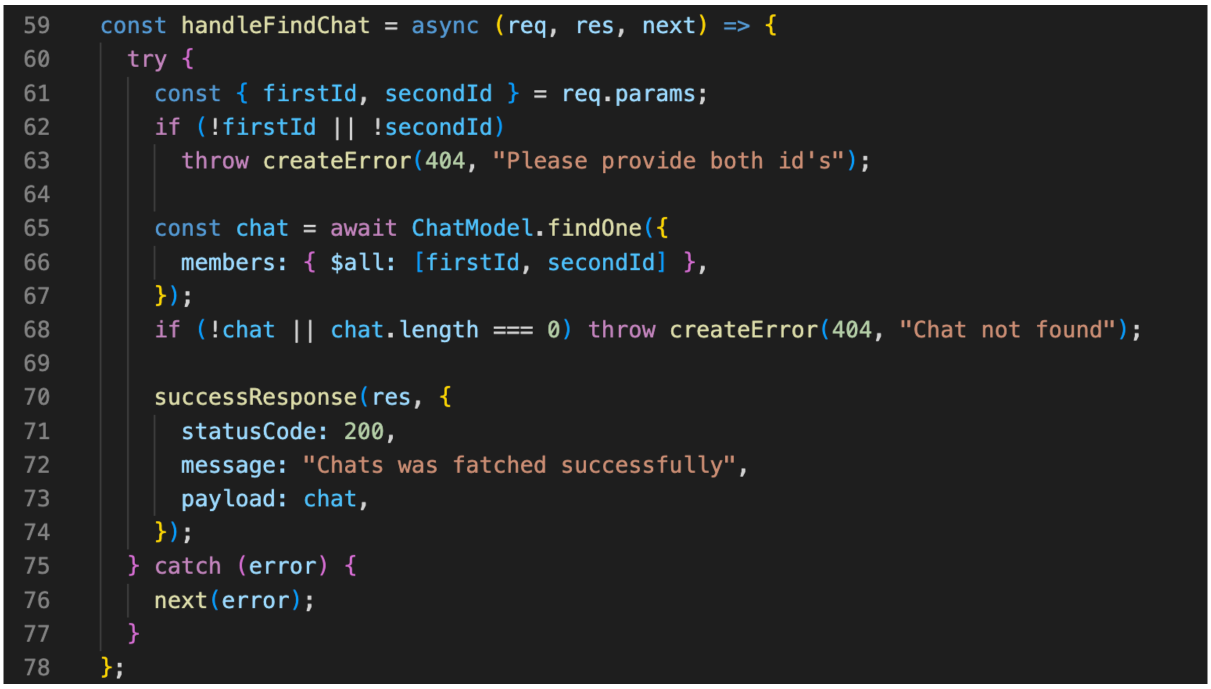
Upon successfully fetching the user's chat list, the function constructs a success response containing a 200-status code, a message indicating the successful retrieval of the chat list from MongoDB **(MongoDB, 2023),** and the payload, which consists of the retrieved chat instances. This response is then sent to the client. In case of any errors during the process, the function passes the error to the next middleware for further handling**.**

Overall, this function serves as an endpoint for obtaining the chat list of a specific user, facilitating efficient communication and interaction within a chat application.

12.5 Chat Retrieval and Search Functionality

The **handleFindChat** function is designed to find and retrieve a specific chat between two users identified by their IDs. It begins by extracting the first and second user IDs from the request parameters and checking their presence; if either ID is missing, the function throws a

404 error, prompting the client to provide both user IDs. Subsequently, the function queries the ChatModel to find a chat instance where both the first and second user IDs are present in the 'members' array. If no such chat is found, the function throws a 404 error, indicating that the requested chat could not be located.



```
const handleFindChat = async (req, res, next) => {
  try {
    const { firstId, secondId } = req.params;
    if (!firstId || !secondId)
      throw createError(404, "Please provide both id's");

    const chat = await ChatModel.findOne({
      members: { $all: [firstId, secondId] },
    });
    if (!chat || chat.length === 0) throw createError(404, "Chat not found");

    successResponse(res, {
      statusCode: 200,
      message: "Chats was fatched successfully",
      payload: chat,
    });
  } catch (error) {
    next(error);
  }
};
```

Figure 31: An example of search functionality (Source Code, 2023)

Upon successfully finding the chat from MongoDB **(MongoDB, 2023)**, the function constructs a success response with a 200-status code, a message indicating the successful retrieval of the chat, and the payload, which consists of the located chat instance. This response is then sent to the client. In case of any errors during the process, the function passes the error to the next middleware for further handling.

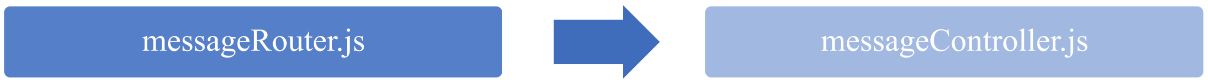
In summary, this function serves as an endpoint for locating and fetching a specific chat between two users, providing a targeted and efficient way to access relevant chat information within a chat application. **(Express.js, 2022).**

13. Message Controller for Chat Application

The **messageController.js** file handles the creation and retrieval of messages in a chat application. In the "handleAddMessage" function, messages are added to the database, requiring information such as the chat ID, sender ID, and the message text. It checks for the existence of essential parameters and returns a success response with the added message payload. On the other hand, the "handleGetMessage" function retrieves messages associated with a specific chat ID using the "api/messages/:chatId" endpoint. The retrieved messages are then returned in the response with a success status code. The module exports these functions to be used in the overall chat application, ensuring proper handling of message-related operations.

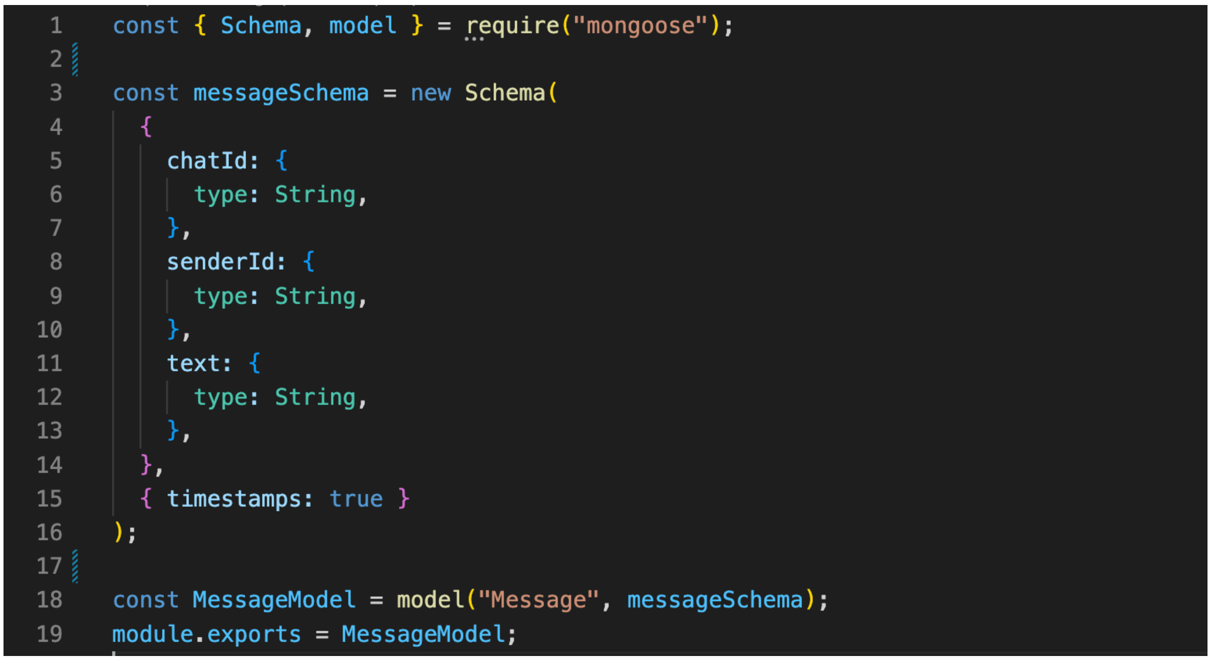
13.1 Message Router and Controllers

Associates the route with the messageRouter. Requests to **http://localhost:4000-/api/messages** will be directed to the routes defined in messageRouter.js file using express router **(Express.js, 2022).** This likely implies handling messages or conversations within the application.



13.2 Message Schema and Model Definition with Mongoose

The provided code defines a Mongoose **(Mongoose, 2023)** schema and model for messages within a chat application. The ‘messageSchema’ outlines the structure with three fields: ‘chatId’, representing the identifier of the chat to which the message belongs; ‘senderId’, indicating the sender of the message; and ‘text’, containing the actual content of the message, represented as a string. The schema includes timestamps to automatically record the creation and update times of message instances.



```
const { Schema, model } = require("mongoose");

const messageSchema = new Schema(
  {
    chatId: {
      type: String,
    },
    senderId: {
      type: String,
    },
    text: {
      type: String,
    },
  },
  { timestamps: true }
);

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

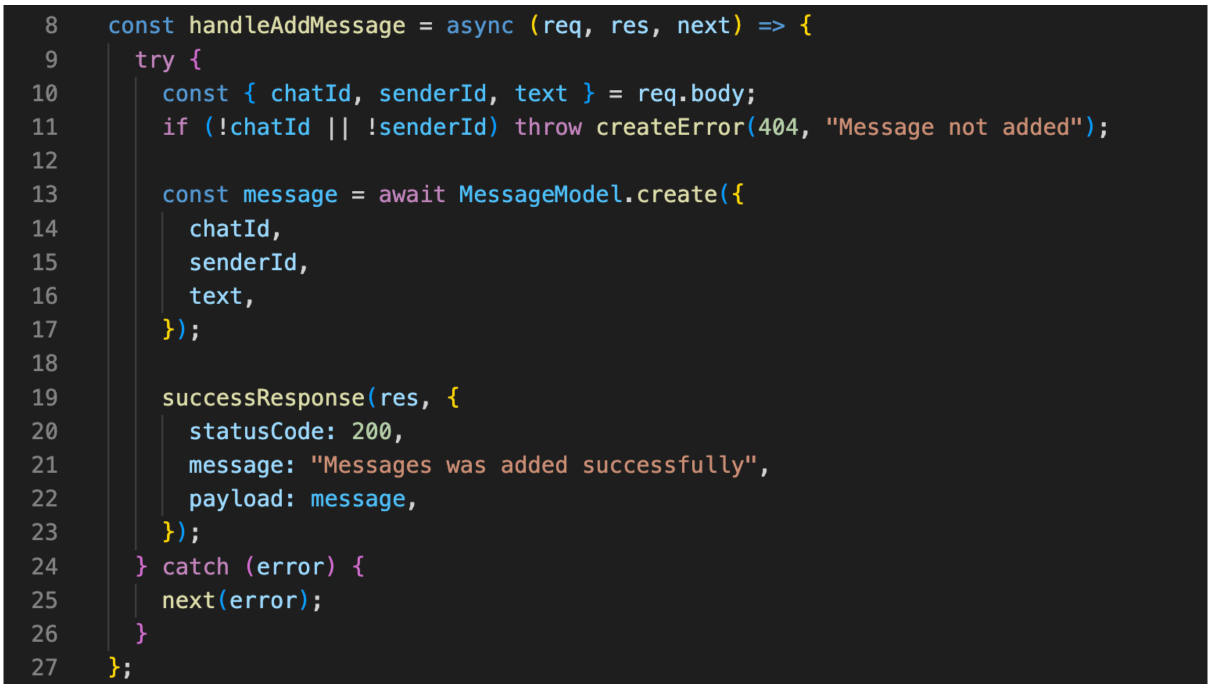
Figure 32: An example of message schema (Source Code, 2023)

Subsequently, the ‘MessageModel’ is created using Mongoose's ‘model’ function, associating it with the "Message" collection in the MongoDB database **(MongoDB, 2023).** This model encapsulates the logic for interacting with messages in the application and is exported for use in other parts of the codebase.

The schema and model facilitate the storage and retrieval of message-related data, providing a standardized structure for messages in the database. The timestamps enhance the model by automatically tracking when messages are created or updated, offering valuable temporal information for the chat application.

13.3 Adding Messages to Chat

The handleAddMessage function in the message controller is responsible for adding new messages to a chat in a chat application. The endpoint receives the necessary data in the request body, including the chatId, senderId, and the text of the message. The function first checks whether the essential data, chatId and senderId, is present; if not, it throws a 404 error, indicating that the message cannot be added. **(Express.js, 2022).**



```javascript
const handleAddMessage = async (req, res, next) => {
  try {
    const { chatId, senderId, text } = req.body;
    if (!chatId || !senderId) throw createError(404, "Message not added");

    const message = await MessageModel.create({
      chatId,
      senderId,
      text,
    });

    successResponse(res, {
      statusCode: 200,
      message: "Messages was added successfully",
      payload: message,
    });
  } catch (error) {
    next(error);
  }
};
```

Figure 33: An example of adding message to chat (Source Code, 2023)

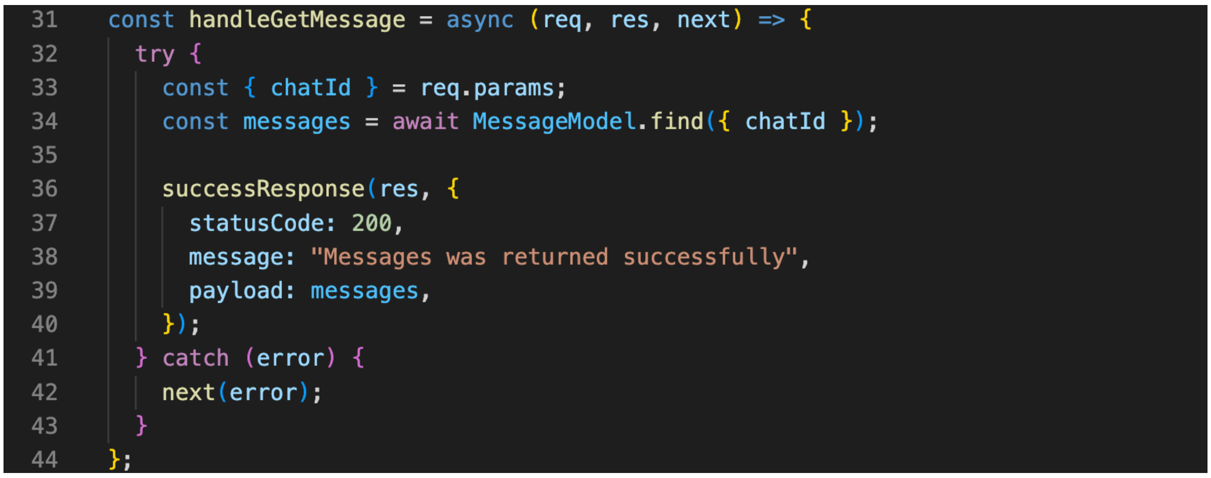
Upon successful validation, the function utilizes the MessageModel.create method to create a new message in the database with the provided details. The created message is then returned as part of a success response, including a 200-status code, a message indicating successful addition, and the payload containing the newly created message.

In case of any errors during the process, the function passes the error to the next middleware, ensuring proper error handling and propagation to the error-handling middleware in the application. This modular approach facilitates the separation of concerns and promotes maintainability in the overall structure of the chat application.

13.4 Retrieve Messages from Chat

The provided code defines a function, ‘handleGetMessage’, which is designed to handle a GET request to retrieve messages associated with a specified chat identifier (‘chatId’). Within a try-catch block, the code attempts to find messages in the MongoDB database using the ‘MessageModel’ model. The ‘find’ method is employed with the ‘chatId’ parameter to filter messages specific to the provided chat.

If the retrieval is successful, the function sends a success response back to the client with a status code of 200, a success message indicating that the messages were returned successfully, and the retrieved messages as the payload. On the other hand, if an error occurs during the process, it is caught in the catch block, and the ‘next’ function is invoked to pass the error to the next middleware or error-handling function **(Express.js, 2022).**



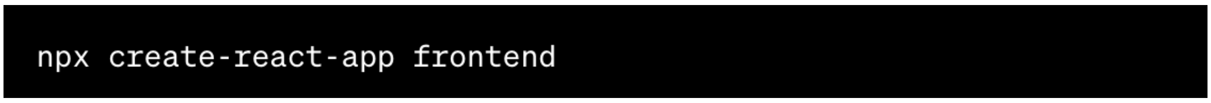
```
const handleGetMessage = async (req, res, next) => {
  try {
    const { chatId } = req.params;
    const messages = await MessageModel.find({ chatId });

    successResponse(res, {
      statusCode: 200,
      message: "Messages was returned successfully",
      payload: messages,
    });
  } catch (error) {
    next(error);
  }
};
```

Figure 34: An example of retrieve message from chat (Source Code, 2023)

In essence, this function serves the purpose of fetching messages related to a particular chat and responding with the retrieved messages or handling errors that may occur during the process.

14. Create the React App

Inside the "social-media" folder, use **npx create-react-app** to generate a new React app named "frontend" **(React, 2022).**



```
npx create-react-app frontend
```

Figure 35: Create the React App (Source Code, 2023)

This will create a new folder named "frontend" within your "social-media" folder, containing the necessary files and structure for a React app. Now Move into the "frontend" folder using command "**cd frontend**"

Run the development server to see your React app in action using command "**npm start**". This command will start the development server, and you can access your app at http://localhost:3000 in your web browser. **(npm, 2023).**

14.1 Simplifying State Management Using Redux Toolkit

Redux Toolkit is a set of tools and utilities provided by the Redux team to simplify and streamline the process of managing state in React applications that use Redux for state management. It aims to address common challenges and boilerplate associated with setting up a Redux store and managing actions and reducers. The toolkit provides a standardized way of organizing and working with Redux code, making it more maintainable and scalable.

One key feature of Redux Toolkit is the **configureStore** function, which simplifies the creation of a Redux store. It automatically sets up the store with commonly used middleware, such as Redux Thunk for handling asynchronous actions, and enables developers to write less boilerplate code. Additionally, the toolkit introduces the createSlice function, a more concise way to define reducers and actions associated with a specific piece of state.

Redux Toolkit encourages best practices by promoting the use of immutable updates and a more modular structure for Redux code. The included utilities aim to enhance developer productivity, reduce boilerplate, and make the codebase more readable. Overall, Redux Toolkit is designed to make Redux more approachable for developers, especially those newer to the ecosystem, while offering advanced capabilities for experienced Redux users.

In summary, Redux Toolkit is a collection of tools and abstractions that simplifies the process of working with Redux in React applications. It provides a more ergonomic and efficient way to manage state, reduces boilerplate code, and promotes best practices for a more maintainable and scalable codebase. **(Reduc toolkit, 2023).**

14.2 Usages of Redux Toolkit

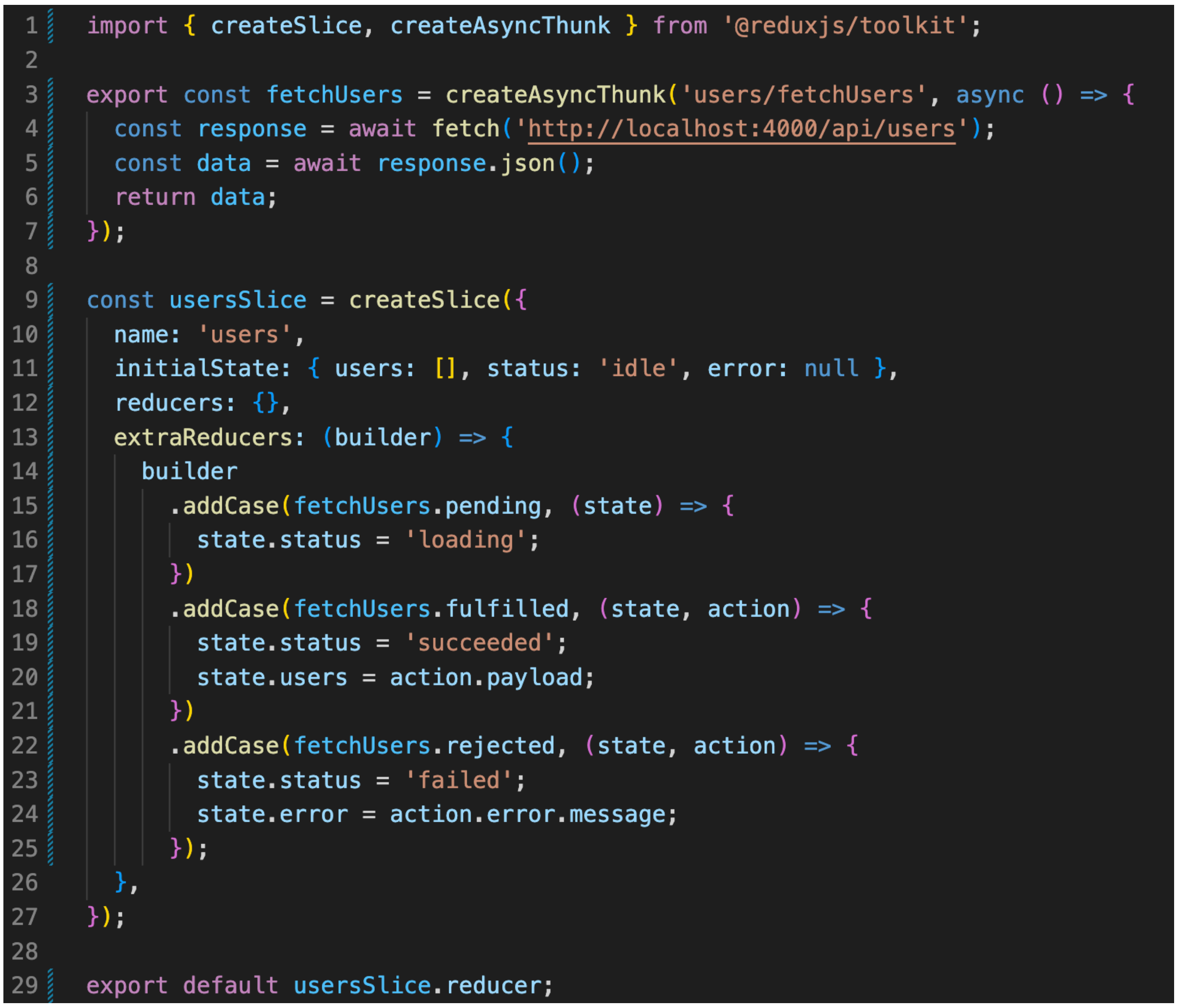
Make sure you have Redux Toolkit installed. If not, Install Dependencies.

```
npm install @reduxjs/toolkit react-redux
```

14.2.1 Efficient User Data Management with Redux Toolkit

This is a part of a Redux Toolkit setup using the **createSlice** and **createAsyncThunk** utilities. In the '**usersSlice.js**' file, the code defines a slice of the Redux store dedicated to managing user-related data. The main asynchronous operation is fetching user data from the server using the 'createAsyncThunk' named 'fetchUsers'. This thunk dispatches actions for different stages of the asynchronous operation, such as 'pending' when the data is being fetched, 'fulfilled' when the operation is successful, and 'rejected' when an error occurs.

The 'createSlice' function is used to create a Redux slice that includes the initial state, reducers, and extra reducers. In this case, it defines an initial state with an empty array for users, a status field to track the loading state, and an error field for potential errors during data fetching. The **extraReducers** section listens to the actions dispatched by the 'fetchUsers' thunk and updates the state accordingly. When the data fetching is pending, the status is set to 'loading,' and upon successful completion, the user data is stored in the state. If an error occurs during the process, the status becomes 'failed,' and the error message is recorded. **(Reduc toolkit, 2023).**



```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';

export const fetchUsers = createAsyncThunk('users/fetchUsers', async () => {
  const response = await fetch('http://localhost:4000/api/users');
  const data = await response.json();
  return data;
});

const usersSlice = createSlice({
  name: 'users',
  initialState: { users: [], status: 'idle', error: null },
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(fetchUsers.pending, (state) => {
        state.status = 'loading';
      })
      .addCase(fetchUsers.fulfilled, (state, action) => {
        state.status = 'succeeded';
        state.users = action.payload;
      })
      .addCase(fetchUsers.rejected, (state, action) => {
        state.status = 'failed';
        state.error = action.error.message;
      });
  },
});

export default usersSlice.reducer;
```

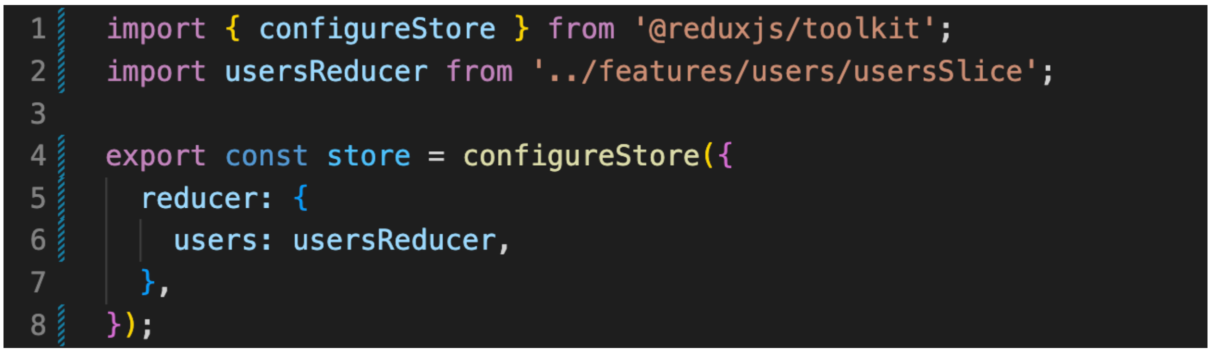
Figure 36: An example of using redux toolkit (Source Code, 2023)

This setup follows the best practices of Redux Toolkit, providing a more concise and readable way to manage asynchronous operations and state in a React application **(React, 2022),** ultimately simplifying the complexities associated with Redux boilerplate code**.**

14.2.2 Redux Store Configuration with Async Operations for User Data

This code snippet showcases the creation of a Redux store using the 'configureStore' function from the **@reduxjs/toolkit** library. The store is configured with a single reducer, **usersReducer**, which is imported from the 'usersSlice' module. This module, as defined in the previous code, manages the state for user data, including handling asynchronous operations such as fetching user data from an API. **(Reduc toolkit, 2023).**

The '**configureStore**' function provides a simplified way to set up a Redux store by automatically configuring the store with common middleware, including the Redux DevTools Extension for debugging. In this case, the store is initialized with a reducer that is responsible for managing the state related to user data.



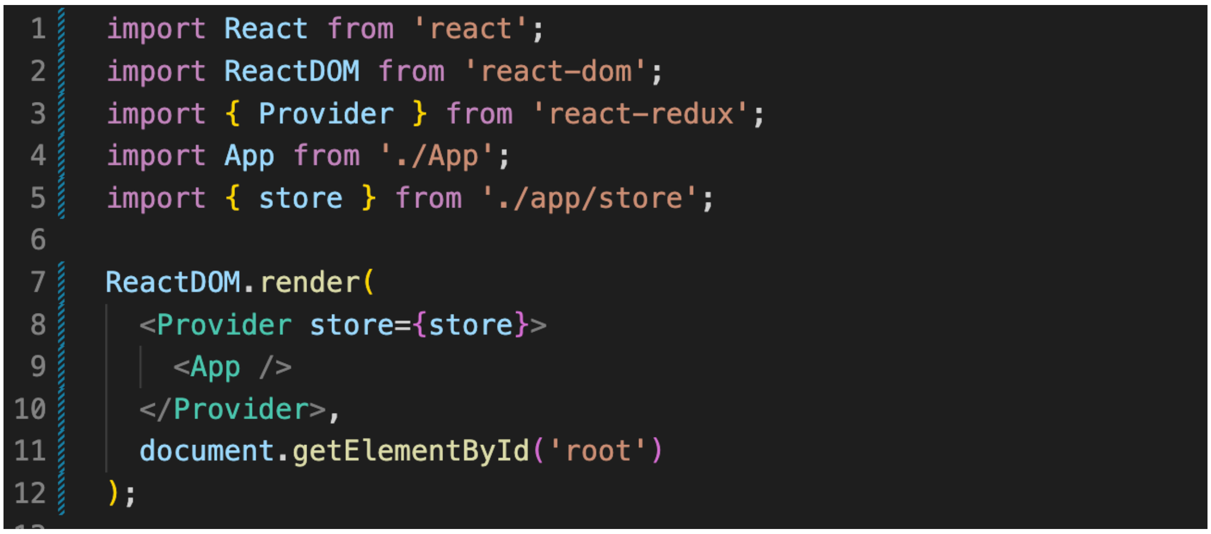
```
import { configureStore } from '@reduxjs/toolkit';
import usersReducer from '../features/users/usersSlice';

export const store = configureStore({
  reducer: {
    users: usersReducer,
  },
});
```

Figure 37: Configure @reduxtoolkit store (Source Code, 2023)

The separation of concerns is evident in this code, adhering to best practices in Redux development. The 'usersSlice' module handles the logic related to fetching users, and the store configuration centralizes the state management for the users' feature. This streamlined approach enhances code maintainability and readability, making it easier to scale the application as needed. **(React, 2022).**

14.2.3 React Application Initialization with Redux Store Integration

The code snippet sets up the Redux store provider for a React application. The '<Provider>' component is imported from the 'react-redux' library and is utilized to wrap the main application component '<App />'. By passing the 'store' as a prop to the provider, the entire application gains access to the Redux store. This ensures that the state managed by Redux is available throughout the component tree, allowing components to connect to the store and access or dispatch actions. **(Reduc toolkit, 2023).**



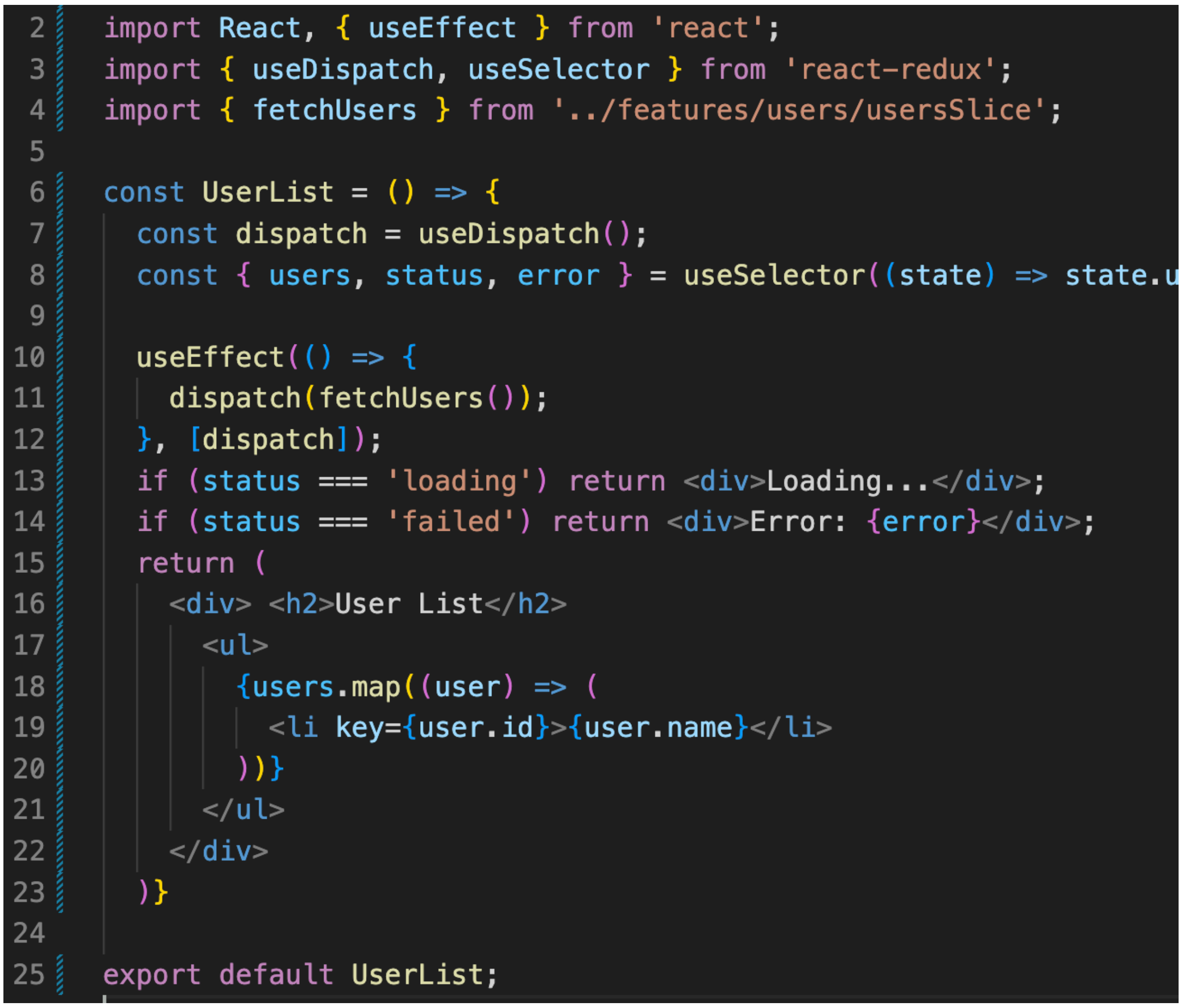
```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import App from './App';
import { store } from './app/store';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

Figure 38: Redux store integration (Source Code, 2023)

In this specific context, the Redux store is configured in the **'store.js'** file, where the 'configureStore' function from the '@reduxjs/toolkit' library is used to create the store. The store is then imported and provided to the entire application through the '<Provider>' component in the 'index.js' file. This setup enables seamless integration of Redux with React, allowing components to interact with the centralized store and manage application state efficiently. **(React, 2022).**

14.2.4 React User List Component with Redux Toolkit Integration

The ‘UserList.js’ file is a React component that utilizes the Redux Toolkit for managing state. The component is responsible for fetching a list of users from an API endpoint using the ‘fetchUsers’ async thunk defined in the ‘usersSlice.js’ file. It uses the ‘useDispatch’ and ‘useSelector’ hooks provided by React Redux to interact with the Redux store. **(Reduc toolkit, 2023).**



```
import React, { useEffect } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { fetchUsers } from '../features/users/usersSlice';

const UserList = () => {
  const dispatch = useDispatch();
  const { users, status, error } = useSelector((state) => state.u

  useEffect(() => {
    dispatch(fetchUsers());
  }, [dispatch]);
  if (status === 'loading') return <div>Loading...</div>;
  if (status === 'failed') return <div>Error: {error}</div>;
  return (
    <div> <h2>User List</h2>
      <ul>
        {users.map((user) => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    </div>
  )}

export default UserList;
```

Figure 39: Fetch user list using redux toolkit (Source Code, 2023)

The ‘useEffect’ hook is employed to dispatch the ‘fetchUsers’ action when the component mounts. This asynchronous action triggers an API call to retrieve user data. The component's render logic then responds to different states of the Redux store, displaying loading text while data is being fetched, rendering an error message if the fetch fails, and rendering the user list once the data is successfully fetched. **(React, 2022).**

The user list is displayed as an unordered list (‘ul’) where each user's name is rendered within a list item (‘li’). The component relies on the Redux store's state to dynamically render content based on the status of the API request, providing a responsive and user-friendly experience.

In summary, ‘UserList.js’ demonstrates a React component that seamlessly integrates with Redux Toolkit, leveraging its async thunk middleware to handle API calls, and effectively managing the UI based on the state of the Redux store. **(Reduc toolkit, 2023).**

15 Socket.IO Server for Real-Time Communication

Socket.IO is a JavaScript library that enables real-time, bidirectional communication between clients and servers. It's particularly well-suited for building applications that require instant updates or live interactions, such as chat applications, online gaming, and collaborative tools. Socket.IO achieves this by establishing a WebSocket connection when possible and falling back to other transport mechanisms like long polling in environments where WebSocket is not supported. **(Socket io, 2023).**

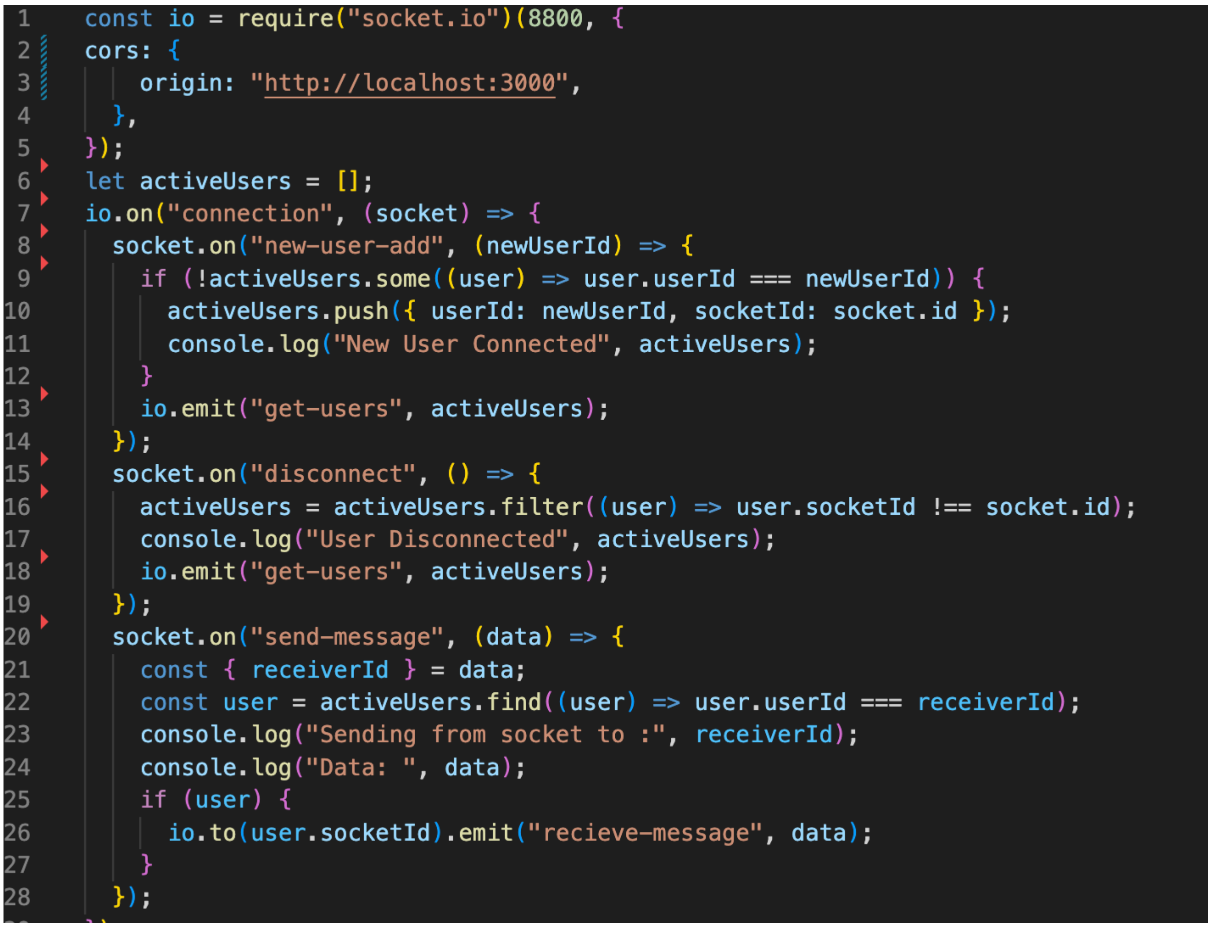
The library is designed to be easy to use and implement. It provides a simple API for both server-side and client-side components, making it accessible for developers of various skill levels. Socket.IO abstracts away the complexity of handling real-time communication protocols, allowing developers to focus on building features that require live updates.

One of Socket.IO's key features is its ability to manage connections and rooms. It allows for the creation of unique rooms, where clients can join and communicate within specific contexts. This makes it scalable for various use cases, such as private messaging or group interactions. Overall, Socket.IO is a powerful tool for enhancing the interactivity and responsiveness of web applications by enabling seamless communication between clients and servers in real-time. **(Node.js, 2023).**

15.1 Real-Time Node.js Server with Socket.IO for User Presence and Messaging

The provided code is a Node.js server using Socket.IO **(Socket io, 2023)** to enable real-time communication between clients. The server listens on port 8800 and allows connections from the specified origin "http://localhost:3000" using CORS settings **(Troy, 2018)**. It maintains an array, ‘**activeUsers**’, to keep track of connected users along with their respective socket IDs.

The Socket.IO event listeners handle various scenarios. When a new user connects (‘new-user-add’ event), the server checks if the user is not already in the ‘activeUsers’ array before adding them. The server then emits a ‘get-users’ event to inform all connected clients about the updated list of active users. **(Node.js, 2023).**



```
const io = require("socket.io")(8800, {
cors: {
    origin: "http://localhost:3000",
  },
});
let activeUsers = [];
io.on("connection", (socket) => {
  socket.on("new-user-add", (newUserId) => {
    if (!activeUsers.some((user) => user.userId === newUserId)) {
      activeUsers.push({ userId: newUserId, socketId: socket.id });
      console.log("New User Connected", activeUsers);
    }
    io.emit("get-users", activeUsers);
  });
  socket.on("disconnect", () => {
    activeUsers = activeUsers.filter((user) => user.socketId !== socket.id);
    console.log("User Disconnected", activeUsers);
    io.emit("get-users", activeUsers);
  });
  socket.on("send-message", (data) => {
    const { receiverId } = data;
    const user = activeUsers.find((user) => user.userId === receiverId);
    console.log("Sending from socket to :", receiverId);
    console.log("Data: ", data);
    if (user) {
      io.to(user.socketId).emit("recieve-message", data);
    }
  });
```

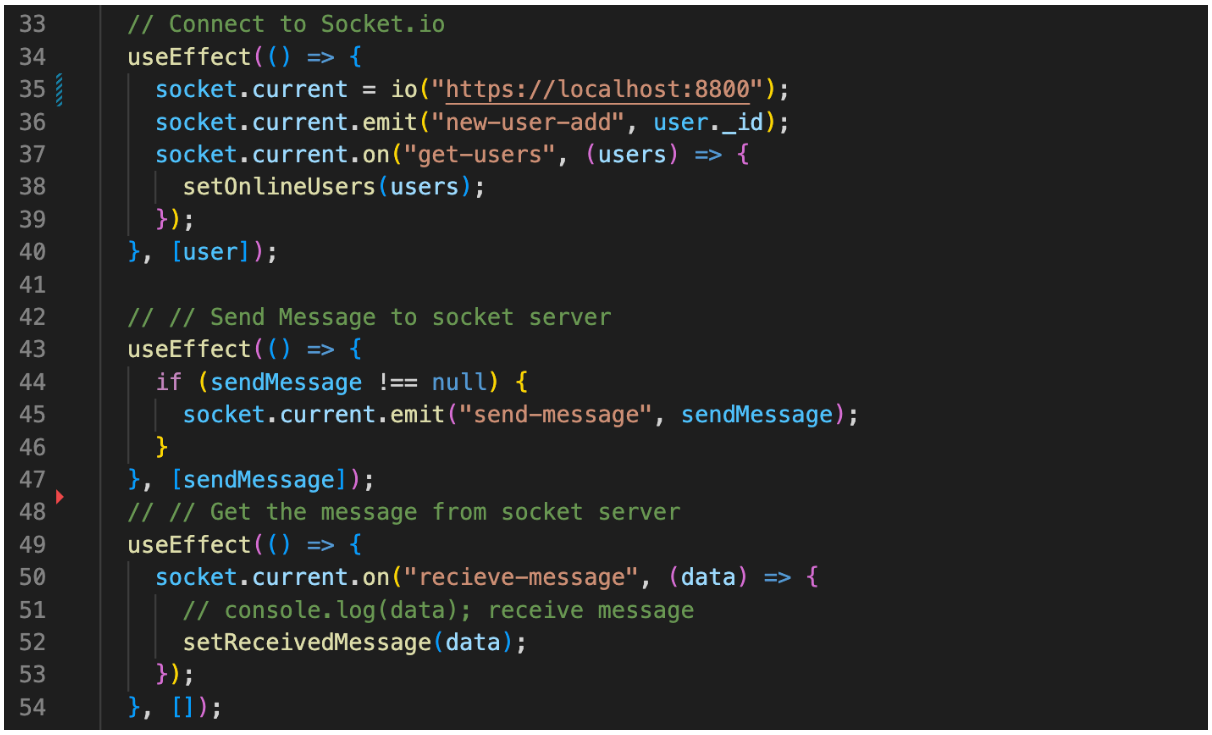
Figure 40: Example of socket.io server setup (Source Code, 2023)

The ‘disconnect’ event is triggered when a user disconnects. In this case, the server removes the disconnected user from the ‘activeUsers’ array and emits the updated list to all clients. Additionally, there's a ‘send-message’ event listener that allows users to send messages. The server identifies the recipient user by their ‘receiverId’ and uses the recipient's socket ID to emit a ‘recieve-message’ event, ensuring that the message is delivered to the correct recipient.

Overall, this server facilitates real-time communication and user tracking, providing a foundation for building features like online user presence and instant messaging in a web application. **(Node.js, 2023).**

15.2 Managing Connections and Messaging

The provided React code snippet utilizes the useEffect hook to manage communication with a Socket.IO server. In the first **useEffect** block, the connection setup is performed. It establishes a socket connection to the server when the component mounts or when the ‘user’ dependency changes. This connection initiation includes emitting a "new-user-add" event with the user's ID to inform the server of the new user and listening for "get-users" events to update the list of online users in the client application. **(React, 2022).**



```
// Connect to Socket.io
useEffect(() => {
  socket.current = io("https://localhost:8800");
  socket.current.emit("new-user-add", user._id);
  socket.current.on("get-users", (users) => {
    setOnlineUsers(users);
  });
}, [user]);

// // Send Message to socket server
useEffect(() => {
  if (sendMessage !== null) {
    socket.current.emit("send-message", sendMessage);
  }
}, [sendMessage]);
// // Get the message from socket server
useEffect(() => {
  socket.current.on("recieve-message", (data) => {
    // console.log(data); receive message
    setReceivedMessage(data);
  });
}, []);
```

Figure 41: Connection socket.io in react app (Source Code, 2023)

Moving on to the second 'useEffect' block, it is dedicated to sending messages. The hook monitors changes in the 'sendMessage' state, and when a new message is present (i.e., 'sendMessage' is not null), it emits a "send-message" event to the Socket.IO server. This mechanism enables the client to send messages to the server, facilitating real-time communication. **(Socket io, 2023).**

The third 'useEffect' block focuses on receiving messages. It listens for "recieve-message" events from the Socket.IO server. When a message is received, the data is processed, and the 'setReceivedMessage' function is invoked to update the state with the received message. This enables the client to react to incoming messages and update the user interface dynamically based on the real-time communication with the server. **(React, 2022).**

16. Final Remark

In conclusion, the development journey of the Social Media Web App presented in this thesis underscores the intricate interplay between backend, frontend, and real-time communication components. The comprehensive exploration began with the establishment of a robust backend infrastructure, delving into the initialization of Node.js, configuration of the Express server, and meticulous design of schemas and controllers for user-centric entities. Crucial aspects such as user management, authentication, social media post handling, and real-time messaging operations were thoroughly examined, providing a holistic understanding of constructing RESTful APIs.

The integration of WebSocket technology brought an additional layer of dynamism to the application, enabling real-time communication for user presence and messaging. The initiation of Node.js for the server and the utilization of socket.io facilitated seamless connections and message transmissions, contributing to an enhanced user experience through instantaneous updates and dynamic interactions.

While the frontend development focused primarily on the utilization of React and essential tools like Redux Toolkit for state management, the emphasis remained on the interaction between the frontend and backend. The integration of Redux Toolkit streamlined state management, and the use of Fetch ensured efficient communication with the RESTful APIs.

Throughout the thesis, the practical implementation was complemented by theoretical underpinnings, providing readers with insights into the decision-making process and the rationale behind technology choices. The thesis aimed to serve as a roadmap for web application developers, offering guidance on navigating the complexities of backend and frontend development, along with the integration of real-time communication features.

Reflecting on the encountered challenges, it is acknowledged that the development process inherently involves overcoming hurdles, be they technical, conceptual, or logistical. The iterative nature of development allowed for the refinement of strategies and solutions, contributing to the project's overall success.

In final remarks, this thesis endeavours to contribute to the ever-evolving landscape of web application development by offering a practical and theoretical guide. The holistic exploration of backend RESTful APIs, frontend development, and real-time communication using WebSocket technology seeks to empower developers with the knowledge and insights necessary to embark on similar endeavours. As technology advances, continued exploration and adaptation remain paramount, and this thesis serves as a foundation for those eager to navigate the intricate path of modern web application development.

17. References:

- **Source Code.** (2023). Retrieved from https://github.com/rana-rayhan/socialMedia-mern-p9

- **Chacon, S & Straub**, B 2014. Pro Git. 2nd Ed. Apress. E-book. URL: https://git-scm.com/book/en/v2 Accessed: 20 March 2022.
- **REST APIs:** Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine. (Roy Thomas Fielding, 2000) https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- **npm**. (2023). About npm. Retrieved from https://docs.npmjs.com/about-npm
- **Mongoose.** (2023). Retrieved from https://mongoosejs.com/
- **Node.js,** c. (2023). About Node.js. Retrieved from https://nodejs.org/en/about
- **JWT**. (2023). Retrieved from Introduction to JSON Web Tokens: https://jwt.io/introduction
- **Troy, G.** (2018). CORS. Retrieved from https://www.npmjs.com/package/cors
- **MongoDB**. (2023). Retrieved from https://www.mongodb.com/docs/
- **bcryptjs**. (2017). Retrieved from https://www.npmjs.com/package/bcryptjs
- **React**. (2022). Retrieved from https://legacy.reactjs.org/
- **Express.js.** (2022). Retrieved from https://expressjs.com/
- **Express rate limit.** (2023). Retrieved from https://express-rate-limit.mintlify.app/overview
- **Morgan**. (2020). Retrieved from https://www.npmjs.com/package/morgan
- **xss clean.** (2023). Retrieved from https://www.npmjs.com/package/xss
- **RegExp.** (2023). Retrieved from Regular expressions: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp
- **Reduc toolkit.** (2023). Retrieved from https://www.npmjs.com/package/@reduxjs/toolkit
- **Socket io**. (2023). Retrieved from https://socket.io/docs/v4/

17.1. Table of Figures and References: