Flutter WebSocket Integration Guide

Complete Implementation Guide for HPlus Medical System

🏥 Real-time Chat System

🚨 Emergency Dispatch & Tracking

📍 Location Sharing & GPS Integration

🏥 Multi-Hospital Role-Based Access

Version 2.0 • December 2024

Interactive PDF Guide

Table of Contents

1 Overview & System Architecture Page 3

2 Prerequisites & Setup Page 5

3 WebSocket Connection Setup Page 7

4 Authentication & User Roles Page 12

5 Chat System Implementation Page 15

6 Emergency System Implementation Page 25

7 Real-time Location Sharing Page 35

8 Message Types & Data Structures Page 40

9 Complete Flutter Implementation Examples Page 45

10 Error Handling & Best Practices Page 60

11 Testing & Debugging Page 65

12 Appendix & Quick Reference Page 70

1. 🏗️ Overview & System Architecture

💡 **What you'll learn:** Understanding the HPlus WebSocket system architecture, key features, and how Flutter integrates with the backend for real-time medical communication.

System Architecture

The HPlus backend provides a comprehensive WebSocket system designed specifically for medical facilities, enabling real-time communication between hospitals, paramedics, and patients. The system is built with role-based isolation and multi-hospital support.

Multi-Hospital Support

Each hospital operates independently with isolated communication channels while maintaining system-wide emergency coordination.

Role-Based Access Control

Secure isolation between doctors, nurses, paramedics, patients, and administrative staff with granular permissions.

Real-time Messaging

Instant chat with delivery status, typing indicators, message reactions, and file attachments.

Emergency Dispatch

Automated emergency request routing with location tracking and real-time ambulance monitoring.

Key Features

✅ **Core Capabilities:**

- **Secure role-based access control** - Each user type has specific permissions
- **Multi-hospital support** - Independent hospital operations with shared emergency services
- **Real-time messaging with delivery status** - Know when messages are sent, delivered, and read
- **Emergency request and response system** - Automated dispatch and tracking
- **GPS location tracking** - Real-time location sharing for emergency services
- **Call management** - Video/voice call initiation, acceptance, and management
- **Automatic reconnection handling** - Robust connection management with offline support

User Roles and Permissions

Role	Access Level	Primary Functions	Hospital ID Required
`doctor`	Hospital Staff	Patient consultation, emergency response, chat with staff/patients	Yes
`nurse`	Hospital Staff	Patient care coordination, chat with staff, emergency assistance	Yes
`paramedic`	Emergency Services	Emergency response, location tracking, ambulance dispatch	Yes
`patient`	External User	Emergency requests, chat with assigned medical staff	No
`admin`	Hospital Admin	System management, user oversight, full hospital access	Yes
`receptionist`	Hospital Staff	Patient coordination, appointment management, staff communication	Yes

Message Flow Architecture

Copy

```
Client (Flutter App)
    ↓ WebSocket Connection with Authentication
Server (Node.js + WebSocket)
    ↓ Message Routing by Type
┌─────────────────┬─────────────────┬─────────────────┐
│   Chat System   │ Emergency System│ Location System │
│   - Messaging   │   - Dispatch    │   - GPS Track   │
│   - Typing      │   - Response    │   - Sharing     │
│   - Read Status │   - Tracking    │   - Updates     │
└─────────────────┴─────────────────┴─────────────────┘
    ↓ Role-based Message Distribution
Hospital Staff ← → Patients ← → Paramedics
```

2. 📋 Prerequisites & Setup

⚠️ **Important:** Ensure you have the latest Flutter SDK (3.0+) and the required permissions set up in your app before proceeding with WebSocket integration.

Flutter Dependencies

Add these dependencies to your `pubspec.yaml` file:

Copy

```
dependencies:
  flutter:
    sdk: flutter

  # WebSocket and HTTP communication
  web_socket_channel: ^2.4.0
  dio: ^5.3.2

  # Location and permissions
  geolocator: ^9.0.2
  permission_handler: ^11.0.1

  # State management and storage
  provider: ^6.1.1
  shared_preferences: ^2.2.2

  # Utilities
  uuid: ^4.1.0
  path_provider: ^2.1.1

  # Optional: For enhanced UI
  flutter_local_notifications: ^16.1.0
  connectivity_plus: ^5.0.1

dev_dependencies:
  flutter_test:
    sdk: flutter
  flutter_lints: ^3.0.0
```

Android Permissions

Add these permissions to `android/app/src/main/AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.VIBRATE" />
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />

<!-- For file uploads in chat -->
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.CAMERA" />
```

Copy

iOS Permissions

Add these to `ios/Runner/Info.plist`:

```
<key>NSLocationWhenInUseUsageDescription</key>
<string>This app needs location access for emergency services and ambulance tracking.</string>

<key>NSLocationAlwaysAndWhenInUseUsageDescription</key>
<string>This app needs background location access for emergency tracking.</string>

<key>NSCameraUsageDescription</key>
<string>This app needs camera access to share images in medical consultations.</string>

<key>NSMicrophoneUsageDescription</key>
<string>This app needs microphone access for voice messages and calls.</string>
```

Copy

Backend Server Configuration

1

Server Connection Details

Configure your WebSocket connection parameters:

- **Development:** `ws://localhost:5000`
- **Production:** `wss://your-domain.com:5000`
- **Protocol:** WebSocket with JSON messages

2

Environment Configuration

Create a configuration file for different environments:

Copy

```
// lib/config/app_config.dart
class AppConfig {
  static const String devWebSocketUrl = 'ws://localhost:5000';
  static const String prodWebSocketUrl = 'wss://api.hplus.com:5000';

  static String get webSocketUrl {
    return const bool.fromEnvironment('dart.vm.product')
        ? prodWebSocketUrl
        : devWebSocketUrl;
  }

  static const Duration heartbeatInterval = Duration(seconds: 30);
  static const Duration reconnectDelay = Duration(seconds: 5);
  static const int maxReconnectAttempts = 5;
}
```

Project Structure

Organize your Flutter project with this recommended structure:

Copy

```
lib/
├── config/
│   └── app_config.dart
├── models/
│   ├── user_model.dart
│   ├── message_model.dart
│   ├── emergency_model.dart
│   └── location_model.dart
├── services/
│   ├── websocket_service.dart
│   ├── chat_service.dart
│   ├── emergency_service.dart
│   ├── location_service.dart
│   └── auth_service.dart
├── providers/
│   ├── websocket_provider.dart
│   ├── chat_provider.dart
│   ├── emergency_provider.dart
│   └── location_provider.dart
├── screens/
│   ├── chat/
│   ├── emergency/
│   ├── dashboard/
│   └── auth/
├── widgets/
│   ├── chat/
│   ├── emergency/
│   └── common/
└── utils/
    ├── constants.dart
    ├── helpers.dart
    └── validators.dart
```

3. 🔌 WebSocket Connection Setup

> 💡 **Connection Parameters:** The WebSocket connection requires specific query parameters for authentication and role-based access. Each user type has different requirements.

Connection URL Format

The WebSocket connection URL follows this format:

```
ws://server-url:port?userId={userId}&role={role}&hospitalId={hospitalId}&department={department}&specialization={specialization}
```

Copy

Required Parameters

Parameter	Type	Required For	Description
`userId`	String	All users	Unique identifier for the user
`role`	String	All users	User role (doctor, nurse, patient, etc.)
`hospitalId`	String	Hospital staff, Paramedics	Hospital identifier for staff isolation
`department`	String	Optional	Department (emergency, cardiology, etc.)
`specialization`	String	Optional	Medical specialization

Flutter WebSocket Service Implementation

1

Basic WebSocket Service

Create a comprehensive WebSocket service:

Copy

```dart
// lib/services/websocket_service.dart
import 'dart:convert';
import 'dart:async';
import 'dart:io';
import 'package:web_socket_channel/web_socket_channel.dart';
import 'package:web_socket_channel/status.dart' as status;
import '../config/app_config.dart';

class WebSocketService {
  static final WebSocketService _instance = WebSocketService._internal();
  factory WebSocketService() => _instance;
  WebSocketService._internal();

  WebSocketChannel? _channel;
  bool _isConnected = false;
  bool _isReconnecting = false;
  int _reconnectAttempts = 0;
  Timer? _heartbeatTimer;
  Timer? _reconnectTimer;

  // Connection parameters
  String? _userId;
  String? _role;
  String? _hospitalId;
  String? _department;
  String? _specialization;

  // Stream controllers for different message types
  final _messageController = StreamController<Map<String, dynamic>>.broadcast();
  final _connectionController = StreamController<bool>.broadcast();
  final _errorController = StreamController<String>.broadcast();

  // Getters
  bool get isConnected => _isConnected;
  bool get isReconnecting => _isReconnecting;
  String? get userId => _userId;
  String? get role => _role;
  String? get hospitalId => _hospitalId;

  // Streams
  Stream<Map<String, dynamic>> get messageStream => _messageController.stream;
  Stream<bool> get connectionStream => _connectionController.stream;
  Stream<String> get errorStream => _errorController.stream;

  /// Connect to WebSocket server
  Future<bool> connect({
    required String userId,
    required String role,
    String? hospitalId,
    String? department,
    String? specialization,
  }) async {
    if (_isConnected) {
      print('Already connected to WebSocket');
      return true;
    }

    _userId = userId;
    _role = role;
    _hospitalId = hospitalId;
    _department = department;
```

```
    _specialization = specialization;

    try {
      final uri = _buildConnectionUri();
      print('Connecting to WebSocket: $uri');

      _channel = WebSocketChannel.connect(uri);

      // Wait for connection establishment
      await _channel!.ready;

      _isConnected = true;
      _reconnectAttempts = 0;
      _connectionController.add(true);

      // Setup message listener
      _setupMessageListener();

      // Start heartbeat
      _startHeartbeat();

      print('WebSocket connected successfully');
      return true;

    } catch (e) {
      print('WebSocket connection failed: $e');
      _errorController.add('Connection failed: $e');
      _scheduleReconnect();
      return false;
    }
  }

  /// Build connection URI with parameters
  Uri _buildConnectionUri() {
    final baseUri = Uri.parse(AppConfig.webSocketUrl);
    final queryParams = <String, String>{
      'userId': _userId!,
      'role': _role!,
    };

    if (_hospitalId ≠ null) queryParams['hospitalId'] = _hospitalId!;
    if (_department ≠ null) queryParams['department'] = _department!;
    if (_specialization ≠ null) queryParams['specialization'] = _specialization!;

    return baseUri.replace(queryParameters: queryParams);
  }

  /// Setup message listener
  void _setupMessageListener() {
    _channel!.stream.listen(
      (data) {
        try {
          final message = json.decode(data) as Map<String, dynamic>;
          print('Received message: $message');
          _messageController.add(message);
        } catch (e) {
          print('Error parsing message: $e');
          _errorController.add('Message parsing error: $e');
        }
      },
      onError: (error) {
        print('WebSocket error: $error');
```

```
        _errorController.add('WebSocket error: $error');
        _handleDisconnection();
      },
      onDone: () {
        print('WebSocket connection closed');
        _handleDisconnection();
      },
    );
  }

  /// Handle connection loss
  void _handleDisconnection() {
    _isConnected = false;
    _connectionController.add(false);
    _stopHeartbeat();

    if (!_isReconnecting) {
      _scheduleReconnect();
    }
  }

  /// Schedule automatic reconnection
  void _scheduleReconnect() {
    if (_reconnectAttempts >= AppConfig.maxReconnectAttempts) {
      print('Max reconnection attempts reached');
      _errorController.add('Unable to reconnect after ${AppConfig.maxReconnectAttempts}
attempts');
      return;
    }

    _isReconnecting = true;
    _reconnectAttempts++;

    print('Scheduling reconnection attempt $_reconnectAttempts in
${AppConfig.reconnectDelay.inSeconds}s');

    _reconnectTimer?.cancel();
    _reconnectTimer = Timer(AppConfig.reconnectDelay, () {
      _attemptReconnection();
    });
  }

  /// Attempt to reconnect
  Future<void> _attemptReconnection() async {
    if (_isConnected) return;

    print('Attempting reconnection...');

    final success = await connect(
      userId: _userId!,
      role: _role!,
      hospitalId: _hospitalId,
      department: _department,
      specialization: _specialization,
    );

    if (success) {
      _isReconnecting = false;
      print('Reconnection successful');
    } else {
      _scheduleReconnect();
    }
```

```
  }

  /// Start heartbeat to keep connection alive
  void _startHeartbeat() {
    _heartbeatTimer?.cancel();
    _heartbeatTimer = Timer.periodic(AppConfig.heartbeatInterval, (_) {
      if (_isConnected) {
        sendMessage({
          'type': 'heartbeat',
          'timestamp': DateTime.now().toIso8601String(),
        });
      }
    });
  }

  /// Stop heartbeat timer
  void _stopHeartbeat() {
    _heartbeatTimer?.cancel();
    _heartbeatTimer = null;
  }

  /// Send message through WebSocket
  bool sendMessage(Map<String, dynamic> message) {
    if (!_isConnected || _channel == null) {
      print('Cannot send message: WebSocket not connected');
      return false;
    }

    try {
      final messageString = json.encode(message);
      _channel!.sink.add(messageString);
      print('Message sent: $message');
      return true;
    } catch (e) {
      print('Error sending message: $e');
      _errorController.add('Send error: $e');
      return false;
    }
  }

  /// Disconnect from WebSocket
  Future<void> disconnect() async {
    print('Disconnecting WebSocket...');

    _isReconnecting = false;
    _reconnectTimer?.cancel();
    _stopHeartbeat();

    if (_channel != null) {
      await _channel!.sink.close(status.goingAway);
      _channel = null;
    }

    _isConnected = false;
    _connectionController.add(false);
    print('WebSocket disconnected');
  }

  /// Dispose resources
  void dispose() {
    disconnect();
    _messageController.close();
```

```
    _connectionController.close();
    _errorController.close();
  }
}
```

2

Connection State Provider

Create a provider to manage WebSocket state across your app:

Copy

```
// lib/providers/websocket_provider.dart
import 'package:flutter/foundation.dart';
import 'package:shared_preferences/shared_preferences.dart';
import '../services/websocket_service.dart';

class WebSocketProvider extends ChangeNotifier {
  final WebSocketService _webSocketService = WebSocketService();

  bool _isConnected = false;
  bool _isConnecting = false;
  String? _error;
  String? _currentUserId;
  String? _currentRole;

  // Getters
  bool get isConnected => _isConnected;
  bool get isConnecting => _isConnecting;
  String? get error => _error;
  WebSocketService get webSocketService => _webSocketService;

  WebSocketProvider() {
    _initializeProvider();
  }

  void _initializeProvider() {
    // Listen to connection status
    _webSocketService.connectionStream.listen((connected) {
      _isConnected = connected;
      _isConnecting = false;
      notifyListeners();
    });

    // Listen to errors
    _webSocketService.errorStream.listen((error) {
      _error = error;
      _isConnecting = false;
      notifyListeners();
    });
  }

  /// Connect with user credentials
  Future<bool> connectUser({
    required String userId,
    required String role,
    String? hospitalId,
    String? department,
    String? specialization,
  }) async {
    if (_isConnected) return true;

    _isConnecting = true;
    _error = null;
    notifyListeners();

    try {
      final success = await _webSocketService.connect(
        userId: userId,
        role: role,
        hospitalId: hospitalId,
        department: department,
        specialization: specialization,
```

```
      );

      if (success) {
        _currentUserId = userId;
        _currentRole = role;
        await _saveConnectionInfo(userId, role, hospitalId);
      }

      return success;
    } catch (e) {
      _error = e.toString();
      _isConnecting = false;
      notifyListeners();
      return false;
    }
  }

  /// Auto-connect from saved credentials
  Future<bool> autoConnect() async {
    final prefs = await SharedPreferences.getInstance();
    final userId = prefs.getString('websocket_user_id');
    final role = prefs.getString('websocket_role');
    final hospitalId = prefs.getString('websocket_hospital_id');

    if (userId != null && role != null) {
      return await connectUser(
        userId: userId,
        role: role,
        hospitalId: hospitalId,
      );
    }

    return false;
  }

  /// Save connection info for auto-connect
  Future<void> _saveConnectionInfo(String userId, String role, String? hospitalId) async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.setString('websocket_user_id', userId);
    await prefs.setString('websocket_role', role);
    if (hospitalId != null) {
      await prefs.setString('websocket_hospital_id', hospitalId);
    }
  }

  /// Disconnect and clear saved info
  Future<void> disconnect() async {
    await _webSocketService.disconnect();
    final prefs = await SharedPreferences.getInstance();
    await prefs.remove('websocket_user_id');
    await prefs.remove('websocket_role');
    await prefs.remove('websocket_hospital_id');

    _currentUserId = null;
    _currentRole = null;
    _error = null;
    notifyListeners();
  }

  /// Clear error
  void clearError() {
    _error = null;
```

```
    notifyListeners();
  }

  @override
  void dispose() {
    _webSocketService.dispose();
    super.dispose();
  }
}
```

3

Connection Testing Widget

Create a widget to test and monitor the WebSocket connection:

Copy

```
// lib/widgets/connection_status_widget.dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import '../providers/websocket_provider.dart';

class ConnectionStatusWidget extends StatelessWidget {
  const ConnectionStatusWidget({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Consumer<WebSocketProvider>(
      builder: (context, provider, child) {
        return Container(
          padding: const EdgeInsets.symmetric(horizontal: 16, vertical: 8),
          decoration: BoxDecoration(
            color: _getStatusColor(provider),
            borderRadius: BorderRadius.circular(20),
          ),
          child: Row(
            mainAxisSize: MainAxisSize.min,
            children: [
              Icon(
                _getStatusIcon(provider),
                color: Colors.white,
                size: 16,
              ),
              const SizedBox(width: 8),
              Text(
                _getStatusText(provider),
                style: const TextStyle(
                  color: Colors.white,
                  fontWeight: FontWeight.w500,
                  fontSize: 12,
                ),
              ),
              if (provider.isConnecting) ...[
                const SizedBox(width: 8),
                const SizedBox(
                  width: 12,
                  height: 12,
                  child: CircularProgressIndicator(
                    strokeWidth: 2,
                    valueColor: AlwaysStoppedAnimation<Color>(Colors.white),
                  ),
                ),
              ],
            ],
          ),
        );
      },
    );
  }

  Color _getStatusColor(WebSocketProvider provider) {
    if (provider.isConnecting) return Colors.orange;
    if (provider.isConnected) return Colors.green;
    return Colors.red;
  }

  IconData _getStatusIcon(WebSocketProvider provider) {
    if (provider.isConnecting) return Icons.sync;
```

```
    if (provider.isConnected) return Icons.wifi;
    return Icons.wifi_off;
  }

  String _getStatusText(WebSocketProvider provider) {
    if (provider.isConnecting) return 'Connecting...';
    if (provider.isConnected) return 'Connected';
    return 'Disconnected';
  }
}
```

Connection Error Handling

⚠️ **Common Connection Issues:**

- **Authentication errors:** Invalid userId or role parameters
- **Network issues:** Poor internet connection or server downtime
- **Permission errors:** Hospital staff missing hospitalId parameter
- **Server overload:** Too many concurrent connections

Copy

```
// Example error handling in your app
void _handleWebSocketError(String error) {
  print('WebSocket Error: $error');

  if (error.contains('Authentication required')) {
    // Redirect to login
    _showAuthenticationDialog();
  } else if (error.contains('Hospital ID is required')) {
    // Show hospital selection
    _showHospitalSelectionDialog();
  } else if (error.contains('Unable to reconnect')) {
    // Show manual reconnect option
    _showReconnectDialog();
  }
}
```

4. 🔐 Authentication & User Roles

User Role Types

> 📝 **HPlus System Roles:**
>
> - **patient:** Individual seeking medical care
> - **doctor:** Medical professional providing care
> - **nurse:** Healthcare support staff
> - **admin:** Hospital administrator
> - **emergency:** Emergency response personnel

Authentication Implementation

Copy

```
class AuthService {
  static const String _baseUrl = 'https://your-api-server.com';

  static Future<Map<String, dynamic>> authenticateUser(
    String email,
    String password
  ) async {
    try {
      final response = await http.post(
        Uri.parse('$_baseUrl/auth/login'),
        headers: {'Content-Type': 'application/json'},
        body: json.encode({
          'email': email,
          'password': password,
        }),
      );

      if (response.statusCode == 200) {
        final data = json.decode(response.body);

        // Store user credentials
        await _storeUserCredentials(data);

        return {
          'success': true,
          'user': data['user'],
          'token': data['token'],
        };
      } else {
        return {
          'success': false,
          'error': 'Invalid credentials',
        };
      }
    } catch (e) {
      return {
        'success': false,
        'error': 'Network error: $e',
      };
    }
  }

  static Future<void> _storeUserCredentials(Map<String, dynamic> data) async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.setString('user_id', data['user']['id']);
    await prefs.setString('user_role', data['user']['role']);
    await prefs.setString('auth_token', data['token']);

    // Store hospital ID for hospital staff
    if (data['user']['hospitalId'] != null) {
      await prefs.setString('hospital_id', data['user']['hospitalId']);
    }
  }
}
```

Role-Based Connection Parameters

Copy

```
class WebSocketService {
  Future<void> connectWithUserRole() async {
    final prefs = await SharedPreferences.getInstance();
    final userId = prefs.getString('user_id');
    final userRole = prefs.getString('user_role');
    final hospitalId = prefs.getString('hospital_id');

    if (userId == null || userRole == null) {
      throw Exception('User not authenticated');
    }

    Map<String, String> queryParams = {
      'userId': userId,
      'role': userRole,
    };

    // Add hospital ID for hospital staff
    if (['doctor', 'nurse', 'admin'].contains(userRole) && hospitalId != null) {
      queryParams['hospitalId'] = hospitalId;
    }

    final uri = Uri.parse('ws://localhost:5000').replace(
      queryParameters: queryParams,
    );

    await connect(uri.toString());
  }
}
```

5. 💬 Chat System Implementation

Chat Message Model

Copy

```
class ChatMessage {
  final String id;
  final String senderId;
  final String senderName;
  final String senderRole;
  final String recipientId;
  final String content;
  final String type;
  final DateTime timestamp;
  final bool isRead;
  final Map<String, dynamic>? metadata;

  ChatMessage({
    required this.id,
    required this.senderId,
    required this.senderName,
    required this.senderRole,
    required this.recipientId,
    required this.content,
    required this.type,
    required this.timestamp,
    this.isRead = false,
    this.metadata,
  });

  factory ChatMessage.fromJson(Map<String, dynamic> json) {
    return ChatMessage(
      id: json['id'],
      senderId: json['senderId'],
      senderName: json['senderName'],
      senderRole: json['senderRole'],
      recipientId: json['recipientId'],
      content: json['content'],
      type: json['type'],
      timestamp: DateTime.parse(json['timestamp']),
      isRead: json['isRead'] ?? false,
      metadata: json['metadata'],
    );
  }

  Map<String, dynamic> toJson() {
    return {
      'id': id,
      'senderId': senderId,
      'senderName': senderName,
      'senderRole': senderRole,
      'recipientId': recipientId,
      'content': content,
      'type': type,
      'timestamp': timestamp.toIso8601String(),
      'isRead': isRead,
      'metadata': metadata,
    };
```

```
  }
}
```

Chat Service Provider

Copy

```
class ChatProvider extends ChangeNotifier {
  final WebSocketService _webSocketService = WebSocketService();
  final List<ChatMessage> _messages = [];
  final Map<String, List<ChatMessage>> _conversations = {};

  List<ChatMessage> get messages => List.unmodifiable(_messages);
  Map<String, List<ChatMessage>> get conversations => Map.unmodifiable(_conversations);

  void initialize() {
    _webSocketService.messageStream.listen((data) {
      if (data['type'] == 'chat_message') {
        _handleChatMessage(data);
      } else if (data['type'] == 'message_read') {
        _handleMessageRead(data);
      }
    });
  }

  void _handleChatMessage(Map<String, dynamic> data) {
    final message = ChatMessage.fromJson(data['payload']);
    _messages.add(message);

    // Organize by conversation
    final conversationKey = _getConversationKey(message.senderId, message.recipientId);
    _conversations.putIfAbsent(conversationKey, () => []);
    _conversations[conversationKey]!.add(message);

    notifyListeners();
  }

  void _handleMessageRead(Map<String, dynamic> data) {
    final messageId = data['payload']['messageId'];
    final messageIndex = _messages.indexWhere((m) => m.id == messageId);

    if (messageIndex != -1) {
      final message = _messages[messageIndex];
      _messages[messageIndex] = ChatMessage(
        id: message.id,
        senderId: message.senderId,
        senderName: message.senderName,
        senderRole: message.senderRole,
        recipientId: message.recipientId,
        content: message.content,
        type: message.type,
        timestamp: message.timestamp,
        isRead: true,
        metadata: message.metadata,
      );
      notifyListeners();
    }
  }

  Future<void> sendMessage(String recipientId, String content, {String type = 'text'}) async {
    final messageData = {
      'type': 'send_message',
      'payload': {
        'recipientId': recipientId,
        'content': content,
```

```
        'messageType': type,
        'timestamp': DateTime.now().toIso8601String(),
      },
    };

    _webSocketService.sendMessage(messageData);
  }

  Future<void> markMessageAsRead(String messageId) async {
    final data = {
      'type': 'mark_message_read',
      'payload': {
        'messageId': messageId,
      },
    };

    _webSocketService.sendMessage(data);
  }

  String _getConversationKey(String userId1, String userId2) {
    final users = [userId1, userId2]..sort();
    return users.join('_');
  }

  List<ChatMessage> getConversation(String otherUserId, String currentUserId) {
    final key = _getConversationKey(otherUserId, currentUserId);
    return _conversations[key] ?? [];
  }
}
```

Chat UI Widget

Copy

```
class ChatScreen extends StatefulWidget {
  final String recipientId;
  final String recipientName;

  const ChatScreen({
    Key? key,
    required this.recipientId,
    required this.recipientName,
  }) : super(key: key);

  @override
  _ChatScreenState createState() => _ChatScreenState();
}

class _ChatScreenState extends State<ChatScreen> {
  final TextEditingController _messageController = TextEditingController();
  final ScrollController _scrollController = ScrollController();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.recipientName),
        backgroundColor: Colors.blue[600],
      ),
      body: Column(
        children: [
          Expanded(
            child: Consumer<ChatProvider>(
              builder: (context, chatProvider, child) {
                final currentUserId = context.read<UserProvider>().currentUser?.id ?? '';
                final messages = chatProvider.getConversation(widget.recipientId, currentUserId);

                return ListView.builder(
                  controller: _scrollController,
                  itemCount: messages.length,
                  itemBuilder: (context, index) {
                    final message = messages[index];
                    final isOwn = message.senderId == currentUserId;

                    return _buildMessageBubble(message, isOwn);
                  },
                );
              },
            ),
          ),
          _buildMessageInput(),
        ],
      ),
    );
  }

  Widget _buildMessageBubble(ChatMessage message, bool isOwn) {
    return Container(
      margin: EdgeInsets.symmetric(vertical: 4, horizontal: 8),
      child: Row(
        mainAxisAlignment: isOwn ? MainAxisAlignment.end : MainAxisAlignment.start,
        children: [
```

```
        if (!isOwn) _buildAvatar(message.senderRole),
        Flexible(
          child: Container(
            padding: EdgeInsets.symmetric(vertical: 10, horizontal: 14),
            decoration: BoxDecoration(
              color: isOwn ? Colors.blue[600] : Colors.grey[300],
              borderRadius: BorderRadius.circular(20),
            ),
            child: Column(
              crossAxisAlignment: CrossAxisAlignment.start,
              children: [
                if (!isOwn)
                  Text(
                    message.senderName,
                    style: TextStyle(
                      fontSize: 12,
                      fontWeight: FontWeight.bold,
                      color: Colors.grey[600],
                    ),
                  ),
                Text(
                  message.content,
                  style: TextStyle(
                    color: isOwn ? Colors.white : Colors.black87,
                  ),
                ),
                SizedBox(height: 4),
                Row(
                  mainAxisSize: MainAxisSize.min,
                  children: [
                    Text(
                      _formatTime(message.timestamp),
                      style: TextStyle(
                        fontSize: 10,
                        color: isOwn ? Colors.white70 : Colors.grey[600],
                      ),
                    ),
                    if (isOwn) ...[
                      SizedBox(width: 4),
                      Icon(
                        message.isRead ? Icons.done_all : Icons.done,
                        size: 12,
                        color: message.isRead ? Colors.green : Colors.white70,
                      ),
                    ],
                  ],
                ),
              ],
            ),
          ),
        ),
        if (isOwn) _buildAvatar(message.senderRole),
      ],
    ),
  );
}

Widget _buildAvatar(String role) {
  IconData icon;
  Color color;

  switch (role) {
```

```
      case 'doctor':
        icon = Icons.medical_services;
        color = Colors.blue;
        break;
      case 'nurse':
        icon = Icons.local_hospital;
        color = Colors.green;
        break;
      case 'patient':
        icon = Icons.person;
        color = Colors.orange;
        break;
      default:
        icon = Icons.person;
        color = Colors.grey;
    }

    return Container(
      width: 32,
      height: 32,
      margin: EdgeInsets.symmetric(horizontal: 8),
      decoration: BoxDecoration(
        color: color,
        shape: BoxShape.circle,
      ),
      child: Icon(icon, color: Colors.white, size: 16),
    );
  }

  Widget _buildMessageInput() {
    return Container(
      padding: EdgeInsets.all(8),
      decoration: BoxDecoration(
        color: Colors.white,
        boxShadow: [
          BoxShadow(
            offset: Offset(0, -2),
            blurRadius: 6,
            color: Colors.black12,
          ),
        ],
      ),
      child: Row(
        children: [
          Expanded(
            child: TextField(
              controller: _messageController,
              decoration: InputDecoration(
                hintText: 'Type a message ... ',
                border: OutlineInputBorder(
                  borderRadius: BorderRadius.circular(25),
                ),
                contentPadding: EdgeInsets.symmetric(horizontal: 16, vertical: 8),
              ),
              maxLines: null,
            ),
          ),
          SizedBox(width: 8),
          FloatingActionButton.small(
            onPressed: _sendMessage,
            backgroundColor: Colors.blue[600],
            child: Icon(Icons.send, color: Colors.white),
```

```
          ),
        ],
      ),
    );
  }

  void _sendMessage() {
    final content = _messageController.text.trim();
    if (content.isNotEmpty) {
      context.read<ChatProvider>().sendMessage(widget.recipientId, content);
      _messageController.clear();

      // Scroll to bottom
      WidgetsBinding.instance.addPostFrameCallback((_) {
        _scrollController.animateTo(
          _scrollController.position.maxScrollExtent,
          duration: Duration(milliseconds: 300),
          curve: Curves.easeOut,
        );
      });
    }
  }

  String _formatTime(DateTime timestamp) {
    return '${timestamp.hour.toString().padLeft(2, '0')}:${timestamp.minute.toString().padLeft(2,
'0')}';
  }
}
```

6. 🚨 Emergency System Implementation

Emergency Alert Model

Copy

```
class EmergencyAlert {
  final String id;
  final String patientId;
  final String patientName;
  final String type;
  final String severity;
  final String description;
  final Map<String, double> location;
  final DateTime timestamp;
  final String status;
  final String? assignedToId;
  final String? assignedToName;
  final Map<String, dynamic>? metadata;

  EmergencyAlert({
    required this.id,
    required this.patientId,
    required this.patientName,
    required this.type,
    required this.severity,
    required this.description,
    required this.location,
    required this.timestamp,
    required this.status,
    this.assignedToId,
    this.assignedToName,
    this.metadata,
  });

  factory EmergencyAlert.fromJson(Map<String, dynamic> json) {
    return EmergencyAlert(
      id: json['id'],
      patientId: json['patientId'],
      patientName: json['patientName'],
      type: json['type'],
      severity: json['severity'],
      description: json['description'],
      location: Map<String, double>.from(json['location']),
      timestamp: DateTime.parse(json['timestamp']),
      status: json['status'],
      assignedToId: json['assignedToId'],
      assignedToName: json['assignedToName'],
      metadata: json['metadata'],
    );
  }
}
```

Emergency Service Provider

Copy

```
class EmergencyProvider extends ChangeNotifier {
  final WebSocketService _webSocketService = WebSocketService();
  final List<EmergencyAlert> _activeAlerts = [];
  EmergencyAlert? _currentAlert;

  List<EmergencyAlert> get activeAlerts => List.unmodifiable(_activeAlerts);
  EmergencyAlert? get currentAlert => _currentAlert;

  void initialize() {
    _webSocketService.messageStream.listen((data) {
      switch (data['type']) {
        case 'emergency_alert':
          _handleEmergencyAlert(data);
          break;
        case 'emergency_update':
          _handleEmergencyUpdate(data);
          break;
        case 'emergency_assigned':
          _handleEmergencyAssigned(data);
          break;
        case 'emergency_resolved':
          _handleEmergencyResolved(data);
          break;
      }
    });
  }

  void _handleEmergencyAlert(Map<String, dynamic> data) {
    final alert = EmergencyAlert.fromJson(data['payload']);
    _activeAlerts.add(alert);

    // Show notification
    _showEmergencyNotification(alert);

    notifyListeners();
  }

  void _handleEmergencyUpdate(Map<String, dynamic> data) {
    final alertId = data['payload']['alertId'];
    final updates = data['payload']['updates'];

    final index = _activeAlerts.indexWhere((alert) => alert.id == alertId);
    if (index != -1) {
      // Update alert with new information
      // Implementation would depend on your specific update structure
      notifyListeners();
    }
  }

  Future<void> createEmergencyAlert({
    required String type,
    required String severity,
    required String description,
    Map<String, double>? location,
  }) async {
    final alertData = {
      'type': 'create_emergency',
      'payload': {
```

```
        'alertType': type,
        'severity': severity,
        'description': description,
        'location': location ?? await _getCurrentLocation(),
        'timestamp': DateTime.now().toIso8601String(),
      },
    };

    _webSocketService.sendMessage(alertData);
  }

  Future<void> acceptEmergencyAlert(String alertId) async {
    final data = {
      'type': 'accept_emergency',
      'payload': {
        'alertId': alertId,
      },
    };

    _webSocketService.sendMessage(data);
  }

  Future<void> updateEmergencyStatus(String alertId, String status, {String? notes}) async {
    final data = {
      'type': 'update_emergency_status',
      'payload': {
        'alertId': alertId,
        'status': status,
        'notes': notes,
        'timestamp': DateTime.now().toIso8601String(),
      },
    };

    _webSocketService.sendMessage(data);
  }

  Future<Map<String, double>> _getCurrentLocation() async {
    // Implementation for getting current location
    final position = await Geolocator.getCurrentPosition();
    return {
      'latitude': position.latitude,
      'longitude': position.longitude,
    };
  }

  void _showEmergencyNotification(EmergencyAlert alert) {
    // Show system notification
    final NotificationService = locator<NotificationService>();
    NotificationService.showEmergencyNotification(
      title: 'Emergency Alert: ${alert.type}',
      body: '${alert.patientName} - ${alert.description}',
      data: {'alertId': alert.id, 'type': 'emergency'},
    );
  }
}
```

Emergency Alert Widget

Copy

```
class EmergencyAlertCard extends StatelessWidget {
  final EmergencyAlert alert;
  final VoidCallback? onAccept;
  final VoidCallback? onViewDetails;

  const EmergencyAlertCard({
    Key? key,
    required this.alert,
    this.onAccept,
    this.onViewDetails,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Card(
      margin: EdgeInsets.all(8),
      elevation: 4,
      color: _getSeverityColor(alert.severity),
      child: Padding(
        padding: EdgeInsets.all(16),
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            Row(
              children: [
                Icon(
                  _getEmergencyIcon(alert.type),
                  color: Colors.white,
                  size: 24,
                ),
                SizedBox(width: 8),
                Expanded(
                  child: Text(
                    alert.type.toUpperCase(),
                    style: TextStyle(
                      color: Colors.white,
                      fontSize: 18,
                      fontWeight: FontWeight.bold,
                    ),
                  ),
                ),
                Container(
                  padding: EdgeInsets.symmetric(horizontal: 8, vertical: 4),
                  decoration: BoxDecoration(
                    color: Colors.white.withOpacity(0.2),
                    borderRadius: BorderRadius.circular(12),
                  ),
                  child: Text(
                    alert.severity,
                    style: TextStyle(
                      color: Colors.white,
                      fontSize: 12,
                      fontWeight: FontWeight.bold,
                    ),
                  ),
                ),
              ],
            ),
```

```
            SizedBox(height: 12),
            Text(
              'Patient: ${alert.patientName}',
              style: TextStyle(
                color: Colors.white,
                fontSize: 16,
                fontWeight: FontWeight.w500,
              ),
            ),
            SizedBox(height: 8),
            Text(
              alert.description,
              style: TextStyle(
                color: Colors.white.withOpacity(0.9),
                fontSize: 14,
              ),
            ),
            SizedBox(height: 12),
            Row(
              children: [
                Icon(Icons.access_time, color: Colors.white, size: 16),
                SizedBox(width: 4),
                Text(
                  _formatTimestamp(alert.timestamp),
                  style: TextStyle(
                    color: Colors.white.withOpacity(0.8),
                    fontSize: 12,
                  ),
                ),
                Spacer(),
                if (alert.assignedToId == null && onAccept != null)
                  ElevatedButton(
                    onPressed: onAccept,
                    style: ElevatedButton.styleFrom(
                      backgroundColor: Colors.white,
                      foregroundColor: _getSeverityColor(alert.severity),
                    ),
                    child: Text('Accept'),
                  ),
                SizedBox(width: 8),
                ElevatedButton(
                  onPressed: onViewDetails,
                  style: ElevatedButton.styleFrom(
                    backgroundColor: Colors.white.withOpacity(0.2),
                    foregroundColor: Colors.white,
                  ),
                  child: Text('Details'),
                ),
              ],
            ),
          ],
        ),
      ),
    );
  }

  Color _getSeverityColor(String severity) {
    switch (severity.toLowerCase()) {
      case 'critical':
        return Colors.red[700]!;
      case 'high':
        return Colors.orange[700]!;
```

```
      case 'medium':
        return Colors.yellow[700]!;
      case 'low':
        return Colors.blue[700]!;
      default:
        return Colors.grey[700]!;
    }
  }

  IconData _getEmergencyIcon(String type) {
    switch (type.toLowerCase()) {
      case 'cardiac':
        return Icons.favorite;
      case 'respiratory':
        return Icons.air;
      case 'trauma':
        return Icons.local_hospital;
      case 'neurological':
        return Icons.psychology;
      case 'overdose':
        return Icons.warning;
      default:
        return Icons.emergency;
    }
  }

  String _formatTimestamp(DateTime timestamp) {
    final now = DateTime.now();
    final difference = now.difference(timestamp);

    if (difference.inMinutes < 1) {
      return 'Just now';
    } else if (difference.inMinutes < 60) {
      return '${difference.inMinutes}m ago';
    } else {
      return '${difference.inHours}h ago';
    }
  }
}
```

7. 📍 Real-time Location Sharing

Location Service Setup

⚠️ **Location Permissions Required:**
Add the following permissions to your `android/app/src/main/AndroidManifest.xml`:

Copy

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />
```

Copy

```
class LocationService {
  static LocationService? _instance;
  static LocationService get instance => _instance ??= LocationService._();
  LocationService._();

  StreamSubscription<Position>? _positionSubscription;
  final WebSocketService _webSocketService = WebSocketService();

  Future<bool> initializeLocationTracking() async {
    // Check permissions
    LocationPermission permission = await Geolocator.checkPermission();
    if (permission == LocationPermission.denied) {
      permission = await Geolocator.requestPermission();
      if (permission == LocationPermission.denied) {
        return false;
      }
    }

    if (permission == LocationPermission.deniedForever) {
      return false;
    }

    // Check location services
    bool serviceEnabled = await Geolocator.isLocationServiceEnabled();
    if (!serviceEnabled) {
      return false;
    }

    return true;
  }

  Future<void> startLocationSharing({
    int intervalSeconds = 30,
    double distanceFilter = 10.0,
  }) async {
    final hasPermission = await initializeLocationTracking();
    if (!hasPermission) {
      throw Exception('Location permission denied');
    }

    const LocationSettings locationSettings = LocationSettings(
      accuracy: LocationAccuracy.high,
      distanceFilter: 10,
    );

    _positionSubscription = Geolocator.getPositionStream(
      locationSettings: locationSettings,
    ).listen((Position position) {
      _shareLocation(position);
    });
  }

  void _shareLocation(Position position) {
    final locationData = {
      'type': 'location_update',
      'payload': {
        'latitude': position.latitude,
        'longitude': position.longitude,
        'accuracy': position.accuracy,
        'altitude': position.altitude,
        'speed': position.speed,
```

```
        'heading': position.heading,
        'timestamp': DateTime.now().toIso8601String(),
      },
    };

    _webSocketService.sendMessage(locationData);
  }

  Future<Position> getCurrentLocation() async {
    final hasPermission = await initializeLocationTracking();
    if (!hasPermission) {
      throw Exception('Location permission denied');
    }

    return await Geolocator.getCurrentPosition(
      desiredAccuracy: LocationAccuracy.high,
    );
  }

  void stopLocationSharing() {
    _positionSubscription?.cancel();
    _positionSubscription = null;
  }

  Future<void> shareEmergencyLocation() async {
    try {
      final position = await getCurrentLocation();
      final emergencyLocationData = {
        'type': 'emergency_location',
        'payload': {
          'latitude': position.latitude,
          'longitude': position.longitude,
          'accuracy': position.accuracy,
          'timestamp': DateTime.now().toIso8601String(),
          'isEmergency': true,
        },
      };

      _webSocketService.sendMessage(emergencyLocationData);
    } catch (e) {
      print('Error sharing emergency location: $e');
    }
  }
}
```

Location Provider

Copy

```
class LocationProvider extends ChangeNotifier {
  final WebSocketService _webSocketService = WebSocketService();
  final Map<String, UserLocation> _userLocations = {};
  UserLocation? _currentUserLocation;
  bool _isSharing = false;

  Map<String, UserLocation> get userLocations => Map.unmodifiable(_userLocations);
  UserLocation? get currentUserLocation => _currentUserLocation;
  bool get isSharing => _isSharing;

  void initialize() {
    _webSocketService.messageStream.listen((data) {
      switch (data['type']) {
        case 'location_update':
          _handleLocationUpdate(data);
          break;
        case 'user_location_list':
          _handleLocationList(data);
          break;
        case 'emergency_location':
          _handleEmergencyLocation(data);
          break;
      }
    });
  }

  void _handleLocationUpdate(Map<String, dynamic> data) {
    final location = UserLocation.fromJson(data['payload']);
    _userLocations[location.userId] = location;
    notifyListeners();
  }

  void _handleLocationList(Map<String, dynamic> data) {
    final locations = data['payload']['locations'] as List;
    _userLocations.clear();

    for (final locationData in locations) {
      final location = UserLocation.fromJson(locationData);
      _userLocations[location.userId] = location;
    }

    notifyListeners();
  }

  void _handleEmergencyLocation(Map<String, dynamic> data) {
    final location = UserLocation.fromJson(data['payload']);
    _userLocations[location.userId] = location;

    // Show emergency location notification
    _showEmergencyLocationAlert(location);

    notifyListeners();
  }

  Future<void> startLocationSharing() async {
    try {
      await LocationService.instance.startLocationSharing();
      _isSharing = true;
```

```
      notifyListeners();
    } catch (e) {
      print('Error starting location sharing: $e');
      rethrow;
    }
  }

  void stopLocationSharing() {
    LocationService.instance.stopLocationSharing();
    _isSharing = false;
    notifyListeners();
  }

  Future<void> requestNearbyUsers(double radiusKm) async {
    try {
      final currentPosition = await LocationService.instance.getCurrentLocation();

      final requestData = {
        'type': 'request_nearby_users',
        'payload': {
          'latitude': currentPosition.latitude,
          'longitude': currentPosition.longitude,
          'radiusKm': radiusKm,
        },
      };

      _webSocketService.sendMessage(requestData);
    } catch (e) {
      print('Error requesting nearby users: $e');
    }
  }

  void _showEmergencyLocationAlert(UserLocation location) {
    // Implementation for showing emergency location alert
    final NotificationService = locator<NotificationService>();
    NotificationService.showLocationAlert(
      title: 'Emergency Location Shared',
      body: '${location.userName} has shared their emergency location',
      data: {'userId': location.userId, 'type': 'emergency_location'},
    );
  }

  double calculateDistance(UserLocation location1, UserLocation location2) {
    return Geolocator.distanceBetween(
      location1.latitude,
      location1.longitude,
      location2.latitude,
      location2.longitude,
    );
  }
}
```

Location Model

Copy

```
class UserLocation {
  final String userId;
  final String userName;
  final String userRole;
  final double latitude;
  final double longitude;
  final double? accuracy;
  final double? altitude;
  final double? speed;
  final double? heading;
  final DateTime timestamp;
  final bool isEmergency;
  final Map<String, dynamic>? metadata;

  UserLocation({
    required this.userId,
    required this.userName,
    required this.userRole,
    required this.latitude,
    required this.longitude,
    this.accuracy,
    this.altitude,
    this.speed,
    this.heading,
    required this.timestamp,
    this.isEmergency = false,
    this.metadata,
  });

  factory UserLocation.fromJson(Map<String, dynamic> json) {
    return UserLocation(
      userId: json['userId'],
      userName: json['userName'],
      userRole: json['userRole'],
      latitude: json['latitude'].toDouble(),
      longitude: json['longitude'].toDouble(),
      accuracy: json['accuracy']?.toDouble(),
      altitude: json['altitude']?.toDouble(),
      speed: json['speed']?.toDouble(),
      heading: json['heading']?.toDouble(),
      timestamp: DateTime.parse(json['timestamp']),
      isEmergency: json['isEmergency'] ?? false,
      metadata: json['metadata'],
    );
  }

  Map<String, dynamic> toJson() {
    return {
      'userId': userId,
      'userName': userName,
      'userRole': userRole,
      'latitude': latitude,
      'longitude': longitude,
      'accuracy': accuracy,
      'altitude': altitude,
      'speed': speed,
      'heading': heading,
      'timestamp': timestamp.toIso8601String(),
```

```
      'isEmergency': isEmergency,
      'metadata': metadata,
    };
  }
}
```

8. 📋 Message Types & Data Structures

WebSocket Message Protocol

📋 **Message Format:**
All WebSocket messages follow this structure:

```
{
  "type": "message_type",
  "payload": {
    // Message-specific data
  },
  "timestamp": "2024-01-15T10:30:00.000Z"
}
```

Outgoing Message Types

1. Send Chat Message

```
{
  "type": "send_message",
  "payload": {
    "recipientId": "user123",
    "content": "Hello, how are you?",
    "messageType": "text", // text, image, file, location
    "metadata": {
      "urgency": "normal" // normal, high, emergency
    }
  }
}
```

2. Mark Message as Read

```
{
  "type": "mark_message_read",
  "payload": {
    "messageId": "msg456"
  }
}
```

3. Create Emergency Alert

Copy

```
{
  "type": "create_emergency",
  "payload": {
    "alertType": "cardiac", // cardiac, respiratory, trauma, etc.
    "severity": "critical", // low, medium, high, critical
    "description": "Patient experiencing chest pain",
    "location": {
      "latitude": 40.7128,
      "longitude": -74.0060
    },
    "vitals": {
      "heartRate": 120,
      "bloodPressure": "140/90",
      "oxygenSaturation": 95
    }
  }
}
```

4. Accept Emergency

Copy

```
{
  "type": "accept_emergency",
  "payload": {
    "alertId": "emergency123"
  }
}
```

5. Update Emergency Status

Copy

```
{
  "type": "update_emergency_status",
  "payload": {
    "alertId": "emergency123",
    "status": "en_route", // pending, accepted, en_route, on_scene, resolved
    "notes": "ETA 5 minutes",
    "location": {
      "latitude": 40.7130,
      "longitude": -74.0058
    }
  }
}
```

6. Share Location

```
{
  "type": "location_update",
  "payload": {
    "latitude": 40.7128,
    "longitude": -74.0060,
    "accuracy": 5.0,
    "altitude": 10.0,
    "speed": 0.0,
    "heading": 180.0
  }
}
```

7. Request Nearby Users

```
{
  "type": "request_nearby_users",
  "payload": {
    "latitude": 40.7128,
    "longitude": -74.0060,
    "radiusKm": 5.0,
    "userTypes": ["doctor", "nurse", "emergency"] // Optional filter
  }
}
```

8. Join Room/Channel

```
{
  "type": "join_room",
  "payload": {
    "roomId": "emergency_dispatch_room",
    "roomType": "emergency" // chat, emergency, announcement
  }
}
```

9. Leave Room/Channel

```
{
  "type": "leave_room",
  "payload": {
    "roomId": "emergency_dispatch_room"
  }
}
```

10. Update User Status

```
{
  "type": "update_status",
  "payload": {
    "status": "available", // available, busy, away, emergency
    "customMessage": "On duty until 6 PM"
  }
}
```

Incoming Message Types

1. Chat Message Received

```
{
  "type": "chat_message",
  "payload": {
    "id": "msg123",
    "senderId": "user456",
    "senderName": "Dr. Smith",
    "senderRole": "doctor",
    "recipientId": "user789",
    "content": "Patient in room 301 needs attention",
    "messageType": "text",
    "timestamp": "2024-01-15T10:30:00.000Z",
    "metadata": {
      "urgency": "high",
      "hospitalId": "hospital123"
    }
  }
}
```

2. Emergency Alert

Copy

```
{
  "type": "emergency_alert",
  "payload": {
    "id": "emergency123",
    "patientId": "patient456",
    "patientName": "John Doe",
    "type": "cardiac",
    "severity": "critical",
    "description": "Cardiac arrest in ER",
    "location": {
      "latitude": 40.7128,
      "longitude": -74.0060,
      "address": "123 Hospital St, New York, NY"
    },
    "timestamp": "2024-01-15T10:30:00.000Z",
    "status": "pending",
    "requiredSpecialties": ["cardiology", "emergency"],
    "estimatedResponseTime": 300
  }
}
```

3. Emergency Assignment

Copy

```
{
  "type": "emergency_assigned",
  "payload": {
    "alertId": "emergency123",
    "assignedToId": "doctor456",
    "assignedToName": "Dr. Johnson",
    "assignedRole": "doctor",
    "timestamp": "2024-01-15T10:32:00.000Z"
  }
}
```

4. Location Update

Copy

```
{
  "type": "location_update",
  "payload": {
    "userId": "user123",
    "userName": "Nurse Kate",
    "userRole": "nurse",
    "latitude": 40.7128,
    "longitude": -74.0060,
    "accuracy": 5.0,
    "timestamp": "2024-01-15T10:30:00.000Z",
    "isEmergency": false
  }
}
```

5. User Status Change

Copy

```
{
  "type": "user_status_change",
  "payload": {
    "userId": "user123",
    "userName": "Dr. Smith",
    "oldStatus": "available",
    "newStatus": "busy",
    "customMessage": "In surgery",
    "timestamp": "2024-01-15T10:30:00.000Z"
  }
}
```

6. System Notification

Copy

```
{
  "type": "system_notification",
  "payload": {
    "id": "notif123",
    "title": "System Maintenance",
    "message": "System will be down for maintenance at 2 AM",
    "severity": "info", // info, warning, error
    "targetRoles": ["all"], // or specific roles
    "timestamp": "2024-01-15T10:30:00.000Z",
    "expiresAt": "2024-01-16T10:30:00.000Z"
  }
}
```

7. Message Read Receipt

Copy

```
{
  "type": "message_read",
  "payload": {
    "messageId": "msg123",
    "readBy": "user456",
    "readAt": "2024-01-15T10:35:00.000Z"
  }
}
```

8. User Connected/Disconnected

Copy

```
{
  "type": "user_connection_change",
  "payload": {
    "userId": "user123",
    "userName": "Dr. Smith",
    "userRole": "doctor",
    "isConnected": true,
    "timestamp": "2024-01-15T10:30:00.000Z"
  }
}
```

Error Message Format

Copy

```
{
  "type": "error",
  "payload": {
    "code": "AUTH_REQUIRED",
    "message": "Authentication required to access this resource",
    "details": {
      "requiredRole": "doctor",
      "currentRole": null
    },
    "timestamp": "2024-01-15T10:30:00.000Z"
  }
}
```

Common Error Codes

🔍 **Error Code Reference:**

- **AUTH_REQUIRED:** User authentication required
- **INVALID_ROLE:** User role not authorized for action
- **HOSPITAL_ID_REQUIRED:** Hospital staff must provide hospital ID
- **INVALID_MESSAGE_FORMAT:** Message format is incorrect
- **RECIPIENT_NOT_FOUND:** Message recipient does not exist
- **EMERGENCY_NOT_FOUND:** Emergency alert does not exist
- **LOCATION_PERMISSION_DENIED:** Location access not granted
- **RATE_LIMIT_EXCEEDED:** Too many messages sent

9. 📱 Complete Flutter Implementation Examples

Complete Application Structure

Copy

```
lib/
├── main.dart
├── models/
│   ├── chat_message.dart
│   ├── emergency_alert.dart
│   ├── user_location.dart
│   └── user.dart
├── providers/
│   ├── websocket_provider.dart
│   ├── chat_provider.dart
│   ├── emergency_provider.dart
│   ├── location_provider.dart
│   └── user_provider.dart
├── services/
│   ├── websocket_service.dart
│   ├── auth_service.dart
│   ├── location_service.dart
│   └── notification_service.dart
├── screens/
│   ├── dashboard_screen.dart
│   ├── chat_screen.dart
│   ├── emergency_screen.dart
│   ├── location_screen.dart
│   └── login_screen.dart
└── widgets/
    ├── connection_status_widget.dart
    ├── emergency_alert_card.dart
    ├── chat_message_bubble.dart
    └── location_map_widget.dart
```

Complete Main Application Setup

Copy

```dart
// main.dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'package:shared_preferences/shared_preferences.dart';

import 'providers/websocket_provider.dart';
import 'providers/chat_provider.dart';
import 'providers/emergency_provider.dart';
import 'providers/location_provider.dart';
import 'providers/user_provider.dart';
import 'screens/login_screen.dart';
import 'screens/dashboard_screen.dart';

void main() async {
  WidgetsFlutterBinding.ensureInitialized();

  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (_) => WebSocketProvider()),
        ChangeNotifierProvider(create: (_) => UserProvider()),
        ChangeNotifierProvider(create: (_) => ChatProvider()),
        ChangeNotifierProvider(create: (_) => EmergencyProvider()),
        ChangeNotifierProvider(create: (_) => LocationProvider()),
      ],
      child: HPlusApp(),
    ),
  );
}

class HPlusApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'HPlus Medical System',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        visualDensity: VisualDensity.adaptivePlatformDensity,
      ),
      home: SplashScreen(),
      routes: {
        '/login': (context) => LoginScreen(),
        '/dashboard': (context) => DashboardScreen(),
      },
    );
  }
}

class SplashScreen extends StatefulWidget {
  @override
  _SplashScreenState createState() => _SplashScreenState();
}

class _SplashScreenState extends State<SplashScreen> {
  @override
  void initState() {
    super.initState();
    _checkAuthStatus();
```

```
  }

  Future<void> _checkAuthStatus() async {
    final prefs = await SharedPreferences.getInstance();
    final userId = prefs.getString('user_id');

    await Future.delayed(Duration(seconds: 2)); // Splash delay

    if (userId != null) {
      // Auto-login if credentials exist
      final userProvider = context.read<UserProvider>();
      final success = await userProvider.autoLogin();

      if (success) {
        Navigator.pushReplacementNamed(context, '/dashboard');
      } else {
        Navigator.pushReplacementNamed(context, '/login');
      }
    } else {
      Navigator.pushReplacementNamed(context, '/login');
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.blue[600],
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Icon(
              Icons.local_hospital,
              size: 100,
              color: Colors.white,
            ),
            SizedBox(height: 20),
            Text(
              'HPlus Medical System',
              style: TextStyle(
                fontSize: 28,
                fontWeight: FontWeight.bold,
                color: Colors.white,
              ),
            ),
            SizedBox(height: 40),
            CircularProgressIndicator(
              valueColor: AlwaysStoppedAnimation<Color>(Colors.white),
            ),
          ],
        ),
      ),
    );
  }
}
```

Complete Dashboard Implementation

Copy

```
// screens/dashboard_screen.dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

class DashboardScreen extends StatefulWidget {
  @override
  _DashboardScreenState createState() => _DashboardScreenState();
}

class _DashboardScreenState extends State<DashboardScreen> {
  int _currentIndex = 0;

  final List<Widget> _screens = [
    DashboardHomeScreen(),
    ChatListScreen(),
    EmergencyScreen(),
    LocationScreen(),
  ];

  @override
  void initState() {
    super.initState();
    _initializeServices();
  }

  Future<void> _initializeServices() async {
    final webSocketProvider = context.read<WebSocketProvider>();
    final chatProvider = context.read<ChatProvider>();
    final emergencyProvider = context.read<EmergencyProvider>();
    final locationProvider = context.read<LocationProvider>();

    // Initialize all providers
    chatProvider.initialize();
    emergencyProvider.initialize();
    locationProvider.initialize();

    // Connect WebSocket
    await webSocketProvider.connectWithUserRole();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('HPlus Medical'),
        backgroundColor: Colors.blue[600],
        actions: [
          // Connection status indicator
          Consumer<WebSocketProvider>(
            builder: (context, provider, child) {
              return Container(
                margin: EdgeInsets.only(right: 16),
                child: Row(
                  children: [
                    Icon(
                      provider.isConnected ? Icons.wifi : Icons.wifi_off,
                      color: provider.isConnected ? Colors.green : Colors.red,
                    ),
```

```
                  SizedBox(width: 4),
                  Text(
                    provider.isConnected ? 'Online' : 'Offline',
                    style: TextStyle(fontSize: 12),
                  ),
                ],
              ),
            );
          },
        ),
        PopupMenuButton(
          itemBuilder: (context) => [
            PopupMenuItem(
              value: 'profile',
              child: Row(
                children: [
                  Icon(Icons.person),
                  SizedBox(width: 8),
                  Text('Profile'),
                ],
              ),
            ),
            PopupMenuItem(
              value: 'settings',
              child: Row(
                children: [
                  Icon(Icons.settings),
                  SizedBox(width: 8),
                  Text('Settings'),
                ],
              ),
            ),
            PopupMenuItem(
              value: 'logout',
              child: Row(
                children: [
                  Icon(Icons.logout),
                  SizedBox(width: 8),
                  Text('Logout'),
                ],
              ),
            ),
          ],
          onSelected: (value) => _handleMenuAction(value),
        ),
      ],
    ),
    body: IndexedStack(
      index: _currentIndex,
      children: _screens,
    ),
    bottomNavigationBar: BottomNavigationBar(
      currentIndex: _currentIndex,
      onTap: (index) => setState(() => _currentIndex = index),
      type: BottomNavigationBarType.fixed,
      selectedItemColor: Colors.blue[600],
      items: [
        BottomNavigationBarItem(
          icon: Icon(Icons.dashboard),
          label: 'Dashboard',
        ),
        BottomNavigationBarItem(
```

```
            icon: _buildChatBadge(),
            label: 'Chat',
          ),
          BottomNavigationBarItem(
            icon: _buildEmergencyBadge(),
            label: 'Emergency',
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.location_on),
            label: 'Location',
          ),
        ],
      ),
    );
  }

  Widget _buildChatBadge() {
    return Consumer<ChatProvider>(
      builder: (context, chatProvider, child) {
        final unreadCount = chatProvider.getUnreadMessageCount();

        return Stack(
          children: [
            Icon(Icons.chat),
            if (unreadCount > 0)
              Positioned(
                right: 0,
                top: 0,
                child: Container(
                  padding: EdgeInsets.all(2),
                  decoration: BoxDecoration(
                    color: Colors.red,
                    borderRadius: BorderRadius.circular(8),
                  ),
                  constraints: BoxConstraints(
                    minWidth: 16,
                    minHeight: 16,
                  ),
                  child: Text(
                    '$unreadCount',
                    style: TextStyle(
                      color: Colors.white,
                      fontSize: 10,
                    ),
                    textAlign: TextAlign.center,
                  ),
                ),
              ),
          ],
        );
      },
    );
  }

  Widget _buildEmergencyBadge() {
    return Consumer<EmergencyProvider>(
      builder: (context, emergencyProvider, child) {
        final activeAlerts = emergencyProvider.activeAlerts.length;

        return Stack(
          children: [
            Icon(Icons.emergency),
```

```
          if (activeAlerts > 0)
            Positioned(
              right: 0,
              top: 0,
              child: Container(
                padding: EdgeInsets.all(2),
                decoration: BoxDecoration(
                  color: Colors.red,
                  borderRadius: BorderRadius.circular(8),
                ),
                constraints: BoxConstraints(
                  minWidth: 16,
                  minHeight: 16,
                ),
                child: Text(
                  '$activeAlerts',
                  style: TextStyle(
                    color: Colors.white,
                    fontSize: 10,
                  ),
                  textAlign: TextAlign.center,
                ),
              ),
            ),
        ],
      );
    },
  );
}

void _handleMenuAction(String action) {
  switch (action) {
    case 'profile':
      // Navigate to profile screen
      break;
    case 'settings':
      // Navigate to settings screen
      break;
    case 'logout':
      _showLogoutDialog();
      break;
  }
}

void _showLogoutDialog() {
  showDialog(
    context: context,
    builder: (context) => AlertDialog(
      title: Text('Logout'),
      content: Text('Are you sure you want to logout?'),
      actions: [
        TextButton(
          onPressed: () => Navigator.pop(context),
          child: Text('Cancel'),
        ),
        TextButton(
          onPressed: () async {
            // Disconnect WebSocket
            await context.read<WebSocketProvider>().disconnect();

            // Clear user data
            await context.read<UserProvider>().logout();
```

```
              // Navigate to login
              Navigator.pushNamedAndRemoveUntil(
                context,
                '/login',
                (route) => false,
              );
            },
            child: Text('Logout'),
          ),
        ],
      ),
    );
  }
}
```

User Provider Implementation

Copy

```
// providers/user_provider.dart
import 'package:flutter/foundation.dart';
import 'package:shared_preferences/shared_preferences.dart';
import '../services/auth_service.dart';
import '../models/user.dart';

class UserProvider extends ChangeNotifier {
  User? _currentUser;
  bool _isLoading = false;

  User? get currentUser => _currentUser;
  bool get isLoading => _isLoading;
  bool get isLoggedIn => _currentUser != null;

  Future<bool> login(String email, String password) async {
    _isLoading = true;
    notifyListeners();

    try {
      final result = await AuthService.authenticateUser(email, password);

      if (result['success']) {
        _currentUser = User.fromJson(result['user']);
        _isLoading = false;
        notifyListeners();
        return true;
      } else {
        _isLoading = false;
        notifyListeners();
        return false;
      }
    } catch (e) {
      _isLoading = false;
      notifyListeners();
      return false;
    }
  }

  Future<bool> autoLogin() async {
    try {
      final prefs = await SharedPreferences.getInstance();
      final userId = prefs.getString('user_id');
      final userRole = prefs.getString('user_role');
      final userName = prefs.getString('user_name');
      final userEmail = prefs.getString('user_email');
      final hospitalId = prefs.getString('hospital_id');

      if (userId != null && userRole != null) {
        _currentUser = User(
          id: userId,
          name: userName ?? '',
          email: userEmail ?? '',
          role: userRole,
          hospitalId: hospitalId,
        );
        notifyListeners();
        return true;
      }
```

```
      return false;
    } catch (e) {
      return false;
    }
  }

  Future<void> logout() async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.clear();
    _currentUser = null;
    notifyListeners();
  }

  Future<void> updateUserStatus(String status, {String? customMessage}) async {
    if (_currentUser != null) {
      // Update user status through WebSocket
      // Implementation depends on your WebSocket service
    }
  }
}
```

10. ⚠️ Error Handling & Best Practices

Comprehensive Error Handling

Copy

```
class WebSocketErrorHandler {
 static void handleWebSocketError(String error, BuildContext context) {
   print('WebSocket Error: $error');

   if (error.contains('Authentication required')) {
     _handleAuthError(context);
   } else if (error.contains('Hospital ID is required')) {
     _handleHospitalIdError(context);
   } else if (error.contains('Connection timeout')) {
     _handleConnectionTimeout(context);
   } else if (error.contains('Unable to reconnect')) {
     _handleReconnectionFailure(context);
   } else {
     _handleGenericError(error, context);
   }
 }

 static void _handleAuthError(BuildContext context) {
   showDialog(
     context: context,
     barrierDismissible: false,
     builder: (context) => AlertDialog(
       title: Row(
         children: [
           Icon(Icons.error, color: Colors.red),
           SizedBox(width: 8),
           Text('Authentication Error'),
         ],
       ),
       content: Text(
         'Your session has expired. Please log in again to continue.',
       ),
       actions: [
         TextButton(
           onPressed: () {
             Navigator.of(context).pop();
             Navigator.pushNamedAndRemoveUntil(
               context,
               '/login',
               (route) => false,
             );
           },
           child: Text('Login Again'),
         ),
       ],
     ),
   );
 }

 static void _handleConnectionTimeout(BuildContext context) {
   ScaffoldMessenger.of(context).showSnackBar(
     SnackBar(
       content: Row(
```

```
          children: [
            Icon(Icons.wifi_off, color: Colors.white),
            SizedBox(width: 8),
            Text('Connection timeout. Attempting to reconnect...'),
          ],
        ),
        backgroundColor: Colors.orange,
        duration: Duration(seconds: 3),
        action: SnackBarAction(
          label: 'Retry',
          textColor: Colors.white,
          onPressed: () {
            context.read<WebSocketProvider>().reconnect();
          },
        ),
      ),
    );
  }

  static void _handleReconnectionFailure(BuildContext context) {
    showModalBottomSheet(
      context: context,
      isDismissible: false,
      builder: (context) => Container(
        padding: EdgeInsets.all(20),
        child: Column(
          mainAxisSize: MainAxisSize.min,
          children: [
            Icon(Icons.cloud_off, size: 48, color: Colors.red),
            SizedBox(height: 16),
            Text(
              'Connection Failed',
              style: TextStyle(fontSize: 20, fontWeight: FontWeight.bold),
            ),
            SizedBox(height: 8),
            Text(
              'Unable to connect to the server. Please check your internet connection and try
again.',
              textAlign: TextAlign.center,
            ),
            SizedBox(height: 20),
            Row(
              children: [
                Expanded(
                  child: ElevatedButton(
                    onPressed: () {
                      Navigator.pop(context);
                      context.read<WebSocketProvider>().reconnect();
                    },
                    child: Text('Retry Connection'),
                  ),
                ),
                SizedBox(width: 16),
                Expanded(
                  child: TextButton(
                    onPressed: () {
                      Navigator.pop(context);
                      // Continue in offline mode
                    },
                    child: Text('Work Offline'),
                  ),
                ),
```

```
                ],
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

Performance Best Practices

> **Performance Optimization Tips:**
>
> - **Message Throttling:** Limit message sending rate to prevent spam
> - **Memory Management:** Clear old messages and limit chat history
> - **Efficient Rebuilds:** Use Consumer widgets strategically
> - **Background Tasks:** Handle WebSocket in background threads
> - **State Persistence:** Save critical data locally

Message Rate Limiting

Copy

```
class MessageRateLimiter {
  static const int maxMessagesPerMinute = 30;
  static const Duration timeWindow = Duration(minutes: 1);

  final List<DateTime> _messageTimes = [];

  bool canSendMessage() {
    final now = DateTime.now();

    // Remove old messages outside time window
    _messageTimes.removeWhere(
      (time) => now.difference(time) > timeWindow,
    );

    if (_messageTimes.length >= maxMessagesPerMinute) {
      return false;
    }

    _messageTimes.add(now);
    return true;
  }

  Duration getWaitTime() {
    if (_messageTimes.isEmpty) return Duration.zero;

    final oldestMessage = _messageTimes.first;
    final timeSinceOldest = DateTime.now().difference(oldestMessage);

    if (timeSinceOldest >= timeWindow) {
      return Duration.zero;
    }

    return timeWindow - timeSinceOldest;
  }
}
```

Memory Management

Copy

```
class ChatProvider extends ChangeNotifier {
  static const int maxMessagesPerConversation = 100;
  static const int maxTotalMessages = 500;

  void _cleanupOldMessages() {
    // Remove old messages when limit is exceeded
    if (_messages.length > maxTotalMessages) {
      final messagesToRemove = _messages.length - maxTotalMessages;
      _messages.removeRange(0, messagesToRemove);
    }

    // Clean up conversations
    for (final conversationKey in _conversations.keys) {
      final messages = _conversations[conversationKey]!;
      if (messages.length > maxMessagesPerConversation) {
        final messagesToRemove = messages.length - maxMessagesPerConversation;
        messages.removeRange(0, messagesToRemove);
      }
    }
  }

  @override
  void dispose() {
    _webSocketService.messageStream.listen((data) {}).cancel();
    super.dispose();
  }
}
```

Offline Mode Support

Copy

```
class OfflineMessageQueue {
  static const String _storageKey = 'offline_messages';
  final List<Map<String, dynamic>> _queuedMessages = [];

  Future<void> queueMessage(Map<String, dynamic> message) async {
    message['queuedAt'] = DateTime.now().toIso8601String();
    _queuedMessages.add(message);
    await _saveToStorage();
  }

  Future<void> sendQueuedMessages(WebSocketService webSocketService) async {
    if (_queuedMessages.isEmpty) return;

    final messagesToSend = List<Map<String, dynamic>>.from(_queuedMessages);
    _queuedMessages.clear();

    for (final message in messagesToSend) {
      try {
        await webSocketService.sendMessage(message);
        await Future.delayed(Duration(milliseconds: 100)); // Throttle
      } catch (e) {
        // Re-queue failed messages
        _queuedMessages.add(message);
      }
    }

    await _saveToStorage();
  }

  Future<void> _saveToStorage() async {
    final prefs = await SharedPreferences.getInstance();
    final jsonString = json.encode(_queuedMessages);
    await prefs.setString(_storageKey, jsonString);
  }

  Future<void> loadFromStorage() async {
    final prefs = await SharedPreferences.getInstance();
    final jsonString = prefs.getString(_storageKey);

    if (jsonString != null) {
      final List<dynamic> decoded = json.decode(jsonString);
      _queuedMessages.clear();
      _queuedMessages.addAll(decoded.cast<Map<String, dynamic>>());
    }
  }
}
```

Security Best Practices

🔒 **Security Guidelines:**

- **Token Security:** Store JWT tokens securely using flutter_secure_storage
- **Input Validation:** Validate all user inputs before sending
- **HTTPS/WSS:** Always use secure connections in production
- **Role Verification:** Verify user permissions on both client and server
- **Message Encryption:** Consider encrypting sensitive messages

Secure Token Storage

Copy

```
import 'package:flutter_secure_storage/flutter_secure_storage.dart';

class SecureStorageService {
  static const FlutterSecureStorage _storage = FlutterSecureStorage(
    aOptions: AndroidOptions(
      encryptedSharedPreferences: true,
    ),
    iOptions: IOSOptions(
      accessibility: IOSAccessibility.first_unlock_this_device,
    ),
  );

  static Future<void> storeToken(String token) async {
    await _storage.write(key: 'auth_token', value: token);
  }

  static Future<String?> getToken() async {
    return await _storage.read(key: 'auth_token');
  }

  static Future<void> deleteToken() async {
    await _storage.delete(key: 'auth_token');
  }

  static Future<void> storeUserCredentials(Map<String, String> credentials) async {
    for (final entry in credentials.entries) {
      await _storage.write(key: entry.key, value: entry.value);
    }
  }

  static Future<void> clearAllSecureData() async {
    await _storage.deleteAll();
  }
}
```

11. 🧪 Testing & Debugging

WebSocket Connection Testing

Copy

```
// test/websocket_test.dart
import 'package:flutter_test/flutter_test.dart';
import 'package:mockito/mockito.dart';
import 'package:web_socket_channel/web_socket_channel.dart';

import '../lib/services/websocket_service.dart';

class MockWebSocketChannel extends Mock implements WebSocketChannel {}

void main() {
  group('WebSocket Service Tests', () {
    late WebSocketService webSocketService;
    late MockWebSocketChannel mockChannel;

    setUp(() {
      webSocketService = WebSocketService();
      mockChannel = MockWebSocketChannel();
    });

    test('should connect successfully with valid parameters', () async {
      // Arrange
      const testUrl = 'ws://localhost:5000?userId=test123&role=doctor';

      // Act
      await webSocketService.connect(testUrl);

      // Assert
      expect(webSocketService.isConnected, isTrue);
    });

    test('should handle connection errors gracefully', () async {
      // Arrange
      const invalidUrl = 'ws://invalid-url:9999';

      // Act & Assert
      expect(
        () => webSocketService.connect(invalidUrl),
        throwsException,
      );
    });

    test('should send messages in correct format', () async {
      // Arrange
      const testMessage = {
        'type': 'send_message',
        'payload': {
          'recipientId': 'user123',
          'content': 'Test message',
          'messageType': 'text',
        },
      };

      // Act
```

```
      webSocketService.sendMessage(testMessage);

      // Assert
      verify(mockChannel.sink.add(any)).called(1);
    });

    test('should reconnect after connection loss', () async {
      // Arrange
      await webSocketService.connect('ws://localhost:5000');

      // Simulate connection loss
      webSocketService.handleConnectionError('Connection lost');

      // Act
      await Future.delayed(Duration(seconds: 2));

      // Assert
      expect(webSocketService.isConnected, isTrue);
    });
  });
}
```

Provider Testing

Copy

```
// test/chat_provider_test.dart
import 'package:flutter_test/flutter_test.dart';
import 'package:mockito/mockito.dart';

import '../lib/providers/chat_provider.dart';
import '../lib/models/chat_message.dart';

void main() {
  group('Chat Provider Tests', () {
    late ChatProvider chatProvider;

    setUp(() {
      chatProvider = ChatProvider();
    });

    test('should add new message to conversation', () {
      // Arrange
      final testMessage = ChatMessage(
        id: 'msg123',
        senderId: 'user1',
        senderName: 'John Doe',
        senderRole: 'doctor',
        recipientId: 'user2',
        content: 'Test message',
        type: 'text',
        timestamp: DateTime.now(),
      );

      // Act
      chatProvider.addMessage(testMessage);

      // Assert
      expect(chatProvider.messages.length, equals(1));
      expect(chatProvider.messages.first.content, equals('Test message'));
    });

    test('should organize messages by conversation', () {
      // Arrange
      final message1 = ChatMessage(
        id: 'msg1',
        senderId: 'user1',
        senderName: 'John',
        senderRole: 'doctor',
        recipientId: 'user2',
        content: 'Hello',
        type: 'text',
        timestamp: DateTime.now(),
      );

      final message2 = ChatMessage(
        id: 'msg2',
        senderId: 'user2',
        senderName: 'Jane',
        senderRole: 'nurse',
        recipientId: 'user1',
        content: 'Hi there',
        type: 'text',
        timestamp: DateTime.now(),
```

```
      );

      // Act
      chatProvider.addMessage(message1);
      chatProvider.addMessage(message2);

      // Assert
      final conversation = chatProvider.getConversation('user1', 'user2');
      expect(conversation.length, equals(2));
    });

    test('should mark message as read', () {
      // Arrange
      final testMessage = ChatMessage(
        id: 'msg123',
        senderId: 'user1',
        senderName: 'John',
        senderRole: 'doctor',
        recipientId: 'user2',
        content: 'Test',
        type: 'text',
        timestamp: DateTime.now(),
        isRead: false,
      );

      chatProvider.addMessage(testMessage);

      // Act
      chatProvider.markMessageAsRead('msg123');

      // Assert
      final updatedMessage = chatProvider.messages.firstWhere(
        (m) => m.id == 'msg123',
      );
      expect(updatedMessage.isRead, isTrue);
    });
  });
}
```

Integration Testing

Copy

```
// integration_test/app_test.dart
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:integration_test/integration_test.dart';

import 'package:hplus_app/main.dart' as app;

void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();

  group('HPlus App Integration Tests', () {
    testWidgets('complete login and chat flow', (WidgetTester tester) async {
      // Start the app
      app.main();
      await tester.pumpAndSettle();

      // Test login flow
      await tester.enterText(
        find.byKey(Key('email_field')),
        'doctor@example.com',
      );
      await tester.enterText(
        find.byKey(Key('password_field')),
        'password123',
      );
      await tester.tap(find.byKey(Key('login_button')));
      await tester.pumpAndSettle(Duration(seconds: 3));

      // Verify dashboard appears
      expect(find.text('HPlus Medical'), findsOneWidget);
      expect(find.byIcon(Icons.dashboard), findsOneWidget);

      // Test WebSocket connection
      await tester.pump(Duration(seconds: 2));
      expect(find.text('Online'), findsOneWidget);

      // Navigate to chat
      await tester.tap(find.byIcon(Icons.chat));
      await tester.pumpAndSettle();

      // Test sending a message
      await tester.enterText(
        find.byKey(Key('message_input')),
        'Test integration message',
      );
      await tester.tap(find.byIcon(Icons.send));
      await tester.pumpAndSettle();

      // Verify message appears
      expect(find.text('Test integration message'), findsOneWidget);
    });

    testWidgets('emergency alert flow', (WidgetTester tester) async {
      app.main();
      await tester.pumpAndSettle();

      // Login first
      await _performLogin(tester);
```

```
      // Navigate to emergency tab
      await tester.tap(find.byIcon(Icons.emergency));
      await tester.pumpAndSettle();

      // Create emergency alert
      await tester.tap(find.byKey(Key('create_emergency_button')));
      await tester.pumpAndSettle();

      await tester.tap(find.text('Cardiac'));
      await tester.tap(find.text('Critical'));
      await tester.enterText(
        find.byKey(Key('emergency_description')),
        'Patient experiencing chest pain',
      );

      await tester.tap(find.byKey(Key('submit_emergency')));
      await tester.pumpAndSettle();

      // Verify alert was created
      expect(find.text('Emergency alert sent'), findsOneWidget);
    });
  });
}

Future<void> _performLogin(WidgetTester tester) async {
  await tester.enterText(
    find.byKey(Key('email_field')),
    'doctor@example.com',
  );
  await tester.enterText(
    find.byKey(Key('password_field')),
    'password123',
  );
  await tester.tap(find.byKey(Key('login_button')));
  await tester.pumpAndSettle(Duration(seconds: 3));
}
```

Debugging Tools

🔧 **Debugging Tools & Techniques:**

- **WebSocket Inspector:** Use browser dev tools to monitor WebSocket traffic
- **Flutter Inspector:** Debug widget tree and state changes
- **Network Monitoring:** Track API calls and responses
- **Console Logging:** Add strategic print statements
- **Error Tracking:** Implement crash reporting with Firebase Crashlytics

Debug Logging Service

Copy

```
class DebugLogger {
  static const bool _isDebugMode = kDebugMode;

  static void logWebSocketEvent(String event, Map<String, dynamic>? data) {
    if (_isDebugMode) {
      print('🔌 WebSocket Event: $event');
      if (data ≠ null) {
        print('📄 Data: ${json.encode(data)}');
      }
      print('⏰ Time: ${DateTime.now().toIso8601String()}');
      print('━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━');
    }
  }

  static void logError(String context, dynamic error, StackTrace? stackTrace) {
    if (_isDebugMode) {
      print('❌ Error in $context: $error');
      if (stackTrace ≠ null) {
        print('📍 Stack Trace:');
        print(stackTrace.toString());
      }
      print('━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━');
    }
  }

  static void logUserAction(String action, Map<String, dynamic>? details) {
    if (_isDebugMode) {
      print('👤 User Action: $action');
      if (details ≠ null) {
        print('📋 Details: ${json.encode(details)}');
      }
      print('⏰ Time: ${DateTime.now().toIso8601String()}');
      print('━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━');
    }
  }
}
```

Performance Monitoring

Copy

```
class PerformanceMonitor {
  static final Map<String, DateTime> _startTimes = {};

  static void startTiming(String operation) {
    _startTimes[operation] = DateTime.now();
  }

  static void endTiming(String operation) {
    final startTime = _startTimes[operation];
    if (startTime ≠ null) {
      final duration = DateTime.now().difference(startTime);
      DebugLogger.logUserAction(
        'Performance: $operation completed',
        {'duration_ms': duration.inMilliseconds},
      );
      _startTimes.remove(operation);
    }
  }

  static void measureWidgetBuild(String widgetName, VoidCallback buildFunction) {
    startTiming('Build $widgetName');
    buildFunction();
    endTiming('Build $widgetName');
  }
}
```

12. 📚 Appendix & Quick Reference

Dependencies Reference

> 📦 **Required Flutter Dependencies:**
> Add these to your `pubspec.yaml` file:

Copy

```
dependencies:
  flutter:
    sdk: flutter

  # WebSocket & Networking
  web_socket_channel: ^2.4.0
  http: ^1.1.0

  # State Management
  provider: ^6.1.1

  # Storage
  shared_preferences: ^2.2.2
  flutter_secure_storage: ^9.0.0

  # Location Services
  geolocator: ^10.1.0
  permission_handler: ^11.0.1

  # UI & Navigation
  flutter_local_notifications: ^16.1.0

  # Utilities
  connectivity_plus: ^5.0.1

dev_dependencies:
  flutter_test:
    sdk: flutter
  integration_test:
    sdk: flutter
  mockito: ^5.4.2
  build_runner: ^2.4.7
```

Environment Configuration

Development Environment

Copy

```
// lib/config/env_config.dart
class EnvConfig {
  static const String environment = String.fromEnvironment(
    'ENVIRONMENT',
    defaultValue: 'development',
  );

  static const String wsBaseUrl = String.fromEnvironment(
    'WS_BASE_URL',
    defaultValue: 'ws://localhost:5000',
  );

  static const String apiBaseUrl = String.fromEnvironment(
    'API_BASE_URL',
    defaultValue: 'http://localhost:3000',
  );

  static bool get isDevelopment => environment == 'development';
  static bool get isProduction => environment == 'production';

  static String get webSocketUrl {
    switch (environment) {
      case 'production':
        return 'wss://api.hplus-medical.com/ws';
      case 'staging':
        return 'wss://staging-api.hplus-medical.com/ws';
      default:
        return wsBaseUrl;
    }
  }
}
```

Production Build Configuration

Copy

```
# Build commands for different environments

# Development
flutter run --dart-define=ENVIRONMENT=development

# Staging
flutter build apk --dart-define=ENVIRONMENT=staging --dart-define=WS_BASE_URL=wss://staging-
api.hplus-medical.com/ws

# Production
flutter build apk --dart-define=ENVIRONMENT=production --dart-define=WS_BASE_URL=wss://api.hplus-
medical.com/ws
```

Troubleshooting Guide

🔍 **Common Issues & Solutions:**

1. WebSocket Connection Fails

- **Check URL:** Ensure WebSocket URL is correct and accessible
- **Network Permissions:** Add INTERNET permission to AndroidManifest.xml
- **Firewall:** Check if firewall is blocking WebSocket connections
- **Server Status:** Verify the WebSocket server is running

2. Authentication Errors

- **Token Expiry:** Implement token refresh mechanism
- **Role Validation:** Ensure user role is correctly set
- **Hospital ID:** Verify hospital staff have hospitalId parameter

3. Location Services Not Working

- **Permissions:** Request location permissions properly
- **GPS Settings:** Check if location services are enabled
- **Background Location:** Handle background location permissions

4. Messages Not Delivered

- **Connection Status:** Check WebSocket connection status
- **Message Format:** Validate message structure
- **Rate Limiting:** Check if rate limits are exceeded
- **Recipient ID:** Verify recipient exists and is online

Code Templates

Quick WebSocket Setup

Copy

```
// Quick setup template for new Flutter project
void main() {
  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (_) => WebSocketProvider()),
        ChangeNotifierProvider(create: (_) => ChatProvider()),
        ChangeNotifierProvider(create: (_) => EmergencyProvider()),
      ],
      child: MyApp(),
    ),
  );
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: Consumer<WebSocketProvider>(
          builder: (context, wsProvider, child) {
            if (!wsProvider.isConnected) {
              return ElevatedButton(
                onPressed: () async {
                  await wsProvider.connect(
                    'ws://localhost:5000?userId=user123&role=doctor'
                  );
                },
                child: Text('Connect to WebSocket'),
              );
            }
            return Text('Connected to WebSocket!');
          },
        ),
      ),
    );
  }
}
```

Emergency Alert Template

```
// Quick emergency alert setup
void sendEmergencyAlert() {
  final emergencyData = {
    'type': 'create_emergency',
    'payload': {
      'alertType': 'cardiac',
      'severity': 'critical',
      'description': 'Patient experiencing chest pain',
      'location': {
        'latitude': 40.7128,
        'longitude': -74.0060,
      },
    },
  };

  context.read<WebSocketProvider>().sendMessage(emergencyData);
}
```

API Reference Summary

Message Type	Direction	Purpose
`send_message`	Outgoing	Send chat message to another user
`chat_message`	Incoming	Receive chat message from another user
`create_emergency`	Outgoing	Create new emergency alert
`emergency_alert`	Incoming	Receive emergency alert notification
`location_update`	Both	Share/receive location updates
`accept_emergency`	Outgoing	Accept emergency alert assignment
`mark_message_read`	Outgoing	Mark message as read
`user_status_change`	Incoming	Receive user status updates

Support & Resources

🔗 **Additional Resources:**

- **Flutter WebSocket Documentation:** web_socket_channel package
- **Provider State Management:** provider package
- **Location Services:** geolocator package
- **Local Notifications:** flutter_local_notifications
- **Flutter Testing:** Official Flutter Testing Guide

📞 **Getting Help:**
If you encounter issues or need assistance:

- Check the troubleshooting guide above
- Review the WebSocket server logs
- Test with the provided code examples
- Verify all dependencies are properly installed
- Contact the development team for advanced support

Version History

Version	Date	Changes
v1.0.0	2024-01-15	Initial release with complete WebSocket integration guide
v1.1.0	2024-01-20	Added emergency system and location sharing
v1.2.0	2024-01-25	Enhanced error handling and testing examples

End of Documentation

Thank you for using the HPlus Medical System WebSocket Integration Guide!

Generated on January 15, 2024 • Version 1.2.0