**Complete Messaging System Implementation Guide**

**Overview**

This guide explains how to implement a real-time messaging system step-by-step. We'll cover everything from user registration to sending messages.

**Database Architecture**

**Required Tables**

You need these 3 main tables:

**1. Users Table** (Assuming you already have this)

sql

CREATE TABLE users (

user\_id VARCHAR(36) PRIMARY KEY,

username VARCHAR(50) UNIQUE,

email VARCHAR(100) UNIQUE,

password\_hash VARCHAR(255),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

**2. Conversations Table**

sql

CREATE TABLE conversations (

conversation\_id VARCHAR(36) PRIMARY KEY,

user1\_id VARCHAR(36) NOT NULL,

user2\_id VARCHAR(36) NOT NULL,

last\_message TEXT,

last\_message\_time TIMESTAMP,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP

);

**3. Messages Table**

sql

CREATE TABLE messages (

message\_id VARCHAR(36) PRIMARY KEY,

conversation\_id VARCHAR(36) NOT NULL,

sender\_id VARCHAR(36) NOT NULL,

content TEXT NOT NULL,

is\_read BOOLEAN DEFAULT FALSE,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

Complete User Flow Implementation

Phase 1: User Registration & Authentication

**Step 1: User Signs Up**

User provides: username, email, password

System creates account in users table

User receives confirmation

**Step 2: User Logs In**

**Endpoint:** POST /api/auth/login

User provides: email/username + password

System validates credentials

Returns JWT token for authentication

Token contains user\_id and other user info

Phase 2: Messaging System Initialization

**Step 3: Connect to Socket Server**

User's app connects to Socket.IO server

**Socket Event:** connect

System establishes WebSocket connection

**Step 4: Register User in Socket System**

**Socket Event:** register

Send: { user\_id, namefirst }

System maps socket ID to user ID

System automatically joins user to all their existing conversation rooms

**Phase 3: Conversation Management**

**Step 5: Load User's Conversations List**

**Endpoint:** GET /api/messages/conversations

**Authentication:** Required (JWT token)

System fetches all conversations where user is participant

Returns list of conversations with last message details

**Step 6: Start New Conversation (Optional)**

**Endpoint:** POST /api/messages/conversation

**Authentication:** Required

Send: { user1\_id, user2\_id }

System checks if conversation already exists between these users

If exists: returns existing conversation\_id

If not: creates new conversation and returns conversation\_id

**Phase 4: Active Messaging**

**Step 7: Join Specific Conversation**

**Socket Event:** join\_conversation

Send: conversationId

User joins the conversation room for real-time updates

System adds user to conversation room

**Step 8: Load Conversation Messages**

**Endpoint:** GET /api/messages/{conversationId}

**Authentication:** Required

System fetches all messages for this conversation

Returns messages ordered by timestamp (oldest first)

User can now see chat history

**Step 9: Send Message**

**Socket Event:** send\_message

Send: { conversationId, senderId, content }

System saves message to database

System broadcasts message to all users in conversation room

Other users receive message instantly via new\_message event

**Step 10: Receive Messages**

**Socket Event:** new\_message (received automatically)

Contains: { conversation\_id, message\_id, sender\_id, content, created\_at, is\_read }

User's app displays new message in chat interface

Message appears in real-time without page refresh

**Step 11: Mark Messages as Read**

**Endpoint:** PUT /api/messages/read/{conversationId}

**Authentication:** Required

System marks all unread messages from other user as read

System emits messages\_read socket event to conversation room

Other user sees "read" status on their messages

Phase 5: Conversation Management

**Step 12: Leave Conversation**

**Socket Event:** leave\_conversation

Send: conversationId

User leaves conversation room

Stops receiving real-time updates for this conversation

**Step 13: Handle Disconnection**

**Socket Event:** disconnect (automatic)

System removes user from all conversation rooms

Cleans up socket-to-user mapping

User can reconnect and repeat from Step 3

Key Endpoints Summary

Authentication Endpoints

POST /api/auth/login - User login

POST http://localhost:4000/api/user/signup

Messaging API Endpoints

GET /api/messages/conversations - Get user's conversations

POST /api/messages/conversation - Create/find conversation

GET /api/messages/{conversationId} - Get conversation messages

PUT /api/messages/read/{conversationId} - Mark messages as read

Socket Events

register - Register user in socket system

join\_conversation - Join conversation room

send\_message - Send new message

new\_message - Receive new message (auto)

messages\_read - Message read notification (auto)

leave\_conversation - Leave conversation room

Security Features

Database Security

All queries use mysql2.escape() to prevent SQL injection

UUID-based IDs for conversations and messages

Foreign key constraints maintain data integrity

API Security

All messaging endpoints require JWT authentication

Users can only access their own conversations

Conversation access validation before allowing operations

Socket Security

User registration required before any messaging operations

Conversation room access controlled by user permissions

Socket connections mapped to authenticated users only

File Structure

├── classes/

│ ├── messageCore.js # Core messaging business logic

│ └── operator.js # Database operations wrapper

├── controllers/

│ └── message.js # REST API endpoints

├── middleware/

│ └── auth.js # JWT authentication middleware

└── index.js # Main server + Socket.IO setup

**Start with Database:** Create tables first, test connections

**Build APIs:** Implement REST endpoints, test with Postman

**Add WebSocket:** Implement socket events one by one

**Test Everything:** Use socket.io client to test real-time features

**Security First:** Always validate user permissions before operations

Common Issues to Watch For

**Memory Leaks:** Clean up socket connections properly

**Race Conditions:** Handle concurrent message sending

**Authentication:** Validate JWT tokens on every request

**Data Validation:** Check all inputs before database operations

**Error Handling:** Return proper error messages to clients

# Messaging: Send and Receive Documentation

This document outlines how to send messages using the provided backend API and how to receive messages and read notifications in real-time using WebSocket events.

## 1. Sending Messages

Messages can be sent using the /api/messages/send REST API endpoint.

### Endpoint:

POST /api/messages/send

### Authentication:

This endpoint requires authentication. Include a valid JWT token in the Authorization header with the Bearer scheme.

Authorization: Bearer YOUR\_AUTH\_TOKEN

### Request Body:

The request body should be a JSON object containing the conversation ID and the message content.

{

"conversationId": "[UUID of the target conversation]",

"content": "[The message text content]"

}

* conversationId: (Required, string) The unique identifier of the conversation the message belongs to.
* content: (Required, string) The actual text content of the message.

### Success Response (200 OK):

{

"success": true,

"message": "Message sent successfully",

"messageId": "[UUID of the newly created message]"

}

### Error Responses:

* 400 Bad Request: Invalid input (e.g., missing conversationId or content).
* {
* "success": false,
* "error": "conversationId and content are required."
* }
* 401 Unauthorized: Missing or invalid authentication token.
* {
* "success": false,
* "error": "MSG-AUTHENTICATION-FAILED"
* // (or other authentication/authorization errors)
* }
* 403 Forbidden: The authenticated user is not a participant in the specified conversation.
* {
* "success": false,
* "error": "You are not part of this conversation."
* }
* 500 Internal Server Error: An error occurred on the server side (e.g., database error, issue saving message).
* {
* "success": false,
* "error": "Internal server error."
* // (or a more specific error message from MessageCore.saveMessage)
* }

### Backend Processing:

Upon receiving a valid request, the backend:

1. Verifies the authenticated user is part of the conversation.
2. Saves the message to the database.
3. Emits a new\_message WebSocket event to all connected clients who are joined to the conversation room.

## 2. Receiving Messages and Read Notifications (via WebSocket)

Messages and read status updates are delivered in real-time via WebSocket events using Socket.IO.

### Connecting to WebSocket:

Establish a Socket.IO connection to the backend server URL.

const socket = io('http://your-backend-url:port');

### User Registration (after connection):

After connecting, register the user with the backend to associate the socket with a user ID and automatically join their existing conversation rooms.

socket.on('connect', () => {

socket.emit('register', {

user\_id: '[Authenticated User ID]',

namefirst: '[Authenticated User First Name]'

});

});

### Joining Specific Conversation Rooms:

To receive messages for a particular conversation, the client must join that conversation's room. This is typically done when the user opens or focuses on a conversation in the UI.

socket.emit('join\_conversation', '[UUID of the conversation to join]');

### Listening for New Messages:

Listen for the new\_message event to receive real-time message updates.

socket.on('new\_message', (message) => {

console.log('New message received:', message);

// Process and display the message in the UI

});

#### new\_message Event Data Structure:

The message object received in the event typically contains:

{

"conversation\_id": "[UUID of the conversation]",

"message\_id": "[UUID of the new message]",

"sender\_id": "[UUID of the sender]",

"content": "[The message text content]",

"created\_at": "[Timestamp of message creation (server time)]",

"is\_read": false // New messages are initially marked as unread

}

### Listening for Messages Read Notifications:

Listen for the messages\_read event to know when another user in a conversation has marked messages as read.

socket.on('messages\_read', (data) => {

console.log('Messages marked as read:', data);

// Update the UI to reflect that messages have been read

});

#### messages\_read Event Data Structure:

The data object received in the event typically contains:

{

"conversationId": "[UUID of the conversation]",

"userId": "[UUID of the user who marked messages as read]"

}