Chat System Design

- Overview
- Architecture
  - Components
- Websocket Implementation
  - 1. Hub Structure
  - 2. Connection Flow
- Security Measures
  - 1. Authentication
  - 2. Authorization
  - 3. Rate Limiting
- Message Handling
  - 1. Message Types
  - 2. Message Flow
- Frontend Integration
  - 1. ChatWindow Component
  - 2. Message Component
- Error Handling
  - 1. Connection Errors
  - 2. Message Errors

Overview

The chat system enables real-time communication between event participants using WebSocket connections. Each event has its own chat room where participants can send and receive messages.

Architecture

Components

- **WebSocket Hub**: Central manager for all WebSocket connections and message broadcasting
- **Client**: Individual WebSocket connection for each user
- **Event Participant Validator**: Ensures only event participants can join the chat

Websocket Implementation

1. Hub Structure

```
type Hub struct {
    // Registered clients for each event
    clients map[int64]map[*Client]bool

    // Channel for broadcasting messages
    broadcast chan *Message

    // Register requests from clients
    register chan *Client

    // Unregister requests from clients
    unregister chan *Client
}
```

```
14
15 type Client struct {
16     // The websocket connection
17     conn *websocket.Conn
18
19     // Event ID this client is connected to
20     eventID int64
21
22     // User information
23     userID int64
24     userName string
25
26     // Buffered channel of outbound messages
27     send chan []byte
28 }
29
30 type Message struct {
31     Type      string    `json:"type"`
32     Content   string    `json:"content"`
33     Sender    string    `json:"sender"`
34     Timestamp time.Time `json:"timestamp"`
35     EventID   int64     `json:"event_id"`
36     UserID    int64     `json:"user_id"`
37 }
```

2. Connection Flow

1. **Connection Initialization**

```
1 Client -> Server: WebSocket connection request with token
2 Server: Validate token and check if user is event participant
3 Server: Create new client instance
4 Server: Register client with hub
5
```

2. **Message Broadcasting**

```
1 Client -> Hub: Send message
2 Hub: Validate message
3 Hub: Broadcast to all clients in the event
4
```

3. **Connection Termination**

```
1 Client -> Hub: Close connection
2 Hub: Unregister client
3 Hub: Clean up resources
```

Security Measures

1. Authentication

- JWT token required for WebSocket connection
- Token passed as query parameter
- Token validation on connection and message sending

2. Authorization

- Only event participants can join chat
- Messages are tied to specific events

3. Rate Limiting

- Maximum message size restrictions
- Basic connection limits per event

Message Handling

1. Message Types

```
{
    "CHAT": {
        "type": "message",
        "content": "Message content",
        "sender": "User name",
        "timestamp": "2024-03-25T10:00:00Z"
    },
    "SYSTEM": {
        "type": "system",
        "content": "User joined the chat",
        "timestamp": "2024-03-25T10:00:00Z"
    }
}

```

2. Message Flow

1. **Sending Messages**

```
Client: Creates message
Client -> Server: Sends WebSocket message
Server: Validates message format and sender
Server: Broadcasts to all clients in the event

```

2. **Receiving Messages**

```
Server -> Client: Broadcasts message
Client: Receives message
Client: Updates UI with new message

```

Frontend Integration

1. ChatWindow Component

```
const ChatWindow = ({ eventId, isParticipant }) => {
    const [messages, setMessages] = useState([]);
    const [socket, setSocket] = useState(null);
    const messagesEndRef = useRef(null);

    // WebSocket connection management
    useEffect(() => {
```

```
        if (!isParticipant) return;

        const token = localStorage.getItem('token');
        const ws = new WebSocket(`ws://localhost:8080/v1/events/${eventId}/chat?token=${token}`);

        ws.onmessage = (event) => {
            const message = JSON.parse(event.data);
            setMessages(prev => [...prev, message]);
        };

        setSocket(ws);
        return () => ws.close();
    }, [eventId, isParticipant]);

    // Auto-scroll to bottom on new messages
    useEffect(() => {
        messagesEndRef.current?.scrollIntoView({ behavior: 'smooth' });
    }, [messages]);

    // Message sending handler
    const sendMessage = (content) => {
        if (socket && socket.readyState === WebSocket.OPEN) {
            socket.send(JSON.stringify({
                type: 'message',
                content: content
            }));
        }
    };

    return (
        <div className="chat-window">
            <div className="messages-container">
                {messages.map((msg, index) => (
                    <Message key={index} message={msg} />
                ))}
                <div ref={messagesEndRef} />
            </div>
            <MessageInput onSend={sendMessage} />
        </div>
    );
};

```

2. Message Component

```
const Message = ({ message }) => {
    const isCurrentUser = message.user_id === parseInt(localStorage.getItem('userId'));

    return (
        <div className={`message ${isCurrentUser ? 'own-message' : ''}`}>
            <div className="message-header">
                <span className="sender">{message.sender}</span>
                <span className="timestamp">
                    {new Date(message.timestamp).toLocaleTimeString()}
                </span>
            </div>
            <div className="message-content">{message.content}</div>
        </div>
```

```
    );
};

```

Error Handling

1. Connection Errors

- Basic reconnection on connection loss
- User notification of connection status

2. Message Errors

- Invalid message format handling
- Connection state validation before sending