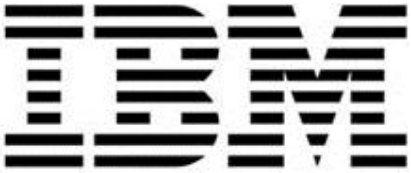


COLLEGE CODE : 9504

COLLEGENAME:DR.G.U.POPECOLLEGE OF ENGINEERING

DEPARTMENT : COMPUTER SCIENCE AND ENGINERRING

STUDENT NM ID : D1468F085E9D450B2A34825AF30855B5

ROLL NO : 11

DATE : 15.09.2025

Completed the Phase-02

PROJECT NAME : IBM-FE- CHAT APPLICATION UI

SUBMITTED BY,

HEPZIBAH E
9843043546

Solution Design & Architecture

Tech Stack Selection

Layer	Technology	Rationale
Frontend	**React** (with Vite)	Component-based architecture, vast ecosystem, and high performance. Vite offers faster cold starts and HMR.
UI Framework	**Tailwind CSS**	Utility-first CSS framework for rapidly building custom, responsive designs without leaving the HTML.
State Management	**React Context API + useReducer** / **Zustand**	Lightweight solution sufficient for MVP-scale global state (auth, conversations). Zustand is a simple, scalable alternative to Redux.
Real-Time	**Socket.IO Client**	Established library for handling real-time, bidirectional communication. Integrates easily with a Node.js backend.
API Client	**Axios** / **Fetch API**	Robust HTTP client for making REST API calls to the backend (for non-real-time operations like search, login).
Backend (For UI Contract)	**Mock Service Worker (MSW)**	Allows mocking API endpoints directly in the browser, enabling full frontend development without a live backend.
Build & Deploy	**Vite** (Build), **Netlify** / **Vercel** (Hosting)	Vite is fast and modern. Netlify/Vercel offer seamless Git-based deployments, previews, and excellent performance for SPAs.

UI Structure / API Schema Design

UI Component Structure (Atomic Design Principles)

- Atoms: Button, Input, Avatar, Icon, Badge
- Molecules: MessageBubble, ConversationListItem, SearchBar
- Organisms: ConversationList, MessageList, MessageInput, Navbar
- Templates: MainLayout (Sidebar + Main View)
- Pages: LoginPage, ChatPage

API Schema Design (Frontend Perspective)

The UI will expect the following data structures from the API:

User Object:

```
{
"id": "string",
"username": "string",
"email": "string",
"avatarUrl": "string | null"
}
```

Conversation Object (for list item):

```
{
"id": "string",
"type": "direct" | "group",
"name": "string", // For groups, custom name. For direct, other user's name.
"avatarUrl": "string | null",
"lastMessage": {
  "text": "string",
  "timestamp": "ISO8601 string",
  "senderId": "string"
},
"unreadCount": "number",
"participants": "User[]" // Simplified for group avatars
}
```

Message Object:

```
 {
"id": "string",
"conversationId": "string",
"senderId": "string",
"text": "string",
"timestamp": "ISO8601 string",
"status": "sending" | "sent" | "delivered" | "read" // For UI state
 }
```

Data Handling Approach

✧ **Server State vs. UI State:**

- Server State: Conversations, Messages, User data. This will be fetched from and sent to the API.
- UI State: Current selected conversation, input field text, loading spinners, modal open/close states. This will be managed locally within components or via React Context.

✧ **State Management Strategy:**

- Auth Context: Global state for currentUser, authToken, and login/logout functions.
- Conversations Context: Global state for the list of conversations and the currently selectedConversation.
- Messages State: Fetched and stored locally within the ChatPage component based on the selectedConversation.id. This avoids storing a massive, ever-growing list of all messages globally.
- Optimistic Updates: For messages, the UI will immediately add a new message to the local state with a status: 'sending' before the API request is complete. Upon confirmation from the server, the message's status will be updated to 'sent'. This provides a seamless user experience.

✧ **Real-Time Data Flow:**

- The client connects to the server via Socket.IO upon successful login.
- The client "joins" rooms based on conversation IDs it is a part of.
- When a new message is received via the socket 'new_message' event, the UI checks if it belongs to the currently open conversation. If yes, it appends it to the message list. If not, it updates the lastMessage and unreadCount in the relevant conversation in the Conversations Context.

Component / Module Diagram

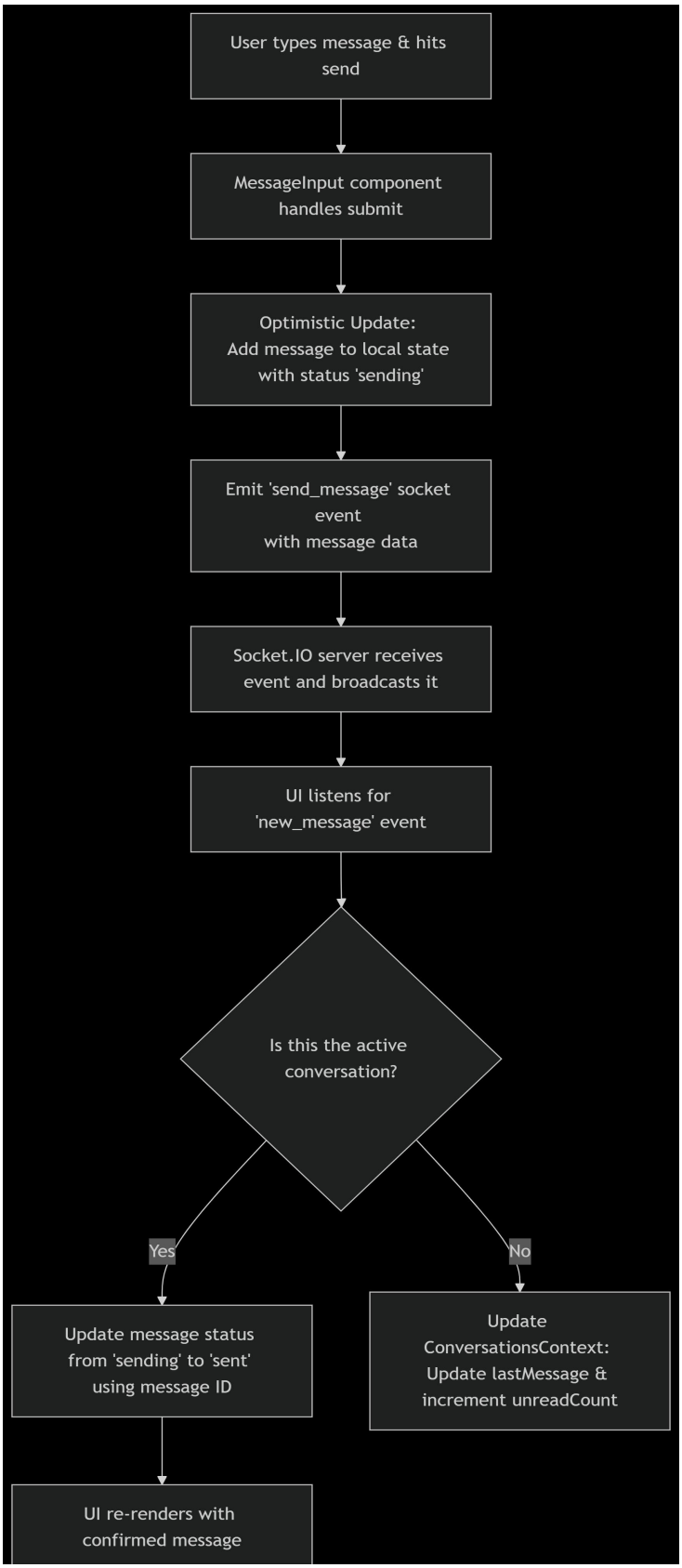
```
src/
|
├── components/
|   ├── atoms/
|   |   ├── Button/
|   |   ├── Input/
|   |   └── Avatar/
|   ├── molecules/
|   |   ├── MessageBubble/
|   |   └── ConversationListItem/
|   ├── organisms/
|   |   ├── ConversationList/
|   |   ├── MessageList/
|   |   └── MessageInput/
|   └── templates/
|       └── MainLayout/
|
├── contexts/
|   ├── AuthContext.jsx  // Manages authentication state
|   └── ConversationsContext.jsx // Manages list of conversations
|
├── hooks/
|   ├── useSocket.js     // Custom hook for socket connection
|   └── useApi.js        // Custom hook for API calls
|
├── pages/
|   ├── LoginPage/
|   └── ChatPage/        // Main app page (uses MainLayout)
|
└── services/
    ├── api.js          // Axios instance with base config
    └── socket.js       // Socket.IO client initialization
```

Key Data Flow:

1. ChatPage component uses the ConversationsContext to get and set the list of conversations.
2. ConversationList organism reads the list from the context and maps over it, rendering a ConversationListItem for each.
3. When an item is clicked, it calls setSelectedConversation from the context.
4. The ChatPage sees this change and uses the useEffect hook to fetch messages for the new selectedConversation.id.
5. MessageInput organism handles the input field. On submit, it first does an optimistic update to the local message state, then sends the message via the Socket.IO connection.

Basic Flow Diagram

Primary User Flow: Sending a Message



Secondary Flow: Receiving a Message (Real-Time)