REAL TIME CHAT APPLICATION

A PROJECT REPORT

by

Simran Tyagi (202410116100208)
Shefali Yadav (202410116100195)
Tanisha Jain (202410116100217)
Vishal Chaturvedi (202410116100248)

Session:2024-2025 (II Semester)

Under the supervision of
Ms. Shruti Aggarwal
Assistant Professor

KIET Group of Institutions, Delhi-NCR, Ghaziabad

DEPARTMENT OF COMPUTER APPLICATIONS
KIET GROUP OF INSTITUTIONS, DELHI-NCR,
GHAZIABAD-201206
(2024- 2025)

ABSTRACT

The Real-time Chat Application is a web-based platform designed to enable users to communicate instantly in a dynamic, interactive, and secure environment. The primary objective of this application is to provide seamless, real-time messaging capabilities, allowing users to send and receive messages without delay, while also supporting features like private and group chats, message notifications, and user presence statuses. The application leverages modern web technologies, including React.js for the frontend, Node.js and Express.js for the backend, Socket.io for real-time communication, and MongoDB for data storage.

In this project, users can register, authenticate, and engage in one-on-one or group conversations. Real-time message delivery and user status (online/offline) are powered by Socket.io, which facilitates continuous communication between the client and server. The user interface is built with React.js, providing an intuitive and responsive design, while Tailwind CSS and Daisy UI ensure a clean and visually appealing layout.

The application focuses on both security and scalability, employing techniques such as JWT (JSON Web Tokens) for secure authentication and Bcrypt.js for password hashing. Data is securely stored in MongoDB, and the system is designed to scale efficiently to accommodate a growing user base.

This project not only demonstrates real-world application development but also provides valuable experience with key technologies used in building scalable, real-time web applications. The Real-time Chat Application is expected to enhance communication for individuals and groups while providing a reliable, engaging, and secure platform for modern online communication.

TABLE OF CONTENTS

1. Introduction

In today's fast-paced digital world, instant communication has become a fundamental necessity. The rapid growth of the internet and advancements in technology have led to a surge in demand for real-time chat applications. These applications serve as essential tools for both personal and professional communication, enabling users to send and receive messages instantly without delays. Traditional messaging systems, such as SMS and emails, often lack the efficiency and speed required for modern interactions. To bridge this gap, real-time chat applications leverage WebSocket's and Socket.io to provide seamless, bi-directional communication between users.

This project focuses on the development of a **Real-time Chat Application** that ensures secure, fast, and user-friendly communication. It integrates features like one-to-one messaging, group chats, real-time notifications, and online presence tracking.

Tailwind CSS, and Daisy UI, ensuring a scalable and responsive design. Security is a critical aspect, and measures such as end-to-end encryption, authentication, and authorization are incorporated to protect user data. By implementing this application, we aim to enhance user engagement and provide an efficient platform for seamless communication in various domains, such as businesses, education, and social networking.

2. Literature Review

Real-time chat applications have become an integral part of modern communication, facilitating instant messaging across various platforms. These applications are widely used in social networking, customer support, and enterprise communication. This literature review explores the existing research and technological advancements in real-time chat applications, focusing on architecture, protocols, security, and performance optimization. The development of real-time chat applications has evolved significantly over the years. Early communication systems relied on SMS and email, which lacked instant interaction. With the advancement of web technologies, platforms like WhatsApp, Messenger, and Slack have introduced instant messaging features that provide real-time communication using WebSocket's and similar protocols.

The evolution of real-time chat applications can be traced from basic text-based messaging systems to complex, multimedia-rich platforms supporting voice, video, and AI-driven interactions. Early messaging systems relied on **Simple Mail Transfer Protocol (SMTP)** for email-based communication and **Internet Relay Chat (IRC)** for text-based chat (Gao et al., 2020). However, these early methods lacked real-time responsiveness. The introduction of **WebSockets and Server-Sent Events (SSE)** revolutionized real-time chat applications by enabling persistent connections, reducing latency, and improving scalability (Riyadh et al., 2021).

WebSocket's enable continuous bidirectional communication between the client and the server, reducing latency and enhancing user experience. Socket.io, a popular JavaScript

library, simplifies the implementation of WebSocket's, ensuring efficient message transmission and handling multiple concurrent users. Additionally, advancements in security protocols, including end-to-end encryption and authentication mechanisms such as JWT and OAuth, have improved data security in modern chat applications.

They have evolved significantly, driven by advancements in networking, security, and AI-powered automation. While WebSockets and MQTT dominate as efficient communication protocols, new technologies like QUIC and gRPC are improving performance and scalability. Security remains a critical concern, with encryption and authentication mechanisms playing a vital role in protecting user data. The future of chat applications lies in decentralized communication, AI integration, and enhanced privacy measures.

This project builds upon existing technologies while integrating a secure and scalable architecture. By leveraging **React.js, Node.js, MongoDB, and Socket.io**, the proposed system aims to provide a reliable, fast, and user-friendly chat experience that meets the growing demands of real-time communication.

3. Project Objective

The objective of the **Real-time Chat Application** project is to develop a web-based communication platform that allows users to engage in instant messaging with each other in real-time. The application aims to provide seamless interaction and facilitate communication through various features, such as private and group chats, message notifications, and real-time status updates.

Key objectives include:

1. **Real-Time Messaging**: Implementing real-time communication using technologies like **Socket.io**, ensuring messages are sent and received instantly without needing to refresh the page.
2. **User Authentication**: Allowing users to register, log in, and manage their profiles securely using **Node.js** for server-side logic and **MongoDB** for storing user information.
3. **Private and Group Chats**: Enabling users to have one-on-one private conversations and create group chats to communicate with multiple users simultaneously.
4. **User Presence**: Displaying user statuses (online/offline) in real time, providing a dynamic user experience.
5. **Message Notifications**: Implementing notification features for new messages and user activities, ensuring users stay informed.
6. **Responsive Design**: Using **Tailwind CSS** and **Daisy UI** to ensure the application works efficiently across different devices and screen sizes.
7. **Scalability**: Designing the application to handle multiple concurrent users efficiently.

4. Hardware and Software Requirements

This project will allow you to gain hands-on experience with **MongoDB**, **React.js**, **Node.js**, and **Socket.io**, while working on a real-world application that requires the integration of backend and frontend technologies for real-time communication.

For your **Real-time Chat Application**, you will need specific hardware and software components to ensure smooth development and deployment. Here's a list of the hardware and software requirements:

Hardware Requirements:

1. **Development Machine:**
    - **Processor:** Intel Core i5 or equivalent (minimum) for smooth development.
    - **RAM:** 8GB or more for running multiple applications (IDE, browser, etc.) efficiently.
    - **Storage:** At least 10GB of free space for development tools, dependencies, and project files.
    - **Network:** A stable internet connection for downloading libraries, packages, and real-time communication.
2. **Server (for Deployment, optional if hosting locally):**
    - **Processor:** Dual-core processor or better for hosting your application.
    - **RAM:** 4GB minimum for running the backend server and handling multiple concurrent users.
    - **Storage:** 20GB SSD or higher for faster data access and app deployment.

- **Network:** High-speed internet for real-time chat communication and user connectivity.

Software Requirements:

1. **Operating System:**
    - **Windows 10/11**, **macOS**, or **Linux (Ubuntu)** for development and server deployment.
2. **Development Tools:**
    - **IDE/Text Editor:** Visual Studio Code (VS Code), Sublime Text, or any preferred editor.
    - **Version Control:** Git for managing code versioning and collaboration.
    - **Browser:** Google Chrome, Mozilla Firefox, or any modern browser for testing the app.
3. **Backend Technologies:**
    - **Node.js**: Runtime environment for running JavaScript on the server-side.
    - **Express.js**: Web framework for handling routing and requests.
    - **Socket.io**: For real-time communication and establishing a WebSocket connection between clients and server.
    - **MongoDB**: NoSQL database for storing user data, chat logs, and other relevant information.
    - **Mongoose**: ODM for interacting with MongoDB from the Node.js environment.
4. **Frontend Technologies:**
    - **React.js**: JavaScript library for building the user interface of the chat application.

- **Tailwind CSS**: For building responsive and flexible UI components.
- **Daisy UI**: Component library for building aesthetically pleasing UI elements with Tailwind CSS.

5. **Additional Tools and Libraries:**
    - **npm (Node Package Manager)**: For managing dependencies.

6. **Deployment:**
    - **Vercel**: Cloud platforms for deploying your Node.js application.
    - **MongoDB Atlas**: Cloud service for hosting MongoDB databases (if you prefer not to manage your own database).

5. Project Flow

The project flow for the **Real-time Chat Application** will be structured as follows:

1. **Requirement Gathering:**
    - Define user stories and functionalities (user authentication, real-time messaging, private/group chats).
    - Analyze the target audience and determine features like user presence, notifications, and message history.
2. **System Design:**
    - Design the architecture, focusing on a client-server model.
    - Choose technologies: **React.js** for frontend, **Node.js** with **Express.js** for backend, and **MongoDB** for data storage.
    - Define data models (e.g., user model, chat history model) and set up the database schema in **MongoDB**.
3. **Frontend Development:**
    - Create UI components for login, registration, chat interface, and user settings using **React.js**.
    - Style the application using **Tailwind CSS** and **Daisy UI** for responsive layouts.
    - Implement client-side logic for real-time communication using **Socket.io**.

4. **Backend Development:**

    - Set up the **Node.js** server with **Express.js** to handle routes (user authentication, chat messages, etc.).
    - Implement real-time communication using **Socket.io** to facilitate message delivery and notifications.
    - Set up **MongoDB** to store user data and chat history, connecting it with **Mongoose** for seamless data interaction.

5. **Testing:**

    - Test the application in various scenarios (single user, multiple users, concurrent messages).
    - Test real-time message delivery, user status updates, and group chats.
    - Perform security testing to ensure data protection, especially for user credentials and messages.

6. **Deployment:**

    - Deploy the application to a cloud platform (e.g., **Vercel**) for production.
    - Ensure the application is scalable and handles multiple users simultaneously.
    - Configure **MongoDB Atlas** for cloud database hosting.

7. **Maintenance and Updates:**

    - Monitor the application for any bugs or performance issues.

6. Project Outcome

The expected outcomes of the **Real-time Chat Application** project are:

1. **Fully Functional Chat Application:**

    - A real-time chat platform where users can send and receive messages instantly.
    - Users can register, log in, and maintain their profiles securely.
    - Group chats and private conversations will be fully supported.

2. **Scalable and Responsive System:**

    - The application will be responsive, providing an optimal experience across devices (desktop, tablet, mobile).
    - Backend will be scalable to handle a large number of concurrent users, leveraging technologies like **Socket.io** for real-time communication.

3. **User Engagement:**

    - The application will feature real-time notifications, user presence status (online/offline), and message history to improve user engagement.

4. **Security:**

    - Secure user authentication using **JWT** and password hashing with **Bcrypt.js**.

- Chat data will be securely stored in **MongoDB**

5. **Administrative & Moderation Features:**

    - Admin panel to manage **users, groups, and reported messages**.
    - Content moderation tools to **filter inappropriate content and prevent abuse**.
    - Role-based access control (**RBAC**) for managing permissions.

6. **Future Enhancements & Integrations:**

    - **Voice and video calling support** for a more interactive experience.
    - Integration with **third-party services** (e.g., Google, Facebook for login).
    - AI-powered **chatbots** and **auto-reply features** for automated responses.
    - Multi-language support to **enhance accessibility** for global users.

7. Proposed Time Duration

The estimated time duration for the **Real-time Chat Application** project is **3-4 months**. Here's a breakdown:

1. **Week 1-2: Requirement Analysis & System Design**
    - Defining features, architecture, and tech stack.
    - Database schema design and planning user flows.
2. **Week 3-5: Frontend Development**
    - Setting up the React.js project.
    - Building UI components (login, chat interface, etc.) using **Tailwind CSS** and **Daisy UI**.
3. **Week 6-8: Backend Development**
    - Setting up **Node.js** and **Express.js**.
    - Implementing real-time messaging with **Socket.io**.
    - Database integration with **MongoDB**.
4. **Week 9-10: Testing & Debugging**
    - Testing application flow, message delivery, and UI responsiveness.

- Fixing bugs and ensuring everything works as expected.

5. **Week 11-12: Deployment & Documentation**

    - Deploying the application to a cloud platform like **Heroku** or **Vercel**.

    - Writing user and developer documentation.

6. **Week 13-14: Maintenance & Updates**

    - Monitoring the application in production, gathering feedback, and implementing improvements.

8. Features & Functions

🐦Flappy Bird Game

1. Core Game Mechanics

- **Canvas-based Rendering**: The game utilizes the `<canvas>` element for real-time graphics rendering, allowing smooth bird movement, pipe generation, and collision visuals.
- **Physics Simulation**: The bird's flight mimics gravity and jumping behavior using `velocity`, `gravity`, and `jumpStrength`. This adds a realistic feel to gameplay.
- **Procedural Obstacles**: Pipes are generated randomly at intervals based on difficulty, making every session unique and challenging.
- **Collision Detection**: Collision is continuously checked between the bird and pipes or screen edges. If a collision occurs, the game ends.

2. Scoring & Progress Tracking

- **Live Scoring**: Players earn points by successfully passing through pipes. Score is updated in real-time.
- **High Score Storage**: The highest score is saved locally using `localStorage`, allowing users to track their personal best even after refreshing.

3. Game Control & State Management

- **Start, Pause, Resume, Restart**:
  - Game starts on click or spacebar.
  - Can be paused with the "Escape" key or button.
  - Displays "Game Over" and allows replay.
- **Keyboard & Click Controls**:
  - Pressing the spacebar or tapping on the screen makes the bird jump.
  - Escape key toggles pause/resume.

4. Difficulty Levels

- Users can choose between **Easy**, **Medium**, or **Hard**, affecting:
  - Pipe speed
  - Pipe frequency
  - Gap height between pipes
  - Bird's gravity and jump strength
- Difficulty selection dynamically adjusts game logic using `useEffect`.

5. User Interface & Design

- **React.js Component-Based Structure**: Game logic and rendering are encapsulated in a functional component using hooks (`useState`, `useEffect`, `useRef`).
- **Responsive Design with TailwindCSS & Daisy UI**: The layout is mobile-friendly and styled cleanly with utility-first classes.
- **Lucide Icons**: Icons like Play, Pause, and Refresh enhance UI intuitiveness.

6. Game States

- **GameStarted**: Indicates if the game has begun.
- **GameOver**: Tracks when the game ends and shows the relevant UI.
- **GamePaused**: Pauses the game loop without resetting the game.

🚦 Traffic Racer Game

1. Core Gameplay Mechanics

- **Player Movement**: Players control a car that can move left or right using the arrow keys, giving full control over dodging obstacles.
- **Obstacle Generation**: Randomly placed vehicles (obstacles) move downward toward the player's car. Each one appears in different lanes and colors to add variation.
- **Collision Detection**: The game checks for overlapping positions between the player's car and obstacles. If a collision occurs, the game ends immediately with a crash message.

2. Scoring System

- **Dynamic Scoring**: Players earn points for each obstacle generated (i.e., distance survived).
- **High Score Persistence**: The highest score is stored in `localStorage`, letting users track progress across multiple game sessions.

3. Difficulty Levels

- The game offers three levels of difficulty:
  - **Easy**: Slower obstacle speed and less frequent traffic.
  - **Medium**: Moderate speed and more frequent obstacles.
  - **Hard**: High-speed gameplay with frequent obstacle spawning.
- Difficulty settings affect both the **speed** and **generation rate** of obstacles, increasing the challenge as needed.

4. Game State Management

- **GameStart/GameOver**: Manages whether the game has started or ended.
- **Pause/Resume**: Players can pause the game using the **Escape** key or a dedicated button. The game loop halts until resumed.
- **Restart Option**: After crashing, the player can restart with a button press.

5. Visuals and Interface

- **Canvas-Based Rendering**: All visual elements (player, obstacles, road, markings) are dynamically drawn on an HTML5 `<canvas>`, ensuring smooth animations.
- **Animated Road Markings**: The center and side road lines move to simulate motion, enhancing realism.
- **Colorful Cars**: Obstacle cars appear in random colors to keep the visuals interesting.
- **Tailwind CSS + Daisy UI**: Used to style the game controls and layout for responsiveness and visual consistency.
- **Lucide Icons**: Adds clean, intuitive UI buttons for Pause, Play, and Restart.

6. Control Scheme

- **Arrow Keys**: Move the player's car left or right.
- **Spacebar**: Start the game.
- **Escape**: Pause/resume during gameplay.

🐍Snake Game

1. Classic Gameplay with Modern Enhancements

- **Grid-Based Movement**: The game operates on a classic grid layout using `canvas`, where each cell is 20x20 pixels. The snake moves one cell at a time, mimicking the original Snake game logic.
- **Food Mechanism**: Randomly placed food appears on the grid. When the snake eats the food, it grows longer and the score increases.

2. Dynamic Snake Behavior

- **Self-Collision Detection**: The game ends if the snake collides with itself.
- **Wall Collision**: If the snake hits the edge of the canvas (the wall), the game ends.
- **Real-Time Direction Handling**: Players control the snake using arrow keys, and rapid key presses are handled using `nextDirectionRef` to queue direction changes. 180-degree turns are prevented.

3. Difficulty Customization

- The game offers 3 levels of difficulty:
  - **Easy**: Snake moves slowly (150ms per move).
  - **Medium**: Moderate speed (100ms).
  - **Hard**: Fast gameplay (70ms).
- Players can select the difficulty before starting the game, which dynamically adjusts the snake's movement speed using `speedRef`.

4. Game State Management

- **Start, Pause, Resume, and Restart**:
  - Start or restart the game using the spacebar or button.
  - Pause/resume with the Escape key.
  - Game over UI is shown when the snake crashes.
- **High Score Tracking**: Scores are stored in `localStorage` and updated when the player beats their previous record.

5. Mobile-Friendly Controls

- **Touch Support**: Includes directional touch buttons (↑, ↓, ←, →) for mobile devices, allowing full control without a keyboard.
- **Pointer-safe UI**: Visual arrows are semi-transparent and non-intrusive but functional for gameplay.

6. Visual Elements & UI

- **Canvas-based Graphics**:
    - The snake and food are drawn on an HTML5 canvas using different colors.
    - The snake head has dynamic **eye rendering** based on direction.
- **Gradient Effect on Snake Body**: Each segment of the snake's body gradually changes shade for a modern visual aesthetic.
- **TailwindCSS + DaisyUI Styling**: Buttons, UI overlays, and layout are styled responsively with Tailwind and Daisy UI.
- **Game Grid Lines**: Optional visual grid using thin stroke lines for better spatial awareness.

7. Input Controls

- **Arrow Keys**: Used to control the snake's direction.
- **Spacebar**: Starts the game.
- **Escape**: Pauses/resumes the game.
- **Touch Buttons**: Functional on-screen directional controls for mobile users.

+◯ Tic-Tac-Toe Game

1. Classic Turn-Based Gameplay

- **2-Player Local Match**: The game supports two players (X and O) taking alternate turns.
- **Win Detection**: The game checks for winning combinations across rows, columns, and diagonals after each move.
- **Draw Recognition**: If the board fills with no winner, the game automatically ends in a draw.

2. Game Logic & UI

- **Dynamic Game Board**: Built using a `3x3` grid with individual square components.
- **Next Player Indicator**: Real-time text shows which player's turn it is (X or O).
- **Winner Announcement**: If a player wins, a victory message is displayed instantly.
- **Color-Coded Moves**:
  - "X" is displayed in a primary color.
  - "O" is shown in a secondary color, improving visual clarity.

3. State Management

- **React State Hooks**:
  - `board`: Holds the state of all 9 squares.
  - `xIsNext`: Tracks which player's turn it is.
  - `winner`: Stores the winner, if any.
  - `gameStatus`: Displays dynamic messages (winner, draw, next move).
- All state changes automatically trigger re-renders to reflect the current game status.

4. Reset & Replay

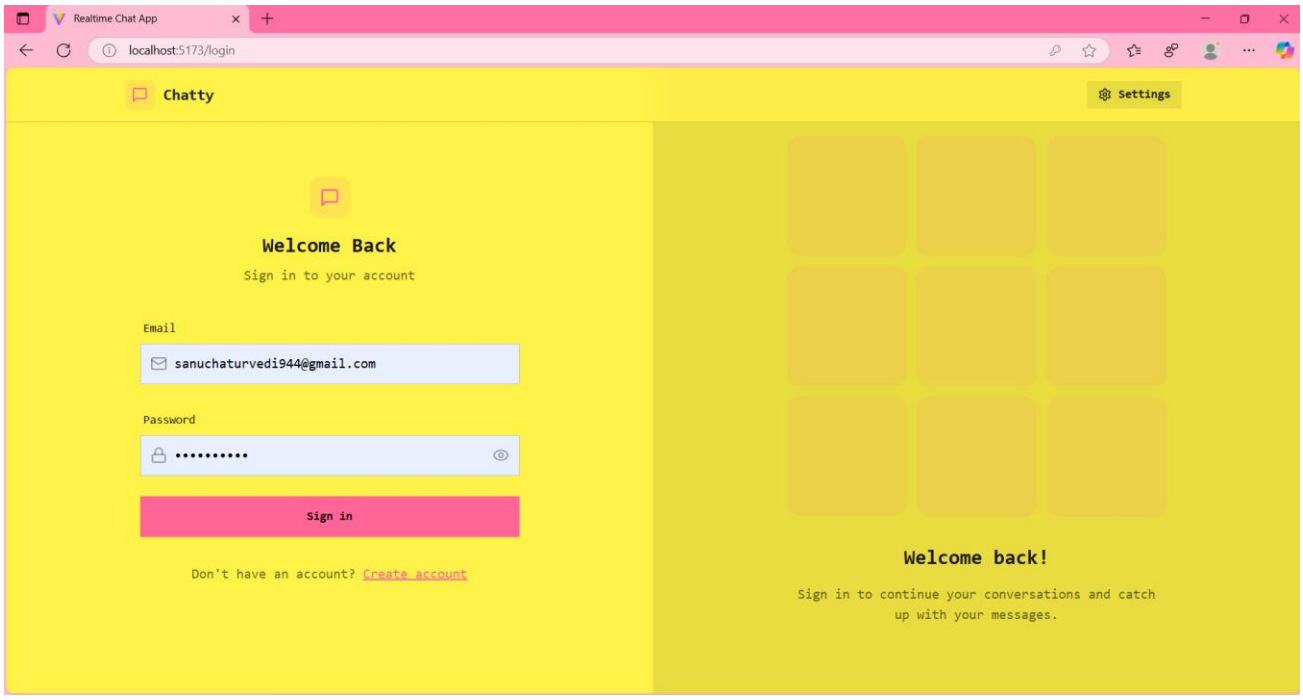
- **Reset Button**: Resets the game board and score with a single click.
- **Visual Feedback**: Disables squares after win or draw to prevent further moves.

5. Responsive UI Design

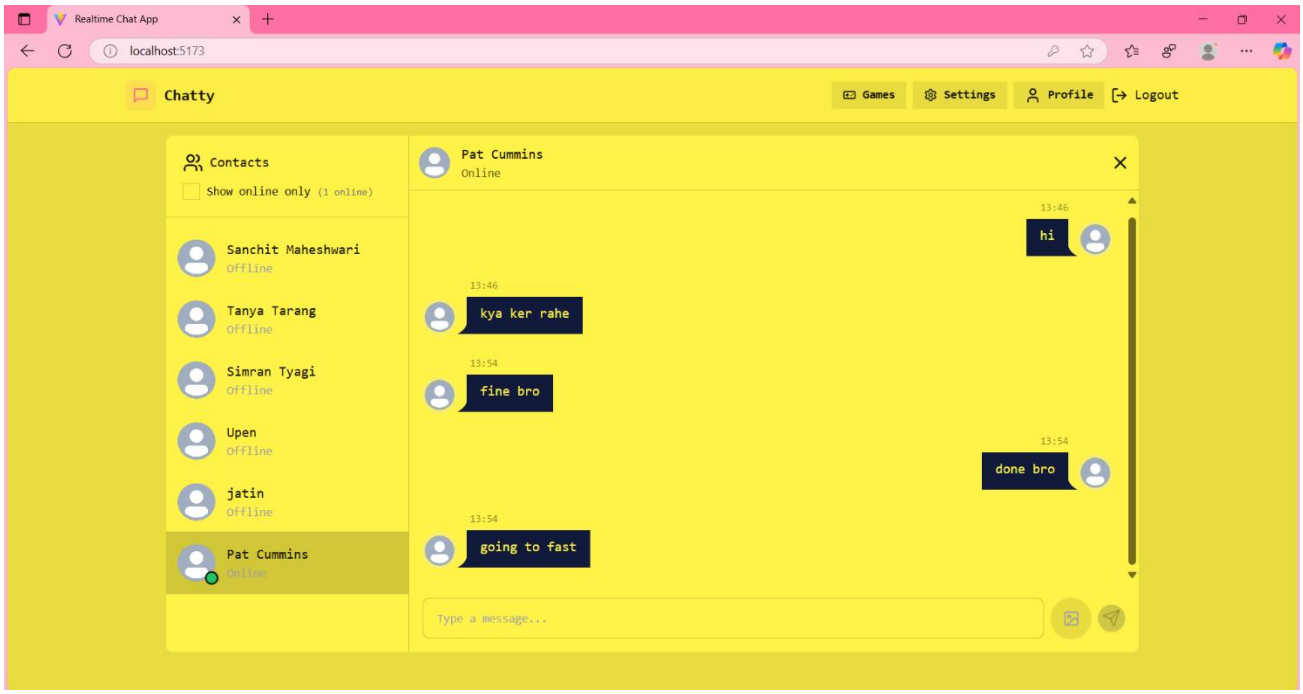
- **TailwindCSS + Daisy UI Styling**:
  - Responsive square sizes (`size-16 sm:size-20`) for consistent layout on mobile and desktop.
  - Rounded edges, smooth hover transitions, and color themes for modern aesthetics.

9 - Snapshots

Login page



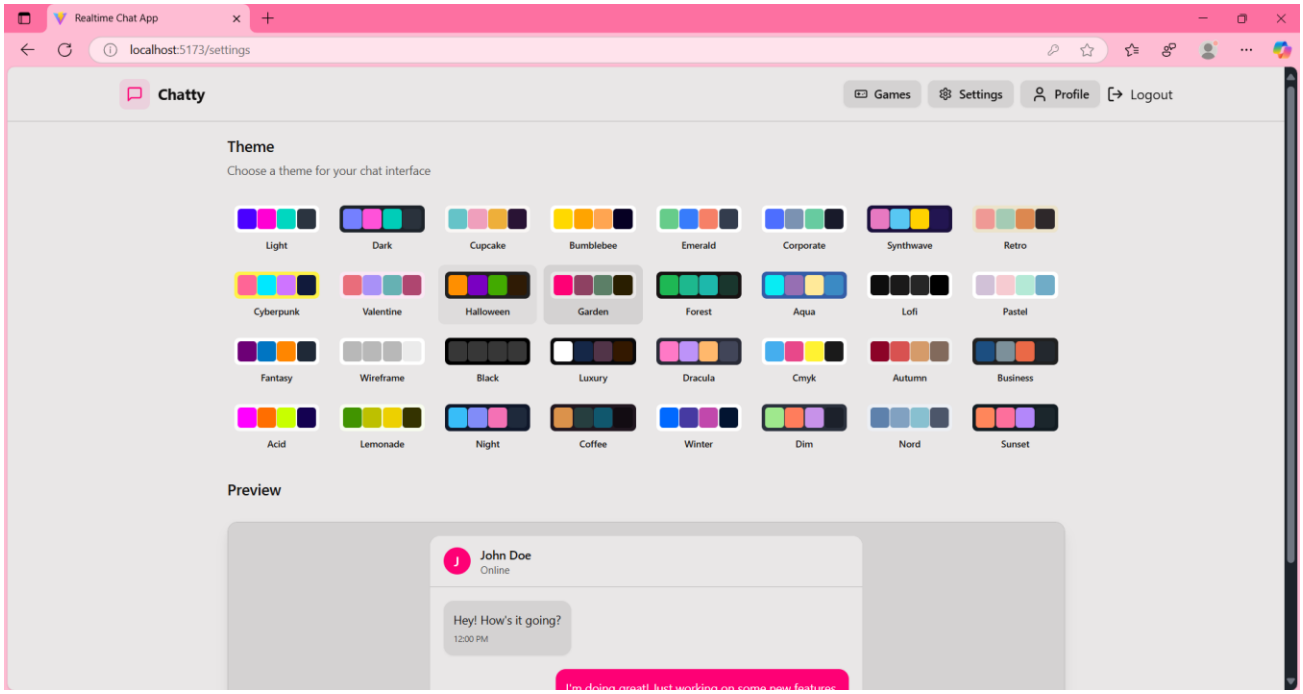
Chat interface



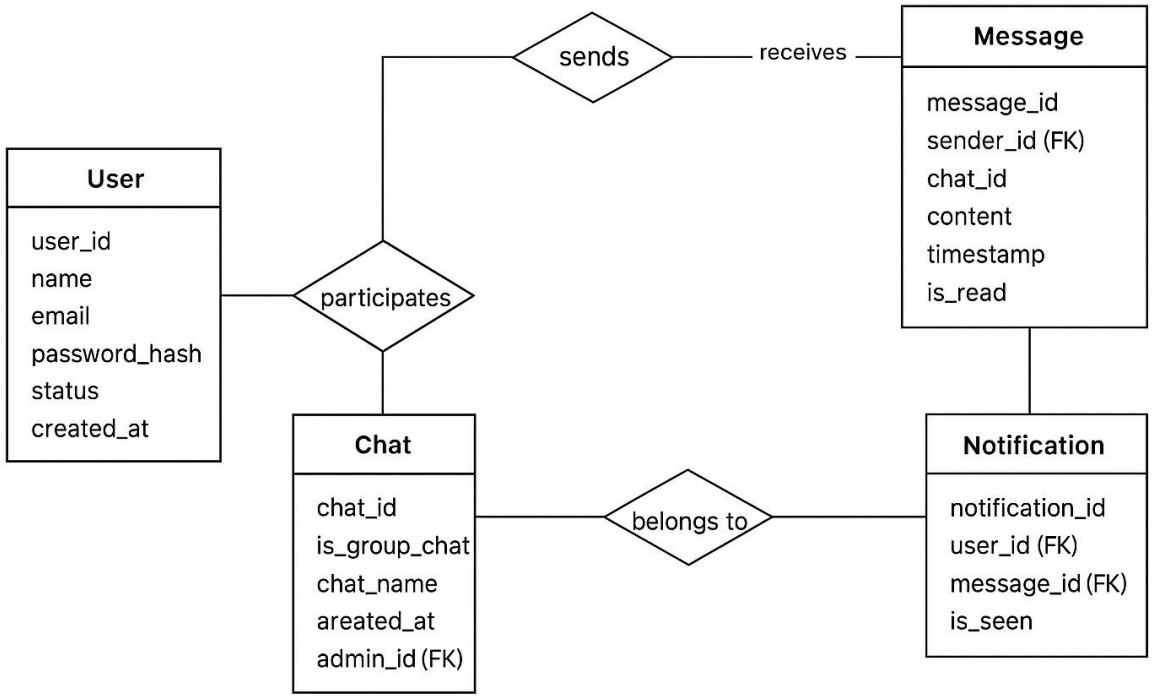
TIC TAC TOE

Traffic Racer

Themes



10 E-R DIGRAM


sends
receives
Message
message_id
sender_id (FK)
chat_id
content
timestamp
is_read
User
user_id
name
email
password_hash
status
created_at
participates
Chat
chat_id
is_group_chat
chat_name
areated_at
admin_id (FK)
belongs to
Notification
notification_id
user_id (FK)
message_id (FK)
is_seen

11 References / Bibliography

1. **Tarek S. S. and K. El-Fakih**, "A Survey of Real-Time Web Applications Using WebSocket's: Real-Time Chat Applications," *International Journal of Computer Applications*, Vol. 112, No. 15, pp. 1-7, February 2015, DOI: 10.5120/19719-4737.

2. **Georgios Chitimacha, Vassilis B. D. and E. T. Nardelli**, "Real-Time Web Applications: Development, Challenges and Future Directions," *IEEE Access*, Vol. 8, pp. 124328–124346, June 2020, DOI: 10.1109/ACCESS.2020.3003613.

3. **Michael S. and Robert J. B.**, "Real-Time Communication using WebSocket's and Node.js," *Proceedings of the 13th International Conference on Computer Engineering*, pp. 314-321, March 2017, DOI: 10.1109/ICEC.2017.133.

4. **Sandeep K. and Arun K. Gupta**, "A Comparative Survey on WebSocket Protocol for Real-Time Communication in Web Applications," *International Journal of Computer Science and Information Technology*, Vol. 8, No. 4, pp. 14-23, April 2019, DOI: 10.5120/ijcsit5934-4056.

5. **Node.js Documentation**, "Node.js: A JavaScript Runtime Built on Chrome's V8 JavaScript Engine," *Node.js Foundation*, 2021, Available: https://nodejs.org/en/docs/

6. **Socket.io Documentation**, "Socket.io: Real-Time WebSocket's," *Socket.io Inc.*, Available: https://socket.io/docs/

7. **React.js Documentation**, "React: A JavaScript Library for Building User Interfaces," *React*, 2021, Available: https://reactjs.org/docs/getting-started.html.

8. **Tailwind CSS Documentation**, "Tailwind CSS: A Utility-First CSS Framework," *Tailwind Labs*, Available: https://tailwindcss.com/docs.