Support Chat System - Complete Documentation

🎯 Overview

This is a real-time support chat system built with Supabase Realtime, allowing users to communicate with admin support staff. The system ensures proper message isolation and prevents message leakage between different conversations.

🏗️ Architecture

Core Concepts

1. Room Structure

- Each **END USER** has exactly **ONE** room
- Room is identified by the end user's `user_id`
- Multiple admins can connect to the same user's room
- Users connect to their own room; admins connect to target users' rooms

2. Database Schema

```
sql

CREATE TABLE user_messages (
    id uuid PRIMARY KEY,
    user_id uuid NOT NULL,      -- ✅ ALWAYS the end user's ID (room owner)
    admin_id uuid,              -- ✅ The admin who sent (NULL for user messages)
    content text NOT NULL,
    sender_type message_sender_type NOT NULL, -- 'user' or 'admin'
    is_read boolean DEFAULT false,
    read_at timestamp,
    created_at timestamp,
    updated_at timestamp
);
```

Key Fields:

- `user_id`: ALWAYS the end user's ID (identifies the room)
- `admin_id`: Only populated when an admin sends a message

- `sender_type`: Either 'user' or 'admin'

🔄 Connection Flow

User Mode Flow

```
1. User logs in
   ├─ User ID: ABC
   └─ User role: regular user

2. Initialize Hook
   ├─ roomId = ABC (their own ID)
   ├─ adminId = null
   └─ mode = USER

3. Create Channel
   ├─ Channel name: "support_chat:ABC"
   └─ Postgres filter: user_id=eq.ABC

4. Send Message
   └─ Data: {
      user_id: ABC,          // Room owner (themselves)
      sender_type: 'user',
      admin_id: null         // Not an admin
    }

5. Receive Messages
   ├─ All messages where user_id = ABC
   ├─ Unread count: sender_type = 'admin' AND is_read = false
   └─ Notification: when sender_type = 'admin'
```

Admin Mode Flow

```
1. Admin logs in
   ├─ Admin ID: XYZ
   └─ Admin role: admin

2. Select User
   └─ Target User ID: ABC

3. Initialize Hook with targetUserId
```

```
   ├─ roomId = ABC (TARGET user's ID, NOT admin's)
   ├─ adminId = XYZ (admin's ID stored separately)
   └─ mode = ADMIN

4. Create Channel
   ├─ Channel name: "support_chat:ABC"
   └─ Postgres filter: user_id=eq.ABC

5. Send Message
   └─ Data: {
      user_id: ABC,          // Room owner (the end user)
      sender_type: 'admin',
      admin_id: XYZ          // The admin sending
     }

6. Receive Messages
   ├─ All messages where user_id = ABC
   ├─ Unread count: sender_type = 'user' AND is_read = false
   └─ Notification: when sender_type = 'user'
```

🔒 Security & Isolation

How Message Isolation Works

1. **Channel Naming**

javascript

```javascript
const channelName = `support_chat:${roomId}`;
// roomId is ALWAYS the end user's ID
// Example: "support_chat:e0054437-50c7-48b5-a388-dbd1da9f7c94"
```

2. **Postgres Change Filter**

javascript

```
postgres_changes: [{
  event: '*',
  schema: 'public',
  table: 'user_messages',
  filter: `user_id=eq.${roomId}`  // CRITICAL: Only this room's messages
}]
```

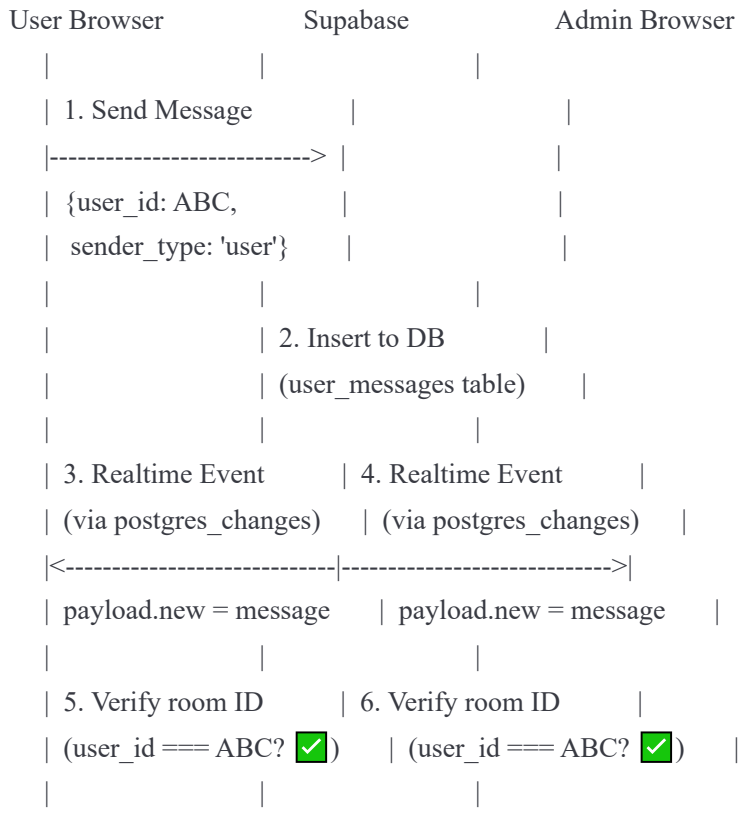
3. **Double Verification**

```
javascript

// When receiving a message via realtime
const messageUserId = payload.new?.user_id || payload.old?.user_id;
if (messageUserId !== roomId) {
  console.warn('🚨 SECURITY: Rejecting message from different room!');
  return; // REJECT the message
}
```

📊 Data Flow Diagrams

User Sends Message



```
| 7. Add to messages[]       | 8. Add to messages[]       |
| (own message, no notif)    | (increment unread, notify) |
|                            |                            |
```

Admin Sends Message

```
Admin Browser                Supabase                User Browser
    |                           |                          |
    | 1. Send Message           |                          |
    |-------------------------> |                          |
    | {user_id: ABC,            |                          |
    |  sender_type: 'admin',    |                          |
    |  admin_id: XYZ}           |                          |
    |                           |                          |
    |                           | 2. Insert to DB          |
    |                           | (user_messages table)    |
    |                           |                          |
    | 3. Realtime Event         | 4. Realtime Event        |
    |<--------------------------|------------------------->|
    | payload.new = message     | payload.new = message    |
    |                           |                          |
    | 5. Verify room ID         | 6. Verify room ID        |
    | (user_id === ABC? ✅)     | (user_id === ABC? ✅)    |
    |                           |                          |
    | 7. Add to messages[]      | 8. Add to messages[]     |
    | (own message, no notif)   | (increment unread, notify) |
    |                           |                          |
```

🔧 Implementation Details

Key Variables

javascript

```
// Hook State
const [roomId, setRoomId] = useState(null);          // The end user's ID
const currentUserIdRef = useRef(null);               // Logged-in user's ID
const adminIdRef = useRef(null);                     // Admin's ID (if admin mode)

// Determination Logic
if (adminMode && targetUserId) {
  roomId = targetUserId;          // Connect to TARGET user's room
  adminId = currentUser.id;       // Store admin's ID separately
} else {
  roomId = currentUser.id;        // Connect to own room
  adminId = null;                 // Not an admin
}
```

Message Sending

javascript

```
const messageData = {
  user_id: roomId,                // ✅ ALWAYS the end user's ID
  content: content.trim(),
  sender_type: adminMode ? 'admin' : 'user',
  is_read: false
};

// Only set admin_id when an admin is sending
if (adminMode && adminIdRef.current) {
  messageData.admin_id = adminIdRef.current;
}
```

Unread Count Logic

javascript

```
// For Users: Count unread messages from admins
const targetSenderType = 'admin';

// For Admins: Count unread messages from users
const targetSenderType = 'user';

// Query
const unread = messages.filter(msg =>
  msg.sender_type === targetSenderType && !msg.is_read
).length;
```

🚨 Common Issues & Solutions

Issue 1: Messages Leaking Between Rooms

Symptom: Admin sees messages from multiple users in one conversation

Cause: Incorrect room ID or missing verification

Solution:

```
javascript

// ALWAYS verify message belongs to current room
const messageUserId = payload.new?.user_id;
if (messageUserId !== roomId) {
  return; // Reject the message
}
```

Issue 2: Admin Messages Have Wrong user_id

Symptom: Admin messages are associated with admin's ID instead of target user

Cause: Using currentUser.id instead of roomId

Solution:

```
javascript
```

```
// ❌ WRONG
user_id: currentUser.id

// ✅ CORRECT
user_id: roomId  // Always the end user's ID
```

Issue 3: Multiple Subscriptions to Same Room

Symptom: Duplicate messages or connection conflicts

Cause: Not cleaning up previous subscriptions

Solution:

```
javascript

// Always cleanup before creating new subscription
cleanupChannel();

// Use connection IDs to track active subscription
connectionIdRef.current += 1;
const currentConnectionId = connectionIdRef.current;

// Verify connection ID in callbacks
if (connectionIdRef.current !== currentConnectionId) {
  return; // Ignore stale events
}
```

📋 Testing Checklist

User-to-Admin Communication

- ☐ User can send message
- ☐ Admin receives message in correct room
- ☐ Admin sees unread count increase
- ☐ Admin can reply
- ☐ User receives admin's reply
- ☐ User sees unread count increase
- ☐ Messages are marked as read when viewed

Room Isolation

- [ ] Admin opens User A's chat → sees only User A's messages
- [ ] Admin opens User B's chat → sees only User B's messages
- [ ] User A doesn't see User B's messages
- [ ] Messages sent to User A don't appear in User B's chat

Multiple Admins

- [ ] Admin 1 and Admin 2 can both connect to User A's room
- [ ] Both admins see the same conversation
- [ ] Both admins see when the other sends a message
- [ ] admin_id field correctly identifies which admin sent each message

Connection Reliability

- [ ] Reconnection works after network drop
- [ ] No duplicate messages on reconnect
- [ ] Heartbeat prevents stale connections
- [ ] Channel cleanup prevents memory leaks

🔍 Debugging

Enable Verbose Logging

All key operations are logged with prefixes:

- [Init] - Initialization
- [Subscription] - Channel subscription
- [Realtime] - Realtime events
- [Send] - Message sending
- [Security] - Security warnings
- [Cleanup] - Resource cleanup

Check Room Assignment

```
javascript
```

```
console.log({
  loggedInUserId: currentUserIdRef.current,
  roomId: roomId,
  adminId: adminIdRef.current,
  mode: adminMode ? 'ADMIN' : 'USER'
});
```

Verify Message Structure

```
javascript

console.log('Message data:', {
  user_id: messageData.user_id,      // Should be end user's ID
  admin_id: messageData.admin_id,    // Should be admin's ID or null
  sender_type: messageData.sender_type
});
```

🎓 Best Practices

1. **Always Use roomId for user_id**
   - Never use currentUser.id directly
   - roomId is calculated based on mode and target
2. **Verify Message Ownership**
   - Always check messageUserId === roomId
   - Reject messages from other rooms
3. **Clean Up Subscriptions**
   - Call cleanup before creating new subscription
   - Use connection IDs to prevent race conditions
4. **Separate Admin ID from User ID**
   - currentUserIdRef: Who is logged in
   - adminIdRef: Admin attribution for messages
   - roomId: Which conversation/room
5. **Handle Edge Cases**
   - Component unmounting
   - Network failures

- Token refresh
- User logout

Related Files

- `useSupportChat.jsx` - Main hook (this file)
- `SupportPortal.jsx` - User-facing chat UI
- `AdminSupportPortal.jsx` - Admin chat interface
- `user_messages` table - Database schema

Support

If messages are leaking between rooms:

1. Check roomId calculation in initialization
2. Verify postgres_changes filter matches roomId
3. Ensure message verification is working
4. Check for multiple active subscriptions

For connection issues:

1. Check Supabase realtime status
2. Verify heartbeat is running
3. Check connection ID tracking
4. Review cleanup logic