



COLLEGE CODE :- 9509

COLLEGE NAME :- Holy Cross Engineering College

DEPARTMENT :- CSE

STUDENT NM-ID:-

992FDAD7A0AF875A0484BF88BC92B3E8

DATE :-15/09/2025

**Completed the project named as
Phase__TECHNOLOGY PROJECT NAME :**

CHAT UI APPLICATION

SUBMITTED BY,

**NAME :- Nisanth.V
MOBILE NO :- 8438747568**

1. Tech Stack Selection

The chosen stack prioritizes developer experience, performance, and a rich ecosystem.

Layer	Technology	Justification
Frontend Framework	React 18+ with TypeScript	Component-based architecture is ideal for UI. TypeScript ensures type safety, reducing bugs and improving developer collaboration.
Build Tool	Vite	Faster builds and Hot Module Replacement (HMR) than alternatives like Create React App, leading to a better development experience.
Styling	Tailwind CSS	Utility-first CSS framework allows for rapid, consistent UI development without context switching. Easy to create responsive designs.
State Management	Zustand (Primary) / React Context (Theming)	Zustand is a simple, unopinionated state management solution perfect for managing chat state (messages, rooms, online users). Lightweight and easy to use. Context is sufficient for theming.
Data Fetching	TanStack Query (React Query)	Handles server state, caching, background updates, and pagination effortlessly. Perfect for fetching message history and user lists.
Real-Time Communication	WebSocket (via socket.io-client)	The standard for bidirectional, real-time communication. Socket.io provides a robust client library and fallback mechanisms.

Layer	Technology	Justification
UI Components	Headless UI or Radix UI	For accessible unstyled components (dropdowns, modals). We'll style them with Tailwind, maintaining full design control.
Testing	Jest (Unit) + React Testing Library	Industry standard for testing React components and logic.
Deployment	Vercel / Netlify	Zero-config deployment, perfect CI/CD integration with Git, and excellent performance for SPAs.

UI Structure (Wireframe Breakdown)

The UI is divided into three primary columns:

- 1. **Sidebar/Navigation:** Contains user profile, list of channels/rooms, and direct message conversations.
- 2. **Messages Panel:** The main area displaying the message history for the selected conversation.
- 3. **Members Sidebar (Optional):** Displays the list of users currently online in the selected room/channel.

Component Tree:

text

<App>



```

|   |— <ConversationList> (Channels & DMs)
|   |   |— <ConversationListItem />
|   |   |— <CreateRoomButton />
|
|— <MainContent>
|   |— <MessageHeader> (Room name, topic, users online)
|   |— <MessageList>
|   |   |— <MessageBubble />
|   |— <TypingIndicator />
|   |— <MessageInput />
|
|— <MembersSidebar> (Conditionally Rendered)
    |— <MemberListItem />

```

API Schema Design (Key Endpoints & WebSocket Events)

We assume a RESTful API for initial data fetching and WebSockets for real-time updates.

REST Endpoints (via React Query):

- GET /api/conversations - Fetch the user's list of conversations (rooms & DMs).
- GET /api/messages?conversationId=:id&page=:page - Fetch paginated message history for a conversation.
- GET /api/users/me - Fetch current user's profile.
- GET /api/users?conversationId=:id - Fetch members of a specific conversation.

WebSocket Events (via [Socket.IO](#)):

• Client -> Server:

- join_room ({ roomId: string }) - Join a specific room to receive messages.
- send_message ({ roomId: string, content: string }) - Send a new message.
- typing_start ({ roomId: string }) - Notify others that the user is typing.
- typing_stop ({ roomId: string }) - Notify others that the user stopped typing.

• Server -> Client:

- message ({ id: string, userId: string, content: string, timestamp: Date }) - Receive a new message.
- user_joined ({ userId: string }) - A user joined the current room.
- user_left ({ userId: string }) - A user left the current room.
- user_typing ({ userId: string, username: string }) - A user is typing.

- `user_stopped_typing ({ userId: string })` - A user stopped typing.
-

3. Data Handling Approach

- **Server State (Messages, Conversations, Users):**

- Managed by **TanStack Query**.
- Messages are fetched paginated. `useInfiniteQuery` will be used for seamless infinite scrolling.
- The cache will be invalidated and refetched when needed (e.g., after sending a message or returning to an old conversation).

- **Real-Time State (New Messages, Typing Indicators, Online Users):**

- Managed by the **Zustand** store.
- When a message event is received via WebSocket, it is appended to the appropriate message list in the store *optimistically*. This provides instant UI feedback.
- Typing statuses and online user lists are also held in the Zustand store and updated via WebSocket events.

- **UI State (Selected Conversation, Modal Open/Close, Theme):**

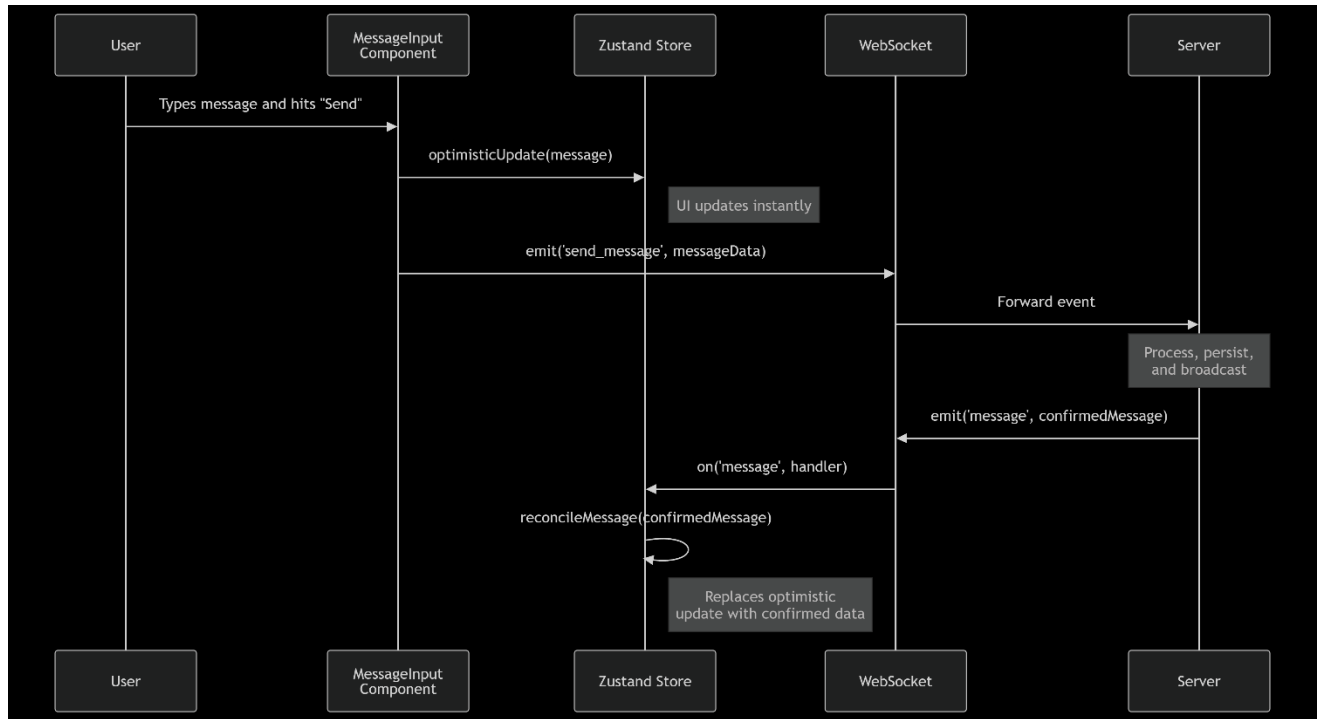
- Local component state (`useState`) for isolated states (e.g., is a dropdown open?).
- Zustand or Context for global UI state like the currently selected `conversationId` or theme preference.

- **Optimistic Updates:**

- Crucial for a responsive chat experience.
 - **Process:** 1) User sends a message. 2) Message is immediately added to the UI (Zustand store). 3) The `send_message` event is emitted. 4) On acknowledgement from the server, the message is marked as "delivered" or an error is shown.
-

4. Component / Module Diagram

This diagram shows the hierarchy and data flow between the main components and external services.



5. Basic Flow Diagram

Primary User Flow: Sending a Message

This sequence diagram illustrates the process of a user sending a message, showcasing the interaction between UI components, state, and the backend.

