**# \*\*System Design Document: Real-Time Chat Application using MERN Stack\*\***

## \*\*Table of Contents\*\*

1. [\*\*1. Overview\*\*](#1-overview)

2. [\*\*2. Key Dependencies\*\*](#2-key-dependencies)

3. [\*\*3. Architecture\*\*](#3-architecture)

- [3.1. System Architecture Diagram](#31-system-architecture-diagram)

- [3.2. Technology Stack](#32-technology-stack)

4. [\*\*4. Components and Workflows\*\*](#4-components-and-workflows)

- [4.1. Authentication Workflow](#41-authentication-workflow)

- [4.2. Chat Workflow](#42-chat-workflow)

- [4.3. Profile Management Workflow](#43-profile-management-workflow)

5. [\*\*5. Database Schema\*\*](#5-database-schema)

- [5.1. User Collection](#51-user-collection)

- [5.2. Chat Collection](#52-chat-collection)

6. [\*\*6. API Design\*\*](#6-api-design)

- [6.1. Authentication APIs](#61-authentication-apis)

- [6.2. Chat APIs](#62-chat-apis)

- [6.3. User Profile APIs](#63-user-profile-apis)

7. [\*\*7. Real-Time Communication\*\*](#7-real-time-communication)

- [7.1. Event Flow Diagram](#72-event-flow-diagram)

8. [\*\*8. Security Considerations\*\*](#8-security-considerations)

9. [\*\*9. Scalability and Performance\*\*](#9-scalability-and-performance)

10. [\*\*10. User Interface Design\*\*](#10-user-interface-design)

- [10.1. UI Components](#101-ui-components)

- [10.2. UI Flow Diagram](#102-ui-flow-diagram)

11. [\*\*11. Conclusion\*\*](#11-conclusion)

---

**## \*\*1. Overview\*\***

The \*\*BuzzLine Application\*\* is designed to facilitate seamless communication between users in real-time. Built using the \*\*MERN (MongoDB, Express.js, React.js, Node.js)\*\* stack, the application leverages \*\*Socket.io\*\* for real-time interactions, \*\*JWT (JSON Web Tokens)\*\* for secure authentication, and \*\*Chakra UI\*\* for a responsive and modern user interface. The application allows users to register, authenticate, join chat rooms, send and receive messages instantly, and manage their profiles.

**## \*\*2. Key Dependencies\*\***

**### \*\*Frontend Dependencies:\*\***

- \*\*React (`^17.0.2`)\*\*: JavaScript library for building user interfaces.

- \*\*@chakra-ui/react (`^1.6.2`)\*\*: UI component library for React.

- \*\*@chakra-ui/icons (`^1.0.13`)\*\*: Icon library for Chakra UI.

- \*\*@emotion/react (`^11`) & @emotion/styled (`^11`)\*\*: CSS-in-JS libraries for styling.

- \*\*react-router-dom (`^5.2.0`)\*\*: Routing library for React applications.

- \*\*axios (`^0.21.1`)\*\*: Promise-based HTTP client for making API requests.

- \*\*socket.io-client (`^4.1.2`)\*\*: Client-side library for Socket.io.

- \*\*Other Libraries\*\*:

- `bcryptjs`: For password hashing.

- `jsonwebtoken`: For JWT-based authentication.

- `colors`, `dotenv`, `express-async-handler`, `mongoose`, `nodemon`, `openai`, `socket.io`: Backend dependencies.

**### \*\*Backend Dependencies:\*\***

- \*\*Node.js (`^14.x`)\*\*: JavaScript runtime environment.

- \*\*Express.js (`^4.17.1`)\*\*: Web framework for Node.js.

- \*\*Mongoose (`^5.12.9`)\*\*: ODM for MongoDB.

- \*\*Socket.io (`^4.1.2`)\*\*: Enables real-time, bidirectional communication.

- \*\*bcryptjs (`^2.4.3`)\*\*: Library for hashing passwords.

- \*\*jsonwebtoken (`^8.5.1`)\*\*: For generating and verifying JWTs.

- \*\*dotenv (`^9.0.2`)\*\*: Loads environment variables from `.env` file.

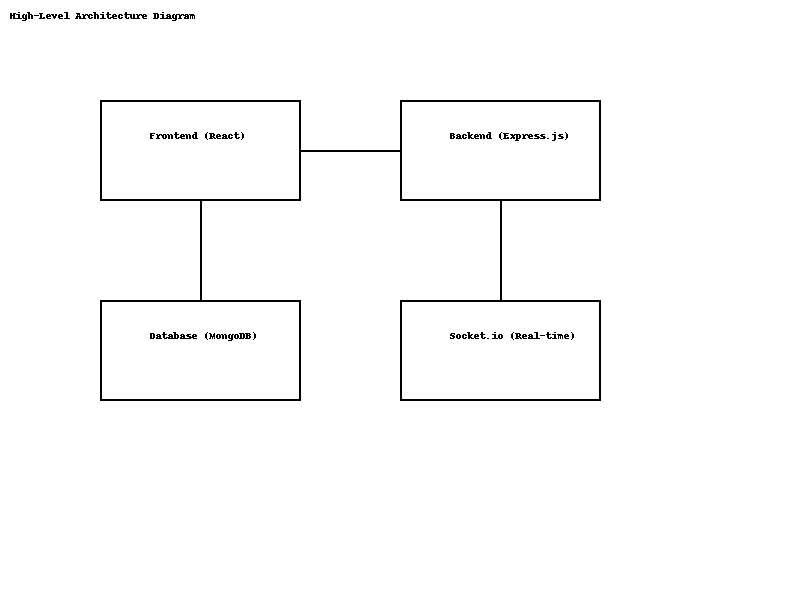
- \*\*express-async-handler (`^1.1.4`)\*\*: Middleware to handle exceptions inside async express routes.

- \*\*nodemon (`^2.0.22`)\*\*: Utility that monitors for any changes in your source and automatically restarts your server.

---

**## \*\*3. Architecture\*\***

**### \*\*3.1. System Architecture Diagram\*\***



\*\*Description of the Diagram:\*\*

1. \*\*Frontend (React JS):\*\*

- Interacts with users through UI components.

- Sends HTTP requests to the backend for authentication and data retrieval.

- Establishes a WebSocket connection using Socket.io for real-time messaging.

2. \*\*Backend (Node.js & Express.js):\*\*

- Handles HTTP requests for authentication, user management, and chat functionalities.

- Manages Socket.io connections for real-time communication.

- Interfaces with MongoDB via Mongoose for data storage and retrieval.

3. \*\*Database (MongoDB):\*\*

- Stores user information, chat rooms, and messages.

4. \*\*Real-Time Communication (Socket.io):\*\*

- Facilitates real-time message exchange between clients through the server.

**### \*\*3.2. Technology Stack\*\***

| Layer | Technology | Description |

|-------------|---------------------------------|------------------------------------------------|

| \*\*Frontend\*\*| React.js, Chakra UI, Socket.io Client | Builds the user interface and handles real-time events. |

| \*\*Backend\*\* | Node.js, Express.js, Socket.io | Manages server-side logic, API endpoints, and real-time communication. |

| \*\*Database\*\*| MongoDB, Mongoose | Stores user data, chat rooms, and messages. |

| \*\*Authentication\*\* | JWT, bcryptjs | Secures user authentication and password storage. |

| \*\*Utilities\*\* | dotenv, nodemon, express-async-handler | Manages environment variables, server restarts, and async error handling. |

---

**## \*\*4. Components and Workflows\*\***

**### \*\*4.1. Authentication Workflow\*\***

\*\*Flow Steps:\*\*

1. \*\*User Registration:\*\*

- User submits registration form with name, email, and password and image.

- Frontend sends a POST request to `/api/user`.

- Backend hashes the password using `bcryptjs` and stores the user in MongoDB.

- Backend generates a JWT token and sends it back to the client.

- Client stores the JWT (e.g., in localStorage) for authenticated requests.

- We use cloudinary to store images.

2. \*\*User Login:\*\*

- User submits login form with email and password.

- Frontend sends a POST request to `/api/user/login`.

- Backend verifies the credentials.

- Upon successful authentication, backend generates a JWT token and sends it to the client.

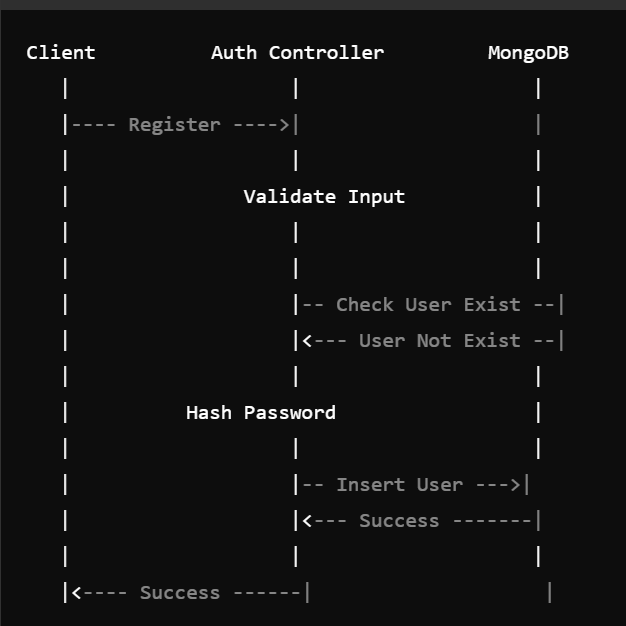
- Client stores the JWT for future authenticated requests.

3. \*\*Protected Routes:\*\*

- Client includes JWT in the Authorization header (`Bearer <token>`) for protected API requests.

- Backend middleware verifies the JWT before granting access to protected resources.

\*\*Authentication Flow Diagram:\*\*



**### \*\*4.2. Chat Workflow\*\***

\*\*Flow Steps:\*\*

1. \*\*Joining a Chat Room:\*\*

- Authenticated user selects or creates a chat room.

- Frontend emits a `join chat` event via Socket.io with room details.

- Backend listens for `join chat` and adds the user to the specified room.

- Backend broadcasts to the room that a new user has joined.

2. \*\*Sending Messages:\*\*

- User types a message and submits.

- Frontend emits a `new message` event with the message content and room ID.

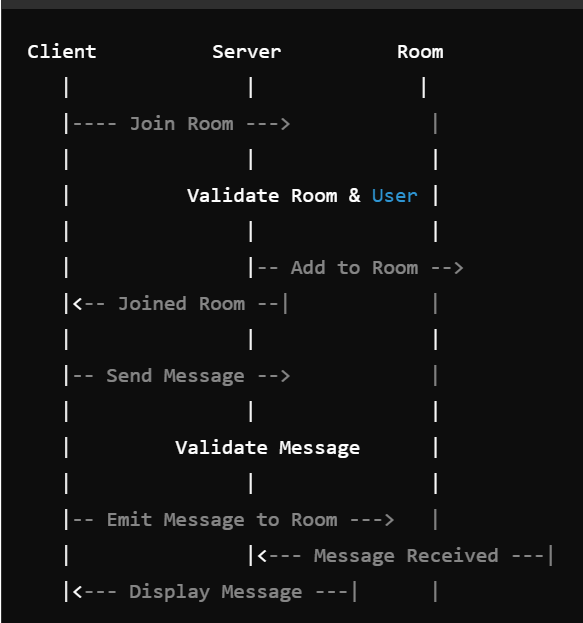
- Backend receives the message, saves it to MongoDB, and broadcasts it to all users in the room via Socket.io.

3. \*\*Receiving Messages:\*\*

- Frontend listens for `message recieved` events.

- Upon receiving a message, the frontend updates the chat interface in real-time.

**\*\*Chat Flow Diagram:\*\***



**### \*\*4.3. Profile Management Workflow\*\***

\*\*Flow Steps:\*\*

1. \*\*Viewing Profile:\*\*

- User navigates to the profile section.

- Frontend fetches user details from the backend using the stored JWT for authentication.

---

**## \*\*5. Database Schema\*\***

### \*\*5.1. User Collection\*\*

\*\*Schema Definition:\*\*

```javascript

const mongoose = require("mongoose");

const bcrypt = require("bcryptjs");

const userSchema = mongoose.Schema(

  {

    name: { type: "String", required: true },

    email: { type: "String", unique: true, required: true },

    password: { type: "String", required: true },

    pic: {

      type: "String",

      required: true,

      default:

        "https://icon-library.com/images/anonymous-avatar-icon/anonymous-avatar-icon-25.jpg",

    },

    isAdmin: {

      type: Boolean,

      required: true,

      default: false,

    },

  },

  { timestaps: true }

);

module.exports = mongoose.model('User', userSchema);

```

### \*\*5.2. Chat Collection\*\*

\*\*Schema Definition:\*\*

```javascript

const mongoose = require("mongoose");

const chatModel = mongoose.Schema(

  {

    chatName: { type: String, trim: true },

    isGroupChat: { type: Boolean, default: false },

    users: [{ type: mongoose.Schema.Types.ObjectId, ref: "User" }],

    latestMessage: {

      type: mongoose.Schema.Types.ObjectId,

      ref: "Message",

    },

    groupAdmin: { type: mongoose.Schema.Types.ObjectId, ref: "User" },

  },

  { timestamps: true }

);

```

---

**## \*\*6. API Design\*\***

### \*\*6.1. Authentication APIs\*\*

| \*\*Endpoint\*\* | \*\*Method\*\* | \*\*Description\*\* | \*\*Request Body\*\* | \*\*Response\*\* |

|---------------------|------------|--------------------------------|------------------------------------------|------------------------------------|

| `/api/user `| `POST` | Register a new user | `{ name, email, password,image }` | `{ token, user }` |

| `/api/user/login` | `POST` | Authenticate user and login | `{ email, password }` | `{ token, user }` |

### \*\*6.2. Chat APIs\*\*

| \*\*Endpoint\*\* | \*\*Method\*\* | \*\*Description\*\* | \*\*Request Body\*\* | \*\*Response\*\* |

|-----------------------|------------|---------------------------------|--------------------------------|------------------------|

| `/api/chat` | `GET` | Retrieve all chat rooms | Headers: `Authorization: Bearer <token>` | `{ chats }` |

| `/api/chat/:chatId` | `GET` | Retrieve messages in a room | Headers: `Authorization: Bearer <token>` | `{ messages }` |

| `/api/chat` | `POST` | Create a new chat room | `{ room }` | `{ chat }` |

### \*\*6.3. User Profile APIs\*\*

| \*\*Endpoint\*\* | \*\*Method\*\* | \*\*Description\*\* | \*\*Request Body\*\* | \*\*Response\*\* |

|-----------------------------|------------|---------------------------------|----------------------------|------------------------|

| `/api/users/profile` | `GET` | Get user profile | Headers: `Authorization: Bearer <token>` | `{ user }` |

---

**## \*\*7. Real-Time Communication\*\***

### \*\*7.1. Event Flow Diagram\*\*



\*\*Description of the Diagram:\*\*

1. \*\*Client A\*\* connects to the server via Socket.io.

2. \*\*Client A\*\* emits a `join\_room` event with the room name.

3. \*\*Server\*\* adds \*\*Client A\*\* to the specified room and emits a `user\_joined` event to all clients in the room.

4. \*\*Client B\*\* in the same room receives the `user\_joined` event.

5. \*\*Client A\*\* sends a message by emitting a `send\_message` event.

6. \*\*Server\*\* receives the message and broadcasts a `receive\_message` event to all clients in the room, including \*\*Client A\*\* and \*\*Client B\*\*.

7. Both clients display the new message in their chat interfaces.

---

**## \*\*8. Security Considerations\*\***

### \*\*8.1. Authentication and Authorization\*\*

- \*\*JWT Tokens\*\*: Securely generate and verify JWT tokens for user authentication. Tokens should be stored securely on the client-side (e.g., HTTP-only cookies or secure storage).

- \*\*Protected Routes\*\*: Implement middleware on the backend to protect sensitive API routes, ensuring only authenticated users can access them.

### \*\*8.2. Password Security\*\*

- \*\*Hashing\*\*: Use `bcryptjs` to hash user passwords before storing them in the database.

- \*\*Salting\*\*: Implement salting to enhance password security against rainbow table attacks.

### \*\*8.3. Data Validation\*\*

- \*\*Input Sanitization\*\*: Validate and sanitize all user inputs to prevent injection attacks.

- \*\*Schema Validation\*\*: Use Mongoose schemas to enforce data integrity.

### \*\*8.4. Secure Communication\*\*

- \*\*HTTPS\*\*: Deploy the application over HTTPS to encrypt data in transit.

- \*\*CORS\*\*: Configure CORS policies to restrict resource access to trusted origins.

### \*\*8.5. Rate Limiting and Throttling\*\*

- \*\*Prevent Brute Force Attacks\*\*: Implement rate limiting on authentication endpoints to mitigate brute force attempts.

- \*\*Socket.io Rate Limiting\*\*: Limit the number of messages a user can send within a specific timeframe to prevent spam.

### \*\*8.6. Environment Variables\*\*

- \*\*Sensitive Data\*\*: Store sensitive information like JWT secret keys, database URIs, and API keys in environment variables using `dotenv`.

- \*\*Configuration Management\*\*: Ensure environment variables are not exposed in the client-side code.

---

**## \*\*9. Scalability and Performance\*\***

### \*\*9.1. Horizontal Scaling\*\*

- \*\*Load Balancing\*\*: Use load balancers to distribute incoming traffic across multiple server instances.

- \*\*Stateless Servers\*\*: Ensure servers are stateless by storing session information in JWTs and databases, facilitating horizontal scaling.

### \*\*9.2. Database Optimization\*\*

- \*\*Indexing\*\*: Implement indexes on frequently queried fields (e.g., email in the User collection) to speed up database operations.

- \*\*Sharding\*\*: For large-scale deployments, consider sharding the MongoDB database to distribute data across multiple machines.

### \*\*9.3. Caching\*\*

- \*\*Redis\*\*: Integrate Redis for caching frequently accessed data, reducing database load and improving response times.

### \*\*9.4. Real-Time Communication Optimization\*\*

- \*\*Socket.io Rooms\*\*: Utilize Socket.io rooms to efficiently manage real-time communication within specific chat rooms.

- \*\*Message Batching\*\*: Implement message batching to handle high-frequency message flows without overwhelming the server.

### \*\*9.5. Frontend Performance\*\*

- \*\*Code Splitting\*\*: Use code splitting and lazy loading in React to reduce initial load times.

- \*\*Optimized Assets\*\*: Compress images and optimize assets to enhance frontend performance.

### \*\*9.6. Monitoring and Logging\*\*

- \*\*Monitoring Tools\*\*: Implement monitoring tools like \*\*PM2\*\*, \*\*New Relic\*\*, or \*\*Datadog\*\* to track application performance and uptime.

- \*\*Logging\*\*: Use structured logging with tools like \*\*Winston\*\* or \*\*Morgan\*\* to capture and analyze logs for troubleshooting.

---

**## \*\*10. User Interface Design\*\***

### \*\*10.1. UI Components\*\*

- \*\*Authentication Pages\*\*:

- \*\*Login\*\*: Form with email and password fields.

- \*\*Register\*\*: Form with name, email, and password fields.

- \*\*Chat Interface\*\*:

- \*\*Chat Rooms List\*\*: Sidebar displaying available users to search for chatting.

- \*\*Chat Window\*\*: Main area displaying messages and input field.

- \*\*Profile Page\*\*:

- \*\*User Information\*\*: Display and edit user details.

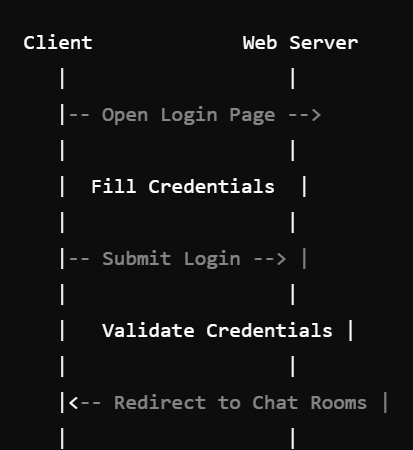
- \*\*Profile Picture\*\*: Upload and display profile images.

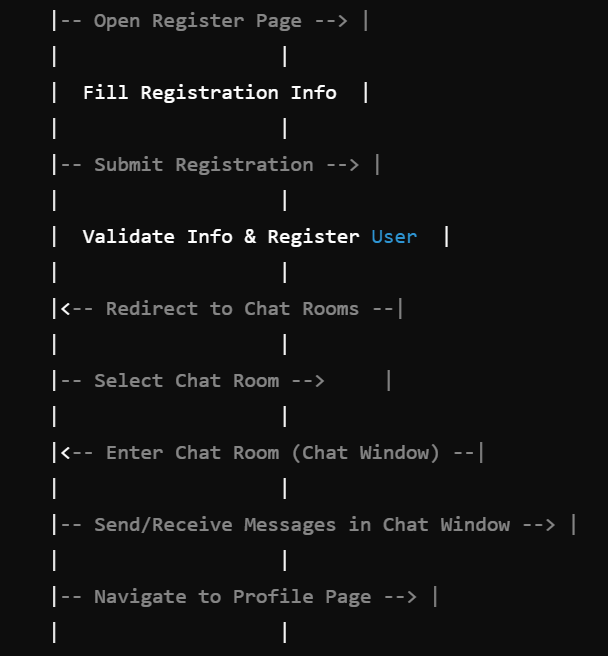
- \*\*Navigation Bar\*\*:

- \*\*Links\*\*: Home, Profile, Logout.

- \*\*Notifications\*\*: Real-time notifications for new messages or users joining.

### \*\*10.2. UI Flow Diagram\*\*





\*\*Description of the Diagram:\*\*

1. \*\*User Accesses Application\*\*:

- If not authenticated, redirected to the \*\*Login\*\* page.

- Option to navigate to the \*\*Register\*\* page.

2. \*\*After Authentication\*\*:

- Redirected to the \*\*Chat Rooms List\*\*.

- Can select a user after searching users to enter the \*\*Chat Window\*\*.

3. \*\*Profile Management\*\*:

- Accessible via the \*\*Navigation Bar\*\*.

- Allows viewing and editing of user profile.

4. \*\*Real-Time Updates\*\*:

- \*\*Chat Window\*\* updates messages in real-time.

- \*\*Navigation Bar\*\* displays notifications for new activities.

---

## \*\*11. Conclusion\*\*

The \*\*BuzzLine Application\*\* built with the MERN stack provides a robust platform for users to communicate seamlessly in real-time. By leveraging technologies like \*\*Socket.io\*\* for instant messaging, \*\*JWT\*\* for secure authentication, and \*\*Chakra UI\*\* for an intuitive interface, the application ensures both functionality and user experience. The system design emphasizes scalability, security, and performance, making it well-suited for handling a growing user base and high message throughput. Proper deployment strategies and continuous monitoring further ensure the application's reliability and maintainability.

---

## \*\*Appendix\*\*

### \*\*A. Glossary\*\*

- \*\*MERN Stack\*\*: A combination of MongoDB, Express.js, React.js, and Node.js used for full-stack web development.

- \*\*Socket.io\*\*: A JavaScript library for real-time web applications.

- \*\*JWT (JSON Web Token)\*\*: A compact, URL-safe means of representing claims to be transferred between two parties.

- \*\*Chakra UI\*\*: A simple, modular, and accessible component library that gives you the building blocks to build React applications.

- \*\*CORS (Cross-Origin Resource Sharing)\*\*: A mechanism to allow or restrict requested resources on a web server depending on where the HTTP request was initiated.

### \*\*B. References\*\*

- [MERN Stack Tutorial](https://www.mongodb.com/mern-stack)

- [Socket.io Documentation](https://socket.io/docs/v4/)

- [Chakra UI Documentation](https://chakra-ui.com/docs/getting-started)

- [Express.js Documentation](https://expressjs.com/)

- [React.js Documentation](https://reactjs.org/docs/getting-started.html)

- [MongoDB Mongoose Documentation](https://mongoosejs.com/docs/guide.html)

---

\*This document provides a comprehensive overview of the system design for a real-time chat application using the MERN stack. For detailed implementation guides, refer to the respective sections and external resources mentioned in the references.\*

**\*\*Finally I am attaching an image to view my workflow of the application:::**

**Link::** [**https://whimsical.com/buzzline-Urud7vtvxqs96C6NJqJhH2**](https://whimsical.com/buzzline-Urud7vtvxqs96C6NJqJhH2)

****