EUROPEAN UNIVERSITY OF LEFKE

FACULTY OF ENGINEERING

Graduation Project 2

Chat Application

Daniel Chanda

194127

With the goal to look into real-time communication through the use of modern technology, I, a student passionate about web development, built this project. The goal of this project is to use the MERN stack (MongoDB, Express.js, React, and Node.js) to build a real-time messaging application. The objective of this project was to develop a chat interface that was both responsive and easy to use, enabling users to send and receive messages rapidly. This application showcases the proficiency of JavaScript frameworks and libraries in creating interactive web apps, thanks to features like Zustand for state management and Socket.io integration for real-time updates.

Supervisor

Zafer Eranel

Publish Date

19/03/2024

Table Of Contents

1.Introduction 1
1.1 Problem definition 1
1.2 Goals and objectives 3
2. Literature Survey 4
3. Background Information 6
3.1 Required & Used software 6
4. Design Documents 8
4.1 Use case flow diagram 8
5. Methodology 8
6. Conclusion 33
6.1 Benefits 33
6.2 Ethics 35
6.3 Future Works 37
7. References 37

1.Introduction

1.1 Problem definition

This is the project for a real time online chat application which will provide the users with the feature of real time chat using socket. io library. A few of the issues with the conventional online applications implement live messages depending on manual refreshes that slow the procedure. In other words, the more difficult aspect is to use variables of socket type. Expected options made available by io include; Real-time options to deliver message changes without having to refresh the page. To enhance user experience and to prevent validation messages during conversations in the application, the application ensures user login and still saves the user's details such as usernames and emails within the session.
This project will in some way tackle this, making use of tools like Zustand for state-management and utilities like TailwindCSS together with other libraries to make it a functional chat application displaying the capability of contemporary web development technologies

Why this project is needed by user?

1. **Real time communication:** One major factor is the rapid and flawless transmission of information by the users without any break. This application's focuses on speeding up occasions for sending and receiving messages so that conversations are productive and the decisions made swiftly..

2. **Privacy and Security**: One-to-one chat enables the users to have a personal conversation where the recipients are the only one who will see the chat..

3. **User Engagement**: By letting users know when their contacts are accessible, the online status feature encourages prompt communication and raises user engagement.

4. **Ease of use**: The user search tool assistance in the discovery and connecting with other users that can lead to a message being started.

5. **Versatility**: For communications with friends, at workplace, or for any kind of business, transactional or marketing language need the program has it in all categories of communications..

6. **Modern experience**: Based on modern technologies, the app provides a seamless, fast, intuitive, and aesthetically pleasing interface that is expected for contemporary users of communication applications.

Why it will be useful?.

As the project's developer, I realize how important it is to create a seamless and efficient communication tool for the everyday user of the internet. Here is how my project will be valuable:

1. **Enhanced communication**: It fosters effective and immediate contact and is very relevant for interpersonal and business interactions; thus it ensures that messages are transmitted and received in real time.
2. **User convenience**: The communication process is also very easy due to factors such as; Online Status and; User Search where users can easily look for and interact with other people.
3. **Privacy Assurance**: The user-to-user chat ensures that private talks are kept confidential, providing users with a safe space to discuss or talk about sensitive topics.
4. **Accessibility**: The design of the application is aimed at being user-friendly, thus enabling a wide range of users with varying technical abilities to use it.

How it works

Here is an overview of how the app functions:

- User Registration: In order to create an account, the user must provide their email address, username, and password.
    - If the user leaves any of the required fields empty, they will be prompted to fill in said field(s) with the correct format(e.g. enter a valid email address).
    - If the user enters incorrect format of details(e.g. password and confirm password are not matching up), the user will receive an error telling them to check the details they have entered into the required field.
    - If all the details are correct, they will be redirected to the chats page. Their details will be stored securely in the database.
- User Login:
    - If the user already has an account, they can login with their existing credentials to access their account
    - Said details will be checked and compared with the already present information to authorise and authenticate the user to make sure it is the right user logging in to their account. If yes, they will be redirected to the chats page and if not, they will be prompted to check their details again to see if they have entered either username or password wrongly.
- Direct messaging: Once on the landing page, the user will be able to send messages to users who are online (online users will have a little green dot showing them as online)
- Search feature: If the logged in user wishes to find a specific contact in a large list of of them, the user will make use of the search bar and search for their desired recipient to which they can send messages to, be it in a new chat or an existing conversation for which the messages will be loaded up from where they left off
- Secure Communication: The application uses end-to-end encryption for all messages and files transferred to guarantee data protection and privacy.

- Logging out: Once the user has already conducted their business and discussions with their peers/contacts, they will be able to logout of their chat space and continue their conversations from where they left off with their messages saved in the database.

Example-Problems :

- Delayed Communication Tools: Conventional messaging services may be slow. The application supports real-time texting, which ensures that messages are delivered and received instantly.
- Difficulty in finding other users: The search users functionality enables users to quickly find their desired contacts to chat with
- Privacy and security concerns: This application will handle data privacy and security concerns by ensuring the private direct messages that users send to one another remains confidential and secure.
- Online visibility: This is where the online feature comes in. It will indicate whether a user is available to receive a message or not, allowing users the ability and luxury of texting others at the right time
- Message delivery concerns: Some messaging services fail to send messages reliably. My application will address this in making sure that the messages are saved in the database and delivered, meaning even if the recipient is unavailable for a given period of time, they would still receive the message and find it once they get back online.
- Many chat apps are unable to handle the increasing number of interested clientele. My application is created with scalable technologies to ensure that this problem does not hinder its functionalities and main purpose of user communication.

This MERN chat project seeks to provide a complete and purpose-built solution for small teams and study groups, tackling typical communication issues and improving collaboration in educational and project-based environments. By focusing on particular goals and addressing relevant issues, this application will provide a helpful tool for users to communicate, collaborate, and work more efficiently together..

1.2 Goals and objectives

- **Facilitate seamless communication:** Allow users to connect effectively through direct messaging on a unified platform for all to use and access.
- **Reliable message delivery**: Ensure that all communications are delivered and kept reliably, even when the users are offline
- **Ensure data privacy and safety:** Implement robust safety precautions, such as end-to-end encryption, to protect user data and preserve their privacy within the chat application.
- **Robust backend infrastructure:** Create a robust backend using Node.js, Express and MongoDB along with socket.io to support the app's features and ensure it runs smoothly
- Create an intuitive and simple interface that is accessible to users of all technical skill levels.

- Create an application that makes full use for the technologies and tools selected to support the increasing number of users without having to compromise on app performance.
- Making the application responsive to any device screen size, ensure its use across different platforms.
- Develop a dependable search functionality that allows users to efficiently find their targeted message recipient.

Project Purpose

I would love for this MERN chat app to be a create and extremely secure communication devices that everyone can build and interact with their respective users similarily the way people used to socialize in some cafe early morning. This app was designed for the purpose of a project that fits into my knowledge of MERN stack in terms of building apps at web scale and problem solving as well as building an application with real world usage.

To do this I used Socket.io for real-time communication and sending messages, MongoDB to store data, and React for the user interface. This application keeps the information of the users private and secure while making sure that the environment is safe enough for sensitive interactions to take place. The simplicity in design increases the usability of the platform such that any person can use it regardless of their level of skills in technology. This project additionally has scalability which allows it to accommodate more individuals without compromising on its speed even when they increase sending or receiving many messages at once.

In summary; the system provides a trustworthy and safe messaging application that meets people's communication needs by being reliable. It is meant to enhance person to person communication efficiency while also availing a service for customers who want instant interactivity

2. Literature Survey

The feature of real-time chat applications enable users to instantly communicate with others which can also become the reason for their growing importance. In order to be able to deliver and receive messages instantly without the need of refreshing page, these apps are using technologies such as WebSockets for real-time messaging. An interaction that fast enhance user engagement as all the discussions are now more fluid and interactive, which will then results in a smooth communication like it is when you meet someone. This bibliographic essay reviews previous relevant projects, highlights the strengths and weaknesses of those models, and identifies unique aspects of ConvoCafe.

I will compare ConvoCafe to widely used applications like WhatsApp, Slack, and Microsoft Teams, emphasizing the project's unique features.

1. **WhatsApp**

WhatsApp is a well-known messaging app that permits users to communicate text messages, voice messages, and also engage in audio and video communications. It utilizes end-to-end cryptography for managing the security and privacy of users' communication. WhatsApp makes use of XMPP to message and WebRTC to make audio and video calls. Additionally, users can partake in group chats, share media, and view status updates.

ConvoCafe, much like WhatsApp, allows rapid messaging and presents an online status. The difference between ConvoCafe and WhatsApp is the tech behind the scenes. ConvoCafe employs the MERN stack (MongoDB, Express.js, React, and Node.js) and utilizes Socket.IO for its real-time communication, which is unique to ConvoCafe. ConvoCafe has placed a strong emphasis on enhancing the user experience by providing quick search functionality and plans for features to come like video calling and read receipts to create an even better experience for users.

2. **Slack**

Slack is a communication application that mainly deals with workplace collaboration. Slack provides channels for team discussions, direct messaging, and connecting with a wide range of third-party services. Slack provides various types of communication and file sharing in real-time through a combination of WebSockets and HTTP APIs.

For professional and corporate communication, Slack is tailored to include features like threaded conversations and an extensive integration with external tools. ConvoCafe is designed with its simple and user-friendly interface that differentiates it from Slack. ConvoCafe's use of React Context and Zustand for state management ensures a smooth user experience, particularly in comparison to personal communication..

3. **Facebook Messenger**

The application Facebook Messenger makes it possible for its users to send messages to one another in form of text. Furthermore, the applications lets users send pictures, videos, or to have a conversation containing voice or video. Additionally, users are able to utilize Facebook Messenger to play games and to use chatbots/ The app is closely tied to the Facebook platform, relying on user accounts and social networks.

The service offered by ConvoCafe does not differ much from real-time texting as it is also possible to transfer messages in real-time. The one thing that stands out is there commitment to upfront about privacy and addressing concerns about data security. ConvoCafe takes a step further providing JWT (JSON Web Tokens) and bcrypt for password hashing rather than utilizing the Facebook social network. This ensures that users information is private and safe allowing users peace of mind when it involves data exposure.

Problem: ConvoCafe is designed to fulfill the need for a secure, user-friendly, and efficient messaging app that encourages real-time conversation without sacrificing data security, privacy is a key issue and many current platforms have privacy concerns and are in ecosystems that do not allow for user control over data.

Solution: ConvoCafe is a secure chat platform that uses JWT for authorization and bcrypt for password hashing. Utilizing the MERN stack helps backend security, while also using Socket.IO for real-time communication to ensure fast and reliable message delivery. With the use of React Context and Zustand for state management the application runs faster and provides a better user experience, which makes it work well and look simple to use..

Unique features of Convocafe

1. Security focus: Stress on secure user authentication and data privacy
2. Tech stack used: Uses the MERN stack with Socket.IO to provide a reliable and scalable solution.
3. User experience: The interface is simple and straightforward, making navigation and use easy.
4. Future enhancements: Future plans include video calling, file attachments, read receipts, and comprehensive group management capabilities.

3. Background Information

3.1 Required & Used software

- **MONGO DB :**
  Because MongoDB is document based, it allows for further versatility and scalability, making it perfect for things such as chat messages and user data when saved in a JSON format. Storing historical conversations or user data and having them be easily retrieved are good use cases for its format being document-based and good for retrieval and manipulation of data.
- **Express JS :**
  Express.js is a lightweight web application framework based on Node.js that provide many of the functionalities necessary to build APIs and Web Servers. It simplifies managing HTTP requests, routing, and using middleware. This will enable us to develop the backend server for our chat application faster.
- **React.js:** React.Js is a JavaScript framework for developing user interfaces, acknowledged for its component-based architecture and virtual DOM rendering. It makes it easier to create interactive and responsive UI components, allowing you to construct a dynamic and engaging frontend to your chat utility. The important components had to construct the chat software's person interface are constructed the usage of React js.
- **Axios:**

Axios is an HTTP client that uses promises to make AJAX queries. It is used to manage API calls and communicate with the backend server.

- **Node.js:**

  Node.js is a runtime environment that allows JavaScript code to run on the server, making it possible to create server-side applications.
  It has event-driven, independent I/O competencies, making it ideal for developing scalable and high-performance server programs just like the backend server for the chat utility.

- **SocketIO:** Socket.IO is a JavaScript library and this is known for real-time, bidirectional communication between internet customers and servers over WebSocket. It is critical for integrating instantaneous messaging and live updates into real-time chat capabilities.
- **React Context:** This is used to manage and pass down global state across the component tree without having to pass props down manually at every level

- **Postman:**

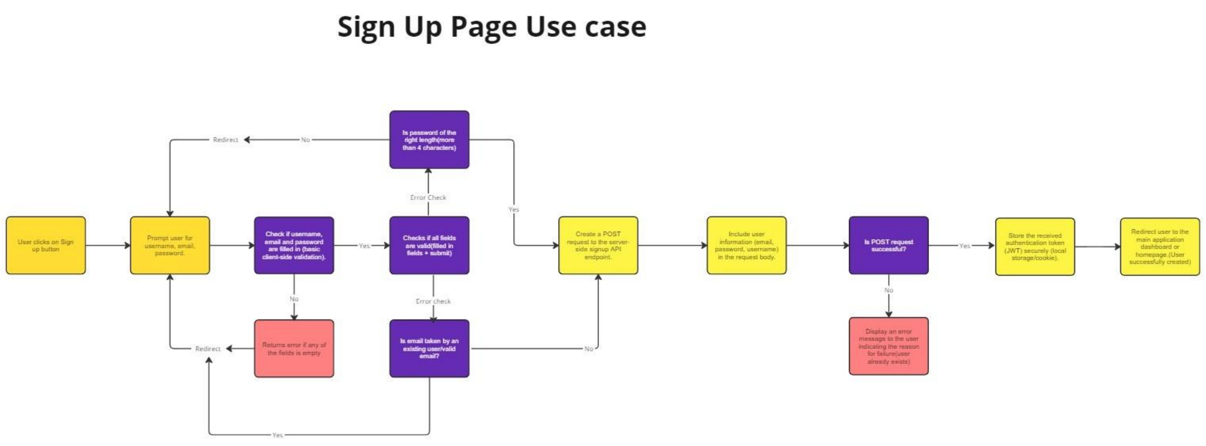
  This is going to be used for testing API routes functionality to make sure that the connections are not only present but also work.
- **Tailwind CSS:**

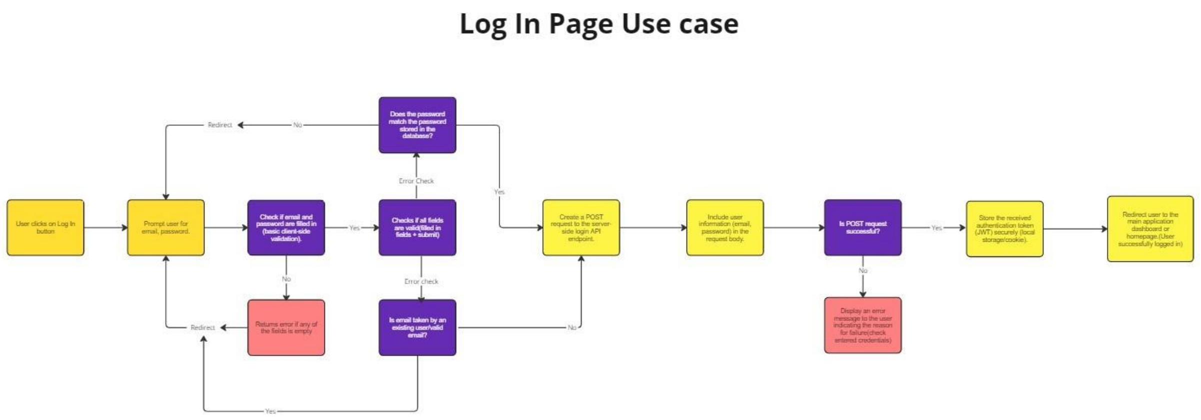
  This is going to be used for styling user components in the application like buttons, text fields etc.
- **Zustand:** Zustand is a lightweight state management toolkit that offers a simple hook-based approach to managing global state in React apps.

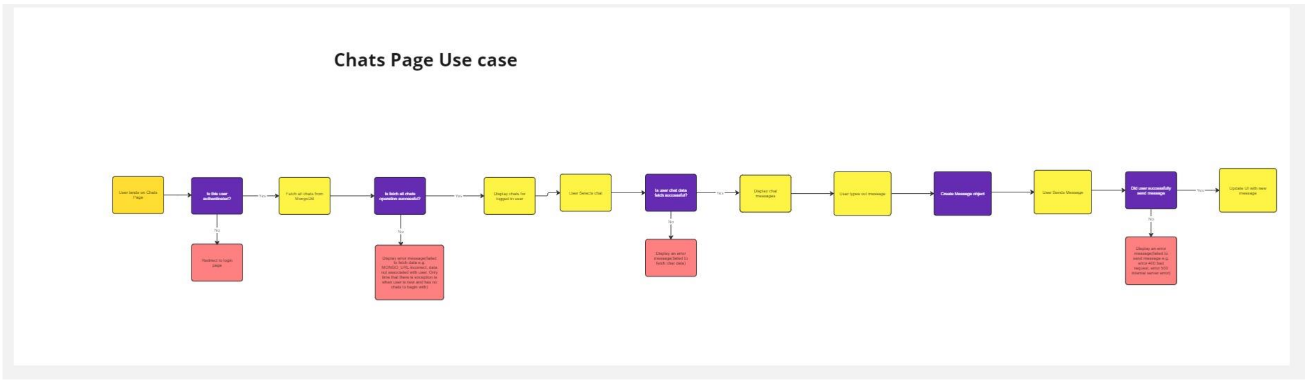
  It is used to manage state throughout the whole application, including user authentication, chat data, and UI states.

4. Design Documents

4.1 Use case flow diagram


Sign Up Page Use case
User clicks on Sign up button
Prompt user for username, email, password.
Check if username, email and password are filled in (basic client-side validation).
Returns error if any of the fields is empty
Checks if all fields are valid(filled in fields + submit)
Is password of the right length(more than 4 characters)
Is email taken by an existing user/valid email?
Create a POST request to the server-side signup API endpoint.
Include user information (email, password, username) in the request body.
Is POST request successful?
Display an error message to the user indicating the reason for failure(user already exists)
Store the received authentication token (JWT) securely (local storage/cookie).
Redirect user to the main application dashboard or homepage.(User successfully created)
Redirect
No
Yes
Error Check
Error check


Log In Page Use case
User clicks on Log In button
Prompt user for email, password.
Check if email and password are filled in (basic client-side validation).
Returns error if any of the fields is empty
Checks if all fields are valid(filled in fields + submit)
Does the password match the password stored in the database?
Is email taken by an existing user/valid email?
Create a POST request to the server-side login API endpoint.
Include user information (email, password) in the request body.
Is POST request successful?
Display an error message to the user indicating the reason for failure(check entered credentials)
Store the received authentication token (JWT) securely (local storage/cookie).
Redirect user to the main application dashboard or homepage.(User successfully logged in)
Redirect
No
Yes
Error Check
Error check


Chats Page Use case
User lands on Chats Page
Is this user authenticated?
Redirect to login page
Fetch all chats from MongoDB
Is fetch all chats operation successful?
Display error message(failed to fetch data e.g. MONGO_URL incorrect, data not associated with user. Only time that there is exception is when user is new and has no chats to begin with)
Display chats for logged in user
User Selects chat
Is user chat data fetch successful?
Display an error message(failed to fetch chat data)
Display chat messages
User types out message
Create Message object
User Sends Message
Did user successfully send message
Display an error message(failed to send message e.g. error 400 bad request, error 500 internal server error)
Update UI with new message
Yes
No

5. Methodology

ConvoCafe was developed in a planned and methodical manner to create a powerful, safe, and user-friendly chat application. Below is the detailed explanation of the features and methodologies used in the project:

Project structure

Backend file structure:

```
server/
|--------controller/
          |--------message.controller.js
          |--------user.controller.js

|---------jwt/
          |--------generateToken.js

|---------middleware/
          |--------secureRoute.js

|---------models/
          |--------conversation.model.js
          |--------message.model.js
          |--------user.model.js

|---------routes/
          |--------message.route.js
          |--------user.model.js

|---------SocketIO/
          |--------server.js

|---------server.js
```

Backend Overview explanation

controller/

- **message**.**controller**.js: It contains the logic that will handle actions related to messaging, like sending and receiving messages.

- **user**.**controller**.js: This logic contains user-related functions like user registration, login and fetching his/her details.

.
server/jwt/

generateToken.js: This file generates JSON Web Tokens (JWTs) for user authentication. It generates tokens containing user information and sets cookies for client-side access.

server/middleware/

secureRoute.js: Route middleware helps to secure the routes. It does this by checking if the request contains a valid JWT token and verifying the user before accessing protected routes.

server/models/

- **conversation**.**model**.js: Defines the database conversation schema.
- **message**.**model**.js: Defines the database's message schema.
- **user**.**model**.js: Defines the database's message schema.

.
server/SocketIO/

server.js(SocketIO setup): Socket.IO configuration and setup for real-time, bidirectional client-server communication are covered in this file.

server.js(Entry point of server): The server's primary point of entry. It starts the server, configures the Express program, and establishes a connection to the database. Furthermore, it sets up Socket.IO for real-time functionality and integrates the routes and middlewares.

Frontend file structure:

```
client/
|--------src/

        |--------assets/
                |--------icon.png
                |--------logo.png
                |--------notification.mp3

        |--------components/
                |--------Loading.jsx
                |--------Login.jsx
                |--------Signup.jsx

        |--------context/
                |--------AuthProvider.jsx
                |--------SocketContext.jsx
                |--------useGetAllUser.jsx
                |--------useGetAllMessage.jsx
                |--------useGetSocketMessage.jsx
                |--------useSendMessage.jsx

        |--------home/
                |--------ChatSpace/
                        |--------Chatuser.jsx
                        |--------Message.jsx
                        |--------Messages.jsx
                        |--------MessageSpace.jsx
                        |--------Typesend.jsx

                |--------LeftPanel/
                        |--------Left.jsx
```

```
          |--------Sidebar/
                    |--------Logout.jsx
                    |--------Search.jsx
                    |--------User.jsx
                    |--------UserDetails.jsx
                    |--------Users.jsx

          |--------zustand/
                    |--------useConversation.jsx

|--------App.jsx

|--------main.jsx
```

Frontend Overview explanation

src/assets/

icon.png, logo.png: These are the images used for the chat application's icons and logos.
notification.mp3: This is the audio file for notification sounds.

src/components/

- **Loading**.**jsx:** This is a component that shows a loading animation while a process or set of data is being fetched
- **Login**.**jsx**: Component that displays the user authentication login form.
- **Signup**.**jsx**: Component that displays the registration form for new users.

src/context/

- **AuthProvider.jsx**: Context provider that manages the state of authentication and offers authentication-related services across the application.

- **SocketContext.jsx**: Provider of context to manage events and connections in Socket.IO, allowing for real-time communication.

- **useGetAllUser.jsx**: Custom hook to retrieve all user information from the database

- **useGetAllMessage.jsx**: Custom hook to retrieve every message associated with the conversation..

- **useGetSocketMessage.jsx**: Custom hook to manage incoming messages via Socket.IO in real time.

- **useSendMessage.jsx:** Custom hook for messaging the server.

src/home/: Contains components related to the main chat interface.

ChatSpace/:

- **Chatuser**.**jsx**: Component in the chat list that represents a user.

- **Message**.**jsx**: component in the conversation that represents a single message bubble
- .
- **Messages**.**jsx**: Component that maps through and shows every message in a chat.

- **MessageSpace**.**jsx**: component that represents the primary message display area.

- **Typesend**.**jsx**: Component for the input textbox space where users can type and send their messages.

LeftPanel/:
Left.jsx: Component of the chat interface's left panel or sidebar.

Sidebar/: Includes parts that are relevant to the sidebar's operation..

- **Logout.jsx**: Component providing the logout functionality with the button

- **Search.jsx**: Component that gives users the ability to look for other users through the search functionality.

- **User.jsx:** Component representing a user in the sidebar with their name and default profile picture.

- **UserDetails.jsx**: Component that shows detailed information about a selected user.

- **Users.jsx**: Component displaying the list of all users in the sidebar.

src/zustand/:

useConversation.**jsx**: Zustand store hook to regulate the state of a conversation, including the messages in the currently selected conversation.

App.jsx: Main application component that establishes the application's routing and general design.

main.jsx: Point of entry for the React program. It wraps the application in the required providers (such as context providers) and initializes the React DOM.

Features of ConvoCafe:

1. User Authentication (Signup/Login)
2. One-to-One Chat & Online Status Indication
3. Search User Function
4. State Management (React Context and Zustand)
5. Real-time Communication (Socket.IO)
6. User Interface (Tailwind CSS and DaisyUI)
7. Backend Infrastructure (Node.js and Express.js)
8. Database Management (MongoDB)
9. Security (JWT and bcrypt)

1) User Authentication (Signup/Login)

User authentication is a vital part of ConvoCafe, allowing users to safely sign up and log in to the application. This process consists of numerous phases and employs a variety of technologies to protect user data while providing a smooth experience. Below is an extended description of the user authentication implementation:

Signup Process

Frontend Handling

- Users can register for an account by providing personal details in a designated registration form, including their desired username, email address, and a chosen password to secure their account.

- The data you enter in a form is checked right away on your device to make sure all the required fields are filled out properly and meet the necessary guidelines.

- If any of the data is missing, you will be prompted to re-enter the missing or invalid data (e.g. email in the wrong format)

- After validation, the data will be sent to the backend server via an HTTP POST request to the "/api/user/signup" endpoint via Axios.

Backend Handling

- The signup request to create a new user account is sent to the server that processes web requests. This server uses Node.js as its runtime environment and Express.js as its framework.

- The request body is processed for user data.

- To avoid repeated registrations, the server checks whether the email address already exists in the MongoDB database.

- When an email address is distinct, the corresponding password is encrypted using bcrypt to guarantee its secure storage in the database. Bcrypt incorporates a random "salt" into each password prior to encryption, rendering it resistant to brute force attacks even if attackers compromise the database.

Token Generation

- Upon successful registration, the server generates a JSON Web Token (JWT) for the user. JWT is a type of data token utilized to securely pass information as a JSON object between users. It is digitally signed, verifying both the authenticity and credibility of the data it contains.

- The user's unique ID and a secure key are stored within the token. This ensures that only the server can decode and confirm the token's validity.

Response to Client

- The server returns the JWT to the client in the response.
- The token is securely stored by the client for use in subsequent authentication requests.

Login Process

Frontend form submission

- The user enters their email and password into a login page.
- The information entered into the form is verified on the client side to make sure both fields
- The data is sent to the backend server via an HTTP POST request to the "/api/user/login" endpoint using Axios.

Backend Handling

- When a user attempts to log in, the server takes in the login request and obtains the user's details from the information provided.

- The server verifies whether the email exists in the MongoDB database.

- When the server finds the email address in its database, it retrieves the hashed password stored with the account. It then uses bcrypt's compare function to check if this hashed password matches the password the user entered.

Token Generation

- In the event that the provided passwords match, the server generates a JWT (JSON Web Token) specifically tailored for the user, mirroring the procedure employed during the user's registration.
- The token contains the user's ID and a secret key.

.

Response to client

- The server returns the JWT to the client in response.
- The token contains the user's ID and a secret key.

Token Verification

Middleware implementation

- To protect routes that require authorization, an intermediary middleware function is employed. This middleware checks and validates the authenticity of the JSON Web Token (JWT) present in the request headers by comparing it against the secret key.
- If the security token is valid, the middleware allows the request to continue to the protected route. If the token is not valid, it gives an error message.

ConvoCafe takes necessary measures to protect the security of its users through a secure authentication system. Passwords are encrypted by bcrypt, so they cannot be accessed by any unauthorized person. The login sessions are handled using JWT tokens, which guarantee the personal data of an individual. The application accommodates high-end backend measures, secure storage of tokens, and frontend validation to allow for hassle-free and secure login by users.

2) One to one chat feature & Online status indication

ConvoCafe's one-on-one chat feature makes it easy for users to communicate with other users in real time. This feature leverages a number of technologies in order to make the delivery experience smooth, effective, and a breeze to use. Here's a full discussion of how the one-on-one chat capability is implemented.

Setting up the Chat Interface

Frontend Interface:

- o **Creation of UI:** Created with React, it delivers a dynamic and responsive user experience.

- **Component structure:** The chat window, along with the message input, and message list are all presented as components. This would make it easier to scale and maintain the code.
- **State management with Zustand:** Zustand is used for managing the status of chat data, enabling high-speed updates and responses to user actions..

User authentication context: User authentication information is handled throughout the project with React Context. This assures that the chat feature can get the information of the authorized user without making it go through prop-drilling, making the code easier to maintain and understand.

***Real-time messaging with Socket.io*:**

Express and Socket.io set up in server: The application handles HTTP requests and responds using Express. An instance of the http module is used to create an HTTP server wrapping the Express app and enabling real-time functionality..

.

Message Sending and receiving:

Client to Server communicaition: Now, when a user sends a message from the client, Socket.IO distributes it to the server.

Server to client communication: The server then processes the message and sends it to the recipient in real time. This happens with listening for message events and then propagating them appropriately.

Storing and retreiving messages:

Database schema

The application stores messages within a MongoDB, retaining conversation history.
A schema for messages is created in Mongoose, having fields for sender ID, recipient ID, content, and timestamp.

Saving messages

Server-side message handling: When the server receives a message, it is stored in the MongoDB database. This ensures that the messages are stored and can later be retrieved..

Retrieving messages

Fetching Messages: Along with that, the server fetches the conversation history from MongoDB when a user starts a chat with another user. Then, the client receives those messages and displays the conversation history.

Frontend display

- **LeftPanel component**: Renders the contact list of users and search bar on the sidebar on the left side of the page.

- **Typesend component**: Enables users to send new messages, which are updated in real time.

- **Messages component**: Displays messages on the ChatSpace of both sender and recipient

- **Message component**: It then proceeds to display the contents of the messages, including their timestamp and the styling of the messages.

The one-on-one chat is created using ConvoCafe, React for UI, Socket.IO for real-time communication, MongoDB to store messages, and Zustand for state management. The approach provides a seamless and efficient real-time messaging experience to users, allowing them to chat anonymously and securely. This blend of technologies makes for a robust and scalable solution for real-time chat capability, enabling users to communicate and converse fluently.

Online status feature

ConvoCafe's online status chat feature brings a much-enhanced user experience through displaying the real-time availability of contacts. This feature monitors the presence of users, updates statuses, and notifies other users. Here's an explanation of how this functionality is implemented.

Real-time tracking

User Connection Management

When a user connects to the server, a connection event is emitted. The server logs the connection and obtains the user's ID from the handshake query. The users object contains the user ID and its related socket ID.

Handling User Disconnection

When a user disconnects, the server realizes the event and removes the user's ID from the users object. Then, the server broadcasts an updated list of online users to every client that is still connected.

Broadcasting and updating user statuses: When somebody establishes a new connection or somebody disconnects, the server broadcasts the getOnlineUsers event to all connected clients. It contains the current list of online user IDs. This ensures that every client is up to date with the current state of their contacts.

Retrieving receiver’s socket ID: A utility method, getReceiverSocketId, is provided to get the socket ID for a given user. This is useful for sending direct messages or notifications to particular individuals.

Client side integration: Clients use Socket.IO to connect to the server. They pass their user ID in the handshake query so the server can identify them. Clients listen to the getOnlineUsers event for online users. The client-side program changes the UI with the current online statuses.

ConvoCafe's online status uses Socket.IO to bring real-time conversation. By handling user connections and disconnections on the server and passing on updates to all clients, the program creates an engaging and responsive user experience. This ensures users are always up to date on the availability of their connections, improving the overall engagement of the chat application.

3) Search User feature

ConvoCafe provides a search feature that helps a user find and link up with a particular individual within the chat application. This feature is targeted to be effective and user-friendly, yielding fast and relevant search results. The following is an in-depth review of the implementation of the search user feature:

Frontend implementation

Component Setup:

- The Search component is a functional component in React. It is responsible for displaying the search input and handling the search logic.
- Imports essential dependencies: React hooks, FaSearch from react-icons for the search button icon, the custom hooks for fetching the user data, useGetAllUsers, state management, useConversation, and toast from react-hot-toast for alerts.

State Management:

- **useState** is used to handle search query input.
- **useGetAllUsers** is a custom hook that obtains all users from the context, ensuring that the search function has access to the most recent user data.
- **useConversation** is a Zustand hook that manages the state of the currently chosen conversation.

Search input and form handling:

- The input for search is a controlled component, value set by search state.
- This ensures that any change in the input field is reflected in the search state.

- A form is used to handle the submission of a search. Submitting the form invokes the handleSubmit function.

Search Logic:

- A form is used to handle search submissions. When the form is submitted, it calls the handleSubmit function.
- The handleSubmit function performs the search query.

- It uses e.preventDefault() to override the default form submission behavior.
- If the search query is empty, the function returns immediately.
- The method scans through the allUsers array looking for a user whose full name contains the search query (case insensitive).

User feedback:

- In case a matched user is found, the Zustand hook function setSelectedConversation changes the state of selected conversation.
- The search input box is cleared by resetting the search status.
- If no match of user is found, a toast notification is sent to the user stating that there was no match.

UI Rendering:

- The component displays an input field and a search button.
- Tailwind CSS classes are used to style the input field, giving it a clean design.
- The search button uses the FaSearch symbol from react-icons and has hover effects to improve user engagement.

Backend Implementation

Fetching Data:

- The useGetAllUsers hook from the frontend makes a call to the API to retrieve all users from the backend. This makes sure the frontend has the most current list of people it can search..
- The backend endpoint /api/users/allusers returns a list of all users in the database. This endpoint is secured by middleware, allowing only authenticated users to access it.

Middleware for Authentication:

The verifyToken middleware ensures that only authenticated users can view the user data. This provides security to the API, thereby protecting sensitive data of users.

API Frontend Implementation:

The getAllUsers hook on the frontend calls the endpoint to retrieve all users in the backend.

This API searches the database to get the record of every user and returns them in JSON format

4) State Management

State management is an integral part of any dynamic program, with special emphasis on a chat application like ConvoCafe, where data changes often and must be represented in real-time across several components. I use two state management technologies in ConvoCafe: React Context and Zustand. The following is a full description of their responsibilities, why they were used, and how they are utilized in the project.

Why React context and Zustand?

Using React Context and Zustand will let me implement each of the approaches' best features in a mix and match approach throughout the application. Here's why i chose this combo.

React Context

- **Purpose**: Ideal for handling global states that do not change frequently and when performance is not the top priority..

- **Usage**: It uses React Context for passing global state, such as user authentication information, across an application without making use of prop-drilling.

- **Reason**: It is built into React and is easier to develop and maintain for basic state requirements.

Zustand

- **Purpose**: A lightweight state management library designed to power complex and dynamic states with ease.

- **Usage**: Zustand is used for managing real-time, fast-changing data and state, such as chat data and user interactions.
-

  Reason: Zustand performs better when you have a complex state to manage, with better state updating and selectors, which avoid useless re-renders.

By using React Context for static global state and Zustand for dynamic state, I ensure that the application is maintainable and performant

React Context implementation for User Authentication

Context Setup:

- AuthContext is created for management of authentication state.
- The functional component AuthProvider provides an authentication context to React. It fetches the data from the cookies or local storage to set the user state and shares it with other components descended from AuthProvider. It allows the components to access and update the authentication state using the useAuth hook.

Usage

- Throughout the application, the useAuth hook gets and modifies authentication information.
- This ensures that the components that need the authentication data can easily consume it without much prop drilling.

Zustand for Chat Data

Store Setup:

- Zustand is used for managing the state of chats, messages, and other dynamic data.
- It is then used for creating a store that can handle chat-related state.

Usage

The useConversationStore hook is used in components that handle or display chat data. This enables quick state updates; eliminating redundant re-renders is possible because Zustand's optimized selectors ensure that only relevant components are updated when the state changes..

In the Search component, zustand is used to manage the conversation state. Referring to the Search Component code (client/src/home/Sidebar/Search.jsx),

- The useConversation hook comes from the useConversation module, which uses zustand to keep the state of the conversation.
- The useConversation hook is used to retrieve the setSelectedConversation function, which updates the selected conversation within the application.
- When a user types in a search query, the handleSubmit function is called. It searches the array allUsers for a user matching the search query and calls the setSelectedConversation method from the useConversation hook to set the selected discussion.
- If a user is found corresponding to the search request, then the conversation becomes the selected conversation and the search input gets removed. In the absence of detection of users, a toast library displays an error message

ConvoCafe handles static global state and dynamic, frequently changing state using both React Context and Zustand. For the details related to user authentication, use is made of React Context, which provides a convenient and smooth way for global state management. Zustand is used for chat-related data, which is rather light and efficient for complicated state management. This combination keeps ConvoCafe performant, maintainable, and scalable, bringing about a smooth user experience.

5) Real Time Communication (Socket.io)

Real-time communication is an essential component of every modern chat platform. ConvoCafe does this with Socket.IO, an intricate JavaScript tool that allows clients and servers to communicate in real time.

Why use Socket.io

Socket.IO is used for real-time communication because it's capable of handling real-time events with low latency, hence ensuring messages and changes are sent virtually instantaneously .
The main reasons for using Socket.io:

- **Real Time Messaging**: Delivers fast message exchange between users for a perfect chatting experience.
- **Bidirectional communication**: Supports bidirectional communication, where both client and server are able to receive messages.
- **Event-based architecture**: Simplifies handling real-time events such as the delivery of messages and changes in user status.
- **Reliability**: Automatic reconnect, multiplex and fallback options—ensuring a stable real-time connection..

The following is an in depth explanation of how Socket.IO is utilized in ConvoCafe to provide real-time communications and updates, with reference to the code:

Server-side

Server setup: On the server side, the program creates an Express.js web application and an HTTP server. Afterward, it creates an instance of a Socket.IO server and connects it to the HTTP server. This integration will allow the Socket.IO server to use the current HTTP server architecture in order to handle WebSocket connections.

The Socket.IO server is then configured to allow Cross-Origin Resource Sharing, given the origin as http://localhost:3001. This will allow the client-side application that runs under a different port to establish a secure connection with the server.

User connection management(connecting and disconnecting):

When a client connects to the server through Socket.IO, it triggers a "connection" event. In this event, the server retrieves the user's ID from the connection request and links it to the Socket.IO socket ID.

When a client disconnects, the "disconnect" event is triggered. The server then removes the user's information from the "users" object and sends the "getOnlineUsers" event to update all connected clients with the current list of online users. The "getReceiverSocketId" function retrieves the Socket.IO socket ID corresponding to a specific user ID, which enables the server to determine who should receive a message.

Every time a client connects or disconnects, the "getOnlineUsers" event is sent to all connected clients. This event keeps each client informed of the current online users, allowing for real-time updates.

Online tracking

The getReceiverSocketId function returns the Socket.IO socket ID of a given user by user ID. This function might be used to determine the target recipient for real-time message delivery. An event gets fired in the form of getOnlineUsers for every user's connection or disconnection to all the connected clients. This event will refresh the list of online users for the client-side application. Hence, this event will update the real-time status of active participants in the chat application.

Practical Implications

Real Time messaging: When a client sends a message, the client-side application can use the getReceiverSocketId method to retrieve the recipient's Socket.IO socket ID. The application can then forward a message event to the server, which will use the socket ID to deliver the message to the intended recipient.

Online presence: The getOnlineUsers event provides the client-side application the ability to maintain a current list of online users; it can therefore provide features such as \"who's online\" notifications and presence management.

Socket.IO enables the creation of real-time chat applications that can handle thousands of users simultaneously. By effectively managing client connections and utilizing WebSocket technology, it ensures reliable and fast communication between users

ConvoCafe utilizes Socket.IO to maintain real-time interactions by delivering instant messages and updates—an essential function in a chat platform. Socket.IO is built on an event-driven mechanism and guarantees connection management reliably, all of which are very important for implementing seamless and stable instant communication.

6) User Interface

ConvoCafe's interface is designed to be visually pleasing and easy to use. This is made possible by using Tailwind CSS and DaisyUI, which are powerful tools for creating consistent and stylish web interfaces. The following is a full description of how these technologies are employed in ConvoCafe to provide a smooth and responsive user experience.

Why TailwindCSS and DaisyUI?

Tailwind CSS is a powerful design tool that allows developers to quickly build website user interfaces by using its low-level utility classes. DaisyUI is an add-on for Tailwind CSS that makes the process even smoother by providing a ready-made collection of visually appealing components, streamlining the creation of consistent user interfaces.

Tailwind CSS:

Utility-First Approach: With Tailwind, you can effortlessly apply styles to elements directly in your JSX code, using its utility-centric approach.

Customizability: Tailwind CSS offers a range of flexible customization options, enabling users to create a distinctive and uniform design aesthetic.

Responsive Design: The application is equipped with tools that automatically adapt the design to any screen size, guaranteeing a visually appealing experience regardless of the device being used.

DaisyUI:

Pre-designed Components: Provides a collection of pre-styled components, such as buttons, forms, and pop-up windows, that seamlessly integrate with Tailwind CSS.

- **Consistency**: The app's overall design maintains a consistent look and feel because it follows a single design scheme.
- **Ease of Use**: Utilizing reusable and adaptable UI components streamlines the process of designing elegant and tailored user interfaces..

Just to show and explain a bit more on this, I will be taking the code shown here from the Chatuser component to show the use of both TailwindCSS and DaisyUI

```
import React from "react";
import useConversation from "../../zustand/useConversation.jsx";
import { useSocketContext } from "../../context/SocketContext.jsx";
import { CiMenuFries } from "react-icons/ci";
import Icon from "../../assets/icon.jpg";

const Chatuser = () => {
  const { selectedConversation } = useConversation();
  const { onlineUsers } = useSocketContext();
  const getOnlineUsersStatus = (userId) => {
    return onlineUsers.includes(userId) ? "Online" : "Offline";
  };

  return (
    <div className="relative flex items-center h-[8%] justify-center gap-4 bg-
slate-800 hover:bg-slate-700 duration-300 rounded-md">
      <label
        htmlFor="my-drawer-2"
        className="btn btn-ghost drawer-button lg:hidden absolute left-5"
      >
        <CiMenuFries className="text-white text-xl" />
      </label>
      <div className="flex space-x-3 items-center justify-center h-[8vh] bg-
gray-800 hover:bg-gray-700 duration-300">
        <div className="avatar online">
          <div className="w-10 rounded-full">
            <img src={Icon} />
          </div>
        </div>
        <div>
          <h1 className="text-xl">{selectedConversation.fullname}</h1>
          <span className="text-sm">
            {getOnlineUsersStatus(selectedConversation._id)}
          </span>
        </div>
      </div>
    </div>
  );
};

export default Chatuser;
```

For the TailwindCSS,

"relative flex items-center h-[8%] justify-center gap-4 bg-slate-800 hover:bg-slate-700 duration-300 rounded-md"

This set of Tailwind CSS utility classes styles the main container for the Chatuser component. It aligns items to the center, sets the height to 8%, centers content horizontally, adds a 4-unit gap, and sets the background color to slate-800, changing to slate-700 on hover, with a 300ms transition duration and a rounded border..

“flex space-x-3 items-center justify-center h-[8vh] bg-gray-800 hover:bg-gray-700 duration-300”

This series of classes tweaks the inner container that holds the user's avatar and name. It flexes the display, gives it 3 units horizontally in spacing between child components, centers the items vertically and horizontally, sets the height to 8vh, sets the background color to gray-800, changes the background color to gray-700 when hovered on, and uses a 300ms transition length.

"w-10 rounded-full"
These classes are applied to the user’s avatar image, setting its width to 10 units and also making it a rounded circle

For the DaisyUI:

“avatar online”

This class uses DaisyUI's avatar component to show users' profile photos. It also includes a feature where an online green indicator is shown on the avatar when the user is online.

"btn btn-ghost drawer-button lg:hidden absolute left-5"

This is the set of classes which uses the DaisyUI class btn, with the modifier of btn-ghost to make the button transparent. There is also a drawer-button class used here, which is specific to DaisyUI, meaning a button which opens a drawer. The lg:hidden class is hiding the button

on larger displays, while absolute left-5 classes position the button exactly to the left of the container.
This code makes use of Tailwind CSS and DaisyUI for the fast and uniform styling of the Chatuser component. Tailwind CSS allows using several utility classes for layout, spacing, and colors, whereas DaisyUI includes pre-designed components like the button and avatar in the outcome.

Applying these technologies, I was able to build an interactive and flexible chat interface following the application principles of design but benefiting from the efficiency and maintainability that Tailwind CSS and DaisyUI can offer.

7) Backend Implementation(Node.js and Express.js)

The ConvoCafe backend is designed using Node.js and Express.js. The combination makes for a reliable and scalable environment for server-side operations, APIs, and real-time communication. The following is an in-depth overview of how Node.js and Express.js are used in ConvoCafe, focusing their responsibilities and the reasoning behind their implementation.

Why Use Node.js?

- **Node.js**: Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. It allows running JavaScript on the server-side, which makes full-stack JavaScript development possible.
- **Non-blocking I/O**: Its non-blocking, event-driven architecture makes it well-suited for dealing effectively with concurrent connections.
- **Single Language**: Using JavaScript on both client and server sides will make development and maintenance easier.
- **Scalability**: Node.js is very scalable, excellent for applications that demand a large number of simultaneous connections, such as a chat application.

Why Use Express.js?

Express.js: Express.js: It is a lightweight and flexible Node.js web application framework that provides all the functionality required to build mobile and web applications.

Middleware: It utilizes middleware to process HTTP requests, which makes it easy to write modular and maintainable code.

Routing: Express.js provides a simple routing method to handle several endpoints in the application.

Integration: It easily connects to many databases and middleware, making it a full development environment.

In my backend server file(server.js), this is how I implemented these technologies to get it working:

- **Importing Dependencies**: It starts by importing the needed dependencies: Express, dotenv (for environment variables), Mongoose (which will allow it to interact with MongoDB), CORS (to allow cross-origin resource sharing), and Cookie-Parser (to process cookies).
- **Configuring the middleware:** The code is setting up Express.js middleware, including express.json() for parsing JSON bodies from requests, cookieParser() to manage cookies, and cors() to enable cross-origin resource sharing.
- **MongoDB connection**: The mongoose.connect() function is used in this code to connect to a MongoDB database with a MongoDB connection URI supplied in the environment variables..
- **Routing**: It configures the routing of the chat application, using modules userRoute and messageRoute to handle user- and message-related actions, respectively.

- **Initializing the server**: It configures the routing of the chat application, using modules userRoute and messageRoute to handle user- and message-related actions, respectively.

By leveraging Node.js and Express.js for the backend server, I can take advantage of the following benefits in the chat application:

Cohesive development: Leveraging JavaScript both on the client-side (web browser) and server-side (web server) streamlines development by enabling code to be shared across both ends. This enhanced interaction between the user interface and the data processing components improves the overall functionality and consistency of the application.

Efficient concurrency handling: Node.js' unique design, which focuses on handling events without blocking, allows a server to efficiently handle a multitude of simultaneous connections. This characteristic is crucial for chat applications that require smooth and responsive real-time communication.

Scalable infrastructure: Node.js and Express.js make the chat app able to grow and change to meet the needs of more users. In this way, the app stays fast and easy to use.

Maintainable and modular codebase: Express.js' middleware-based approach and routing techniques are a way for me to build up a modular, maintainable backend structure, whereby new features and functions can be easily added as the chat application grows..

Overall, integrating Node.js and Express.js into the chat application's backend server serves as the foundation for creating a real-time, scalable, and flexible communication platform.

8) Database Management(MongoDB)

Besides, every program, especially a chat application, has to take data integrity, scalability, and speed into consideration. ConvoCafe utilizes MongoDB, a NoSQL database, to store its data. Here is an in-depth look at how MongoDB is utilized in ConvoCafe to maintain the user data, messages, and other important information.

Why Use MongoDB?

Scalability: MongoDB can scale horizontally and handle tons of data, which makes it perfect for applications that have an increasing number of users and corresponding data..

- **Flexibility**: Document-oriented architecture in MongoDB enables flexible schema designing, hence enhancing agility in scenarios that change quickly..
- **Performance**: MongoDB excels in processing large amounts of data quickly, both for reading and writing, making it ideal for apps demanding instantaneity, such as chat applications.
- **JSON-Like Documents**: MongoDB stores information in a structure similar to JavaScript Object Notation (JSON). This format allows for seamless integration with JavaScript-based applications and makes it easier to manage data

Conversation Model

This model represents the conversations between users in the chat application.

Members:

- The "members" field in the database record stores a list of unique identifiers (ObjectIds in MongoDB) that represent the users participating in the chat

- In the database schema, the \`ref\` (reference) option for the \`members\` field is linked to the \`User\` model. This tells us that the \`members\` field contains references to users who were involved in the conversation.

- This design enables the chat application to effortlessly obtain information about the users involved in a specific chat conversation.

Messages:

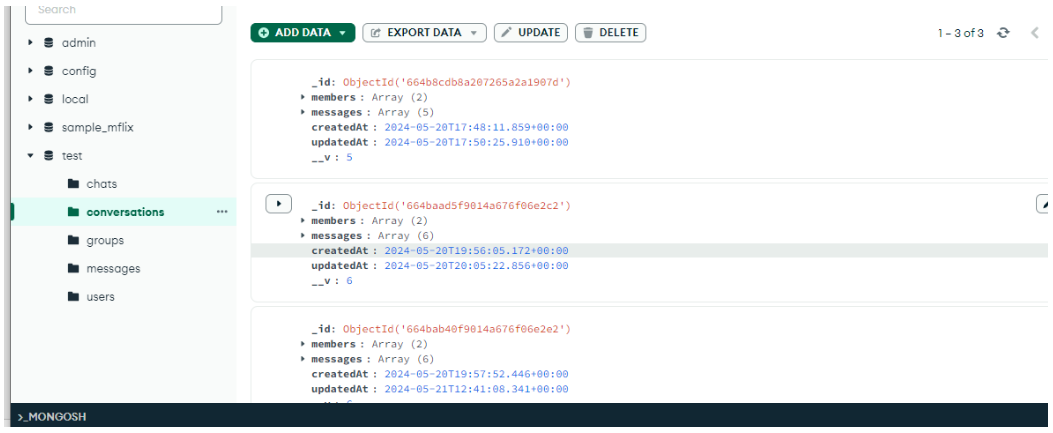
- The "messages" field in the database is a list that contains unique identifiers (in the form of MongoDB ObjectIds) for all the messages that belong to the discussion.

- The "ref" option is configured to link messages to a particular discussion by associating them with the Message model.

- By organizing messages into conversations, the chat application can efficiently retrieve all messages related to a specific conversation, eliminating the requirement for a separate database or structure to store messages.

Timestamps:
Enabling the "{ timestamps: true }" option in the Conversation schema adds two fields: "createdAt" and "updatedAt." These fields track when a conversation was created or modified. The Conversation model is connected to the User and Message models using MongoDB ObjectIds and the "ref" option in the schema definition.

.



```
_id: ObjectId('664b8cdb8a207265a2a1907d')
members : Array (2)
messages : Array (5)
createdAt : 2024-05-20T17:48:11.859+00:00
updatedAt : 2024-05-20T17:50:25.910+00:00
__v : 5

_id: ObjectId('664baad5f9014a676f06e2c2')
members : Array (2)
messages : Array (6)
createdAt : 2024-05-20T19:56:05.172+00:00
updatedAt : 2024-05-20T20:05:22.856+00:00
__v : 6

_id: ObjectId('664bab40f9014a676f06e2e2')
members : Array (2)
messages : Array (6)
createdAt : 2024-05-20T19:57:52.446+00:00
updatedAt : 2024-05-21T12:41:08.341+00:00
```

Message Model

Represents the individual exchanged messages between users in the chat applicaition.

Sender and Receiver:

- The senderId field stores the MongoDB ObjectId of the user who sent the message.
- The receiverId field stores the MongoDB ObjectId of the user who received the message.
- The ref option in the schema definition is set to the "User" model, indicating that the senderId and receiverId columns relate to the User model.

This design enables the chat application to quickly identify the people engaged in a given message exchange and collect their information as needed.

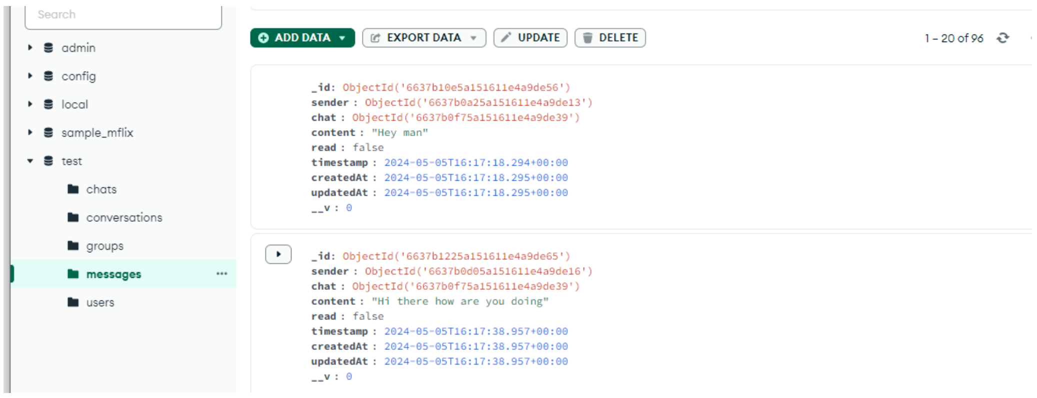
Message Content:

- The message field stores the actual content of the message as a string.
- This field is marked as required, ensuring that every message has a non-empty message content.

Timestamps:

The { timestamps: true } option in the schema definition adds the createdAt and updatedAt columns to the Message document, allowing the application to track each message's creation and modification timestamp.

The senderId and receiverId attributes, which use MongoDB ObjectIds to relate to the User model, create the relationship between the Message and User models.



```
_id: ObjectId('6637b10e5a151611e4a9de56')
sender : ObjectId('6637b0a25a151611e4a9de13')
chat : ObjectId('6637b0f75a151611e4a9de39')
content : "Hey man"
read : false
timestamp : 2024-05-05T16:17:18.294+00:00
createdAt : 2024-05-05T16:17:18.295+00:00
updatedAt : 2024-05-05T16:17:18.295+00:00
__v : 0

_id: ObjectId('6637b1225a151611e4a9de65')
sender : ObjectId('6637b0d05a151611e4a9de16')
chat : ObjectId('6637b0f75a151611e4a9de39')
content : "Hi there how are you doing"
read : false
timestamp : 2024-05-05T16:17:38.957+00:00
createdAt : 2024-05-05T16:17:38.957+00:00
updatedAt : 2024-05-05T16:17:38.957+00:00
__v : 0
```

User Model

Represents the user accounts in the new chat application

Fullname:

- The fullname field contains the user's full name as a string.
- This field is marked as essential to ensure that all users have a full name.

Email:

- The email field contains the user's email address as a string.
- This field has been set as required and unique, so that each user has a unique email address..

Password:

- The password field contains the user's password as a string.
- This field is marked as required to ensure that all users have a password.

Confirm Password:

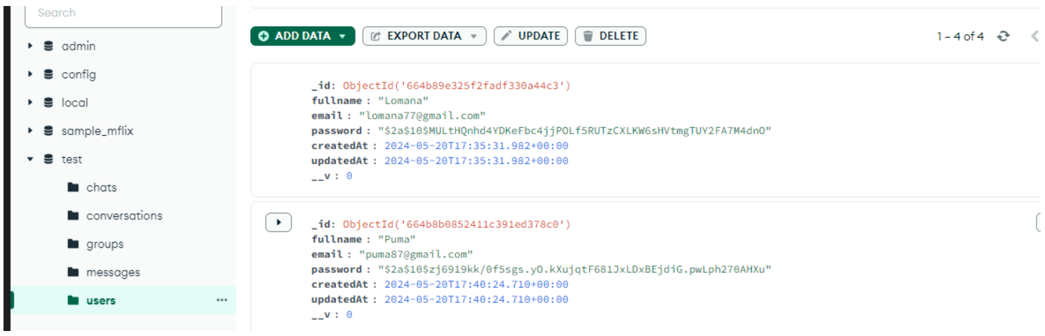
- The confirmPassword field contains the user's confirmed password as a string.

- This field is not listed as required since it is normally used to validate user input during the registration process but is not saved in the database.

Timestamps:

The { timestamps: true } option in the schema definition adds createdAt and updatedAt fields to the User document, allowing the application to track the creation and modification timestamps for each user account..

The User model serves as the foundation for user-related operations in the chat application, such as authentication, user profile management, and user-to-user interactions.



```
_id: ObjectId('664b89e325f2fadf330a44c3')
fullname : "Lomana"
email : "lomana77@gmail.com"
password : "$2a$10$MULtHQnhd4YDKeFbc4jjPOLf5RUTzCXLKW6sHVtmgTUY2FA7M4dnO"
createdAt : 2024-05-20T17:35:31.982+00:00
updatedAt : 2024-05-20T17:35:31.982+00:00
__v : 0
```

```
_id: ObjectId('664b8b0852411c391ed378c0')
fullname : "Puma"
email : "puma87@gmail.com"
password : "$2a$10$zj6919kk/0f5sgs.yO.kXujqtF681JxLDxBEjdiG.pwLph270AHXu"
createdAt : 2024-05-20T17:40:24.710+00:00
updatedAt : 2024-05-20T17:40:24.710+00:00
__v : 0
```

9) Security

Security is integral to any website, more so with the chat application like ConvoCafe, which transmits and stores sensitive information belonging to users. ConvoCafe utilizes JSON Web Tokens for secure authentication and bcrypt for password hashing, ensuring a user's data is safe at all times. Below is an extensive overview of how ConvoCafe uses JWT and bcrypt to improve security.

Why Use JWT and bcrypt?

- **JSON Web Tokens (JWT)**: JWT is a compact, URL-safe, and self-contained mechanism for securely transferring information from one party to another as a JSON object. It's commonly used for authentication because:
  - **Stateless Authentication**: There is no need to keep session information on the server.
  - **Security**: Information may be verified and trusted since it has been digitally signed.
  - **Flexibility**: Can be used across different platforms and domains.

  - **bcrypt**: bcrypt is a password hashing method meant to be costly to compute while resisting brute-force assaults.

It is chosen because:

- **Salting**: Each password is assigned a unique salt before hashing; this ensures that identical passwords will have different hashes.
- **Security**: Designed to be slow and resistant to brute-force attacks.

How it is implemented

Token generation(generateToken.js)

- **JWT Generation**: jwt.sign is generating a JWT token with the user's ID as a payload. That token is signed by the secret key process.env.JWT_TOKEN and is set to last for 10 days..
- **Setting the Cookie**: The created token is stored as a cookie in the user's browser. The cookie settings improve security.:

- “httpOnly”: "true”: Prevents client-side scripts from accessing the cookie, providing protection against cross-site scripting (XSS) attacks..
- “secure: true”: Ensures that the cookie is only transmitted via HTTPS.
- “sameSite”: "strict": Prevents cross-site request forgery (CSRF) by ensuring that the cookie is only transmitted in requests from the same site.

Middleware configuration for protected routes

- **Token Retrieval**: Middleware extracts the JWT from the cookies in the request..

- **Token Verification**: jwt.verify validates the token's validity using the secret key. If the token is incorrect or expires, an error will be returned..

- **User Retrieval**: If the token is valid, the middleware extracts the user's information from the database (except the password) using the user ID included in the token..

- **Attaching User to Request**: The user information is connected to the request object, which may be accessed by later middleware or route handlers.

- **Error Handling**: If any phase fails, suitable error messages are generated, and internal errors are logged.

Password hashing using bcrypt

- **Generating Salt**: bcrypt.genSalt(10) generates a salt with 10 rounds, increasing the complexity of the hashing operation.
- **Hashing Password**: bcrypt.hash(password, salt) hashes the user's password with the salt, ensuring unique hashes for identical passwords..
- **Saving User**: The hashed password is stored in the database rather than the plaintext password, adding an extra degree of protection.

Password verification

- **Password Comparison**: bcrypt.compare(password, hashedPassword) compares the passed-in username and password against the stored hash. This function does the salting and hashing internally, to make sure correctness is maintained..
- **User Verification**: If the passwords match, a JWT is generated for the user, which allows secure authentication.

.Full authentication workflow

Signup process

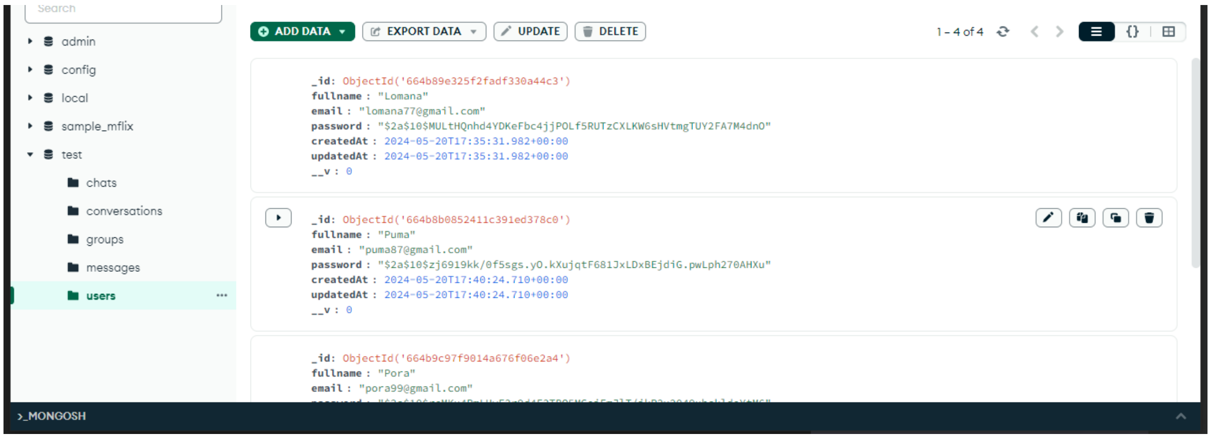
- **Hashing Password**: The user's password is hashed before being saved to the database..
- **Creating User**: A new user document is created and stored.
- **Generating Token**: A JWT is created and stored as a cookie in the user's browser.

Login Process

- **User Retrieval**: The user is fetched from the database using the email address given.
- **Password Verification**: The inputted password is compared to the stored hash.
- **Generating Token**: When a user successfully logs in, a JWT is generated and saved as a cookie in their browser.

ConvoCafe protects user data during transmission and storage by using JWT authentication and bcrypt password hashing. JWT offers a safe and stateless method of authenticating user identities, whereas bcrypt adds another degree of security by hashing passwords before storing them. This mix of technologies not only improves the application's security but also offers users with a smooth and swift authentication experience.

How this all looks in MongoDB:



```
_id: ObjectId('664b89e325f2fadf330a44c3')
fullname : "Lomana"
email : "lomana77@gmail.com"
password : "$2a$10$MULtHQnhd4YDKeFbc4jjPOLf5RUTzCXLKW6sHVtmgTUY2FA7M4dnO"
createdAt : 2024-05-20T17:35:31.982+00:00
updatedAt : 2024-05-20T17:35:31.982+00:00
__v : 0

_id: ObjectId('664b8b0852411c391ed378c0')
fullname : "Puma"
email : "puma87@gmail.com"
password : "$2a$10$zj6919kk/0f5sgs.yO.kXujqtF681JxLDxBEjdiG.pwLph270AHXu"
createdAt : 2024-05-20T17:40:24.710+00:00
updatedAt : 2024-05-20T17:40:24.710+00:00
__v : 0

_id: ObjectId('664b9c97f9014a676f06e2a4')
fullname : "Pora"
email : "pora99@gmail.com"
```

6. Conclusion

This real-time chat application developed with Socket.IO and the MERN stack (MongoDB, Express, React, and Node.js) is a noteworthy milestone in building a dependable and scalable communication tool. This program ensures data confidentiality and effective state management in addition to offering users a smooth and user-friendly interface for real-time messaging. The application showcases modern web development best practices by integrating Zustand for effective state management and JWT for safe authentication.

6.1 Benefits

To the Users

Real-Time Communication: Instant messaging allows users to communicate with colleagues and close friends, improving social interaction.

Secure Environment: By implementing JWT and bcrypt, a secure communication platform is provided and user data is protected.

User-Friendly Interface: The program is simple to use and navigate thanks to Daisy UI's clear, intuitive layout, which enhances the user experience overall..

Search Functionality: The application becomes more interactive and engaging when users can locate and connect with other users with ease..

Responsive Design: Because of its responsive design, the application operates effectively across a range of gadgets, including PCs, tablets, and smartphones.

To me:

Scalability: MongoDB, a NoSQL database, enables the application to effectively manage a great deal of data and heavy traffic.

Maintainability: The codebase's scalability and maintainability are guaranteed by the usage of the MERN stack. Updates and maintenance are made simpler by a separation of tasks between the front-end and back-end.

Modern Technologies: The project makes use of modern web technologies and frameworks to offer a platform for learning web development best practices.

Extensibility: Because of the application's modular nature, adding additional features and functionality in the future will be simple

Future Enhancements

- **Group Chat Functionality**: Group chat functionality should be included to enable users to speak with multiple people at once.
- **File Sharing**: Putting in place a feature that enables chat participants to exchange data, photos, and videos.
- **Voice and Video Calls**: Combining video conference and VoIP features to offer a more engaging conversation experience.
- **Push Notifications**: Push notifications should be included so that users are informed of new messages or activity even when they don't use the app.
- **Message Reactions**: enabling emoji responses to messages, similar to social networking platform reactions.

AI Chatbots: Introducing chatbots powered by AI to help users with common questions and provide automated solutions, similar to WhatsApp’s implementation of Meta AI.

Tools for Moderation: Including means of reporting and banning abusive people.

Content Filtering: putting in place automated processes to identify and remove objectionable content.

Benefits of the Future Enhancements

To the Users

- **Enhanced Communication**: Group chat, audio, and video conversations are just a few of the features that will greatly improve user communication.
- **Better User Engagement**: Users will stay informed and involved with push notifications and message reactions.

To me

- **User Retention**: Improved features will increase user retention and satisfaction.
- **Increased Value**: The application will gain a great deal of value from these features, increasing its appeal to possible acquirers or investors.

6.2 Ethics

You should explain the ethics of your project area, ethics should be supported with your resources

To ensure that real-time chat applications are developed and implemented in a way that serves users fairly and responsibly, a number of ethical guidelines must be followed. The following are the main moral issues that are pertinent to this project:

1. User Privacy

Protecting user privacy is critical. The chat program protects sensitive user data, such as private chats and personal information.. Adhering to privacy laws such as the General Data Protection Regulation (GDPR) is essential. This involves:

Data Encryption: Data encryption safeguards against unwanted access by encrypting data while it's in motion and at rest.

User Consent: Before collecting personal information, users should be informed about data collection procedures and their consent should be obtained.

Minimal Data Collection: To minimize potential privacy hazards, just the data required for the application's functionality is collected.

2. Data Security

- Robust security measures must be implemented by the application to guard against cyberattacks and data breaches. This includes:

- Authorization and Authentication: Ensuring that users can only access the information and features they are permitted to access by employing JWT for safe user authentication.

- Frequent Security Audits: To detect and reduce potential security risks, perform regular security audits and vulnerability assessments.

3. Transparency and Accountability

Transparency about how the application operates and handles user data is crucial. This involves:

- **Clear regulations**: Having easily comprehensible terms of service and privacy regulations.

- **Measures for Accountability**: Putting in place procedures for transparent reporting and handling user complaints and data breaches.

4. Inclusivity and Accessibility

The application should be designed to be inclusive and accessible to all users, regardless of their abilities or backgrounds. This includes:

- **Accessibility features** include making sure the application complies with guidelines like the Web Content Accessibility Guidelines (WCAG) and is useable by individuals with impairments.

- **Diverse User demands**: During the design and development phase, take into account the various user demands and settings.

5. Fairness and Non-Discrimination

The application ought to ensure that no group is discriminated against and that all users are treated equally. This involves:

Bias mitigation, which is the process of making sure algorithms and AI components are created without biases that could cause consumers to be treated unfairly.

Equal Access: Giving every user equal access to all of the application's features and functionalities
.

Why did I choose this project?

ConvoCafe is a platform that aims to enhance real-time communication. It recognizes the growing demand for instant messaging apps in the evolving technological landscape. The focus of this project was to design a chat application that facilitates effortless communication while emphasizing user experience, security, and scalability. This effort involved exploring web programming complexities using MERN stack, Socket.IO, and advanced state management techniques, providing a comprehensive learning experience. The project delved into practical challenges such as secure authentication, real-time data management, and optimizing user interactions.

6.3 Future Works

Upon graduation, I intend to enhance ConvoCafe further by introducing crucial features that elevate its functionality and user experience. A primary objective is to implement group chat capabilities. Currently offering one-on-one conversations, ConvoCafe will expand to enable dynamic and collaborative group interactions. This will involve developing features for group creation, management, and administration, alongside robust notification systems to ensure users remain informed about group activities. Additionally, integrating video calling and file sharing functionalities will transform ConvoCafe into a versatile communication platform catering to diverse user needs. Furthermore, I plan to bolster the application's security. While ConvoCafe currently utilizes JWT for authentication, I aim to implement additional security measures to safeguard user data and privacy. By employing encryption protocols, two-factor authentication, and regular security assessments, I will ensure ConvoCafe meets the highest standards of data protection.

7. References

1. **https://www.pubnub.com/blog/web-based-chat-application/**
2. **https://www.w3schools.in/what-is-client-server-architecture/**
3. **https://www.w3schools.in/what-is-client-server-architecture/**
4. **https://medium.com/@kartikpatel_97737/building-a-node-js-websocket-chat-app-with-socket-io-437f3ba65b**
5. **https://expressjs.com/**
6. **https://medium.com/@skhans/building-web-applications-with-express-js-a-comprehensive-guide-113a77be1b11**
7. **https://medium.com/@nonamedev/building-full-stack-applications-with-mern-stack-a-comprehensive-guide-402e99d2e959**
8. **https://medium.com/@pushkaraj2007/what-is-the-mern-stack-a-complete-introduction-and-guide-for-beginners-95e288bcac3e**

9. **https://www.cometchat.com/tutorials/how-to-build-a-chat-app-with-websockets-and-node-js**
10. **https://javascript.plainenglish.io/how-to-build-a-websocket-chat-application-819399d55800**
11. **https://piehost.com/websocket/how-to-build-websocket-server-and-client-in-nodejs**
12. **https://daily.dev/blog/tailwind-css-basics-for-beginners**
13. **https://www.tyntec.com/blogs/challenges-facing-messaging-industry/**