CollabChat - Frontend Documentation

Project Overview

CollabChat is a real-time collaborative chat application built with **Flutter**. It enables users to communicate through private one-on-one chats and group conversations. The application features end-to-end encryption for secure messaging, real-time socket communication, and a beautiful Material Design UI.

Key Features

- **User Authentication**: Secure registration and login system with JWT tokens
- **Private Chats**: One-on-one messaging with individual users
- **Group Chats**: Create and manage group conversations with multiple members
- **Real-time Messaging**: WebSocket-based instant message delivery via Socket.IO
- **End-to-End Encryption**: All messages are encrypted using AES-256 encryption
- **File Sharing**: Users can share files in chats (stored on Cloudinary)
- **Online Status**: Real-time display of user online/offline status
- **Typing Indicators**: Shows when other users are typing
- **User Management**: View all users and manage group members

Backend URL

`https://collabchatbackend.onrender.com`

Dependencies & Libraries

Core Flutter

- **flutter**: SDK for building cross-platform mobile/web applications
- **flutter_lints**: Dart linting rules for code quality

State Management

- **riverpod** (v2.4.0): Provider-based reactive dependency injection
- **flutter_riverpod** (v2.4.0): Flutter integration for Riverpod state management

Navigation

- **go_router** (v13.0.0): Declarative routing for Flutter with deep linking support

HTTP & Networking

- **dio** (v5.3.0): Powerful HTTP client with interceptors and request/response handling
- **socket_io_client** (v2.0.0): WebSocket client for real-time Socket.IO communication

Data Models & Serialization

- **freezed_annotation** (v2.4.0): Annotation package for code generation with Freezed
- **json_annotation** (v4.9.0): Annotation package for JSON serialization
- **json_serializable** (v6.7.0): Dart/Flutter serialization library for generating fromJson/toJson
- **freezed** (v2.4.0): Code generator for immutable model creation
- **build_runner** (v2.4.0): Build system for Dart code generation

Storage

- **shared_preferences** (v2.2.0): Local persistent storage for non-sensitive data
- **flutter_secure_storage** (v9.2.0): Secure storage for sensitive data like authentication tokens

File Handling

- **file_picker** (v6.0.0): Cross-platform file picker for selecting files to share

Date/Time

- **intl** (v0.19.0): Internationalization and localization library with date formatting

Encryption

- **encrypt** (v5.0.3): AES encryption/decryption for end-to-end message security
- **crypto** (v3.0.3): Cryptographic hash functions for key generation

UI

- **cupertino_icons** (v1.0.8): iOS-style icons for Flutter

Folder Structure

```
frontend/
```

```
lib/
   main.dart                          # App entry point
   router.dart                        # Route configuration with GoRouter

   constants/
      api_constants.dart             # API endpoints, URLs, socket events
      app_theme.dart                 # Color palettes and Material theme

   models/
      user_model.dart                # User and AuthResponse data models
      user_model.freezed.dart        # Auto-generated User model (Freezed)
      user_model.g.dart              # Auto-generated JSON serialization
      chat_model.dart                # Chat data model
      chat_model.freezed.dart        # Auto-generated Chat model
      chat_model.g.dart              # Auto-generated JSON serialization
      message_model.dart             # Message data model with encryption
      message_model.freezed.dart     # Auto-generated Message model
      group_model.dart               # Group data model with custom converters
      group_model.freezed.dart       # Auto-generated Group model
      group_model.g.dart             # Auto-generated JSON serialization

   providers/
      auth_provider.dart             # Authentication logic (login, register, logout)
      chat_provider.dart             # Chat messages and socket listeners
      user_provider.dart             # Users list and group management

   services/
      api_service.dart               # Dio HTTP client with authentication
      socket_service.dart            # Socket.IO WebSocket management
      encryption_service.dart        # AES encryption/decryption
      storage_service.dart           # Secure and local storage management

   screens/
      splash_screen.dart             # Initial loading screen
      login_screen.dart              # User login form
      register_screen.dart           # User registration form
      home_screen.dart               # Main chat list (private + groups)
      chat_screen.dart               # Private chat messages display
      group_chat_screen.dart         # Group chat messages display
      create_group_screen.dart       # Group creation form
      group_members_screen.dart      # View/manage group members
      profile_screen.dart            # User profile information

   widgets/
      chat_bubble.dart               # Message bubble UI component
      message_input.dart             # Message input field with file picker
```

```
        user_tile.dart                # User list item UI component

      utils/
        validators.dart              # Form validation functions
        extensions.dart              # Dart extension methods

  pubspec.yaml                        # Project dependencies
  README.md                           # Project documentation
```

API Structure

Base URL

```
https://collabchatbackend.onrender.com/api/v1
```

Authentication Endpoints

Register User

```
POST /auth/register
Content-Type: application/json

{
  "username": "string",
  "password": "string"
}

Response (200):
{
  "token": "jwt_token",
  "user": {
    "_id": "user_id",
    "username": "username",
    "isOnline": false,
    "lastSeen": "2024-02-11T10:30:00Z",
    "createdAt": "2024-02-11T10:30:00Z",
    "updatedAt": "2024-02-11T10:30:00Z"
  }
}
```

Login User

```
POST /auth/login
Content-Type: application/json

{
  "username": "string",
  "password": "string"
}

Response (200): Same format as register
```

User Endpoints

Get Current User

```
GET /users/me
Authorization: Bearer {token}

Response (200):
{
  "_id": "user_id",
  "username": "username",
  "isOnline": true,
  "lastSeen": "2024-02-11T10:30:00Z"
}
```

Get All Users (excluding current user)

```
GET /users
Authorization: Bearer {token}

Response (200):
[
  {
    "_id": "user_id",
    "username": "username",
    "isOnline": true,
    "lastSeen": "2024-02-11T10:30:00Z"
  },
  ...
]
```

Chat Endpoints

Get Private Chat Messages

```
GET /chats/private/{currentUserId}/{otherUserId}?limit=20&offset=0
Authorization: Bearer {token}

Response (200):
[
  {
    "_id": "message_id",
    "senderId": "user_id",
    "chatId": "chat_id",
    "content": "message_content",
    "type": "text",
    "createdAt": "2024-02-11T10:30:00Z",
    "sender": { user object },
    "isEncrypted": true
  },
  ...
]
```

Get Group Chat Messages

```
GET /chats/group/{groupId}?limit=20&offset=0
Authorization: Bearer {token}

Response (200): Same format as private messages
```

Send Message

```
POST /chats/send
Authorization: Bearer {token}
Content-Type: application/json

{
  "chatId": "chat_id",          # For group chats
  "recipientId": "user_id",     # For private chats
  "content": "message_content",
  "type": "text"
}

Response (200): Message object
```

Group Endpoints

Create Group

```
POST /groups
Authorization: Bearer {token}
Content-Type: application/json

{
  "name": "group_name",
  "members": ["user_id_1", "user_id_2"]
}

Response (200):
```

```
{
  "_id": "group_id",
  "name": "group_name",
  "members": ["user_id_1", "user_id_2"],
  "adminId": "admin_user_id",
  "createdAt": "2024-02-11T10:30:00Z",
  "chatId": "associated_chat_id"
}
```

Get My Groups

```
GET /groups/my
Authorization: Bearer {token}

Response (200):
[
  {
    "_id": "group_id",
    "name": "group_name",
    "members": ["user_id_1", "user_id_2"],
    "adminId": "admin_user_id",
    "createdAt": "2024-02-11T10:30:00Z"
  },
  ...
]
```

Add Group Member

```
POST /groups/{groupId}
Authorization: Bearer {token}
Content-Type: application/json

{
  "userId": "user_id"
}

Response (200): Updated group object
```

WebSocket Events (Socket.IO)

Connection

```
Event: connect
Triggered when socket successfully connects to server
```

Disconnect

```
Event: disconnect
Triggered when socket disconnects from server
```

Join Chat

```
Emit: join
{
  "chatId": "chat_id",
  "recipientId": "user_id"  # Optional, for private chats
}
```

Send Message

```
Emit: send_message
{
  "chatId": "chat_id",
  "message": "message_content",
  "type": "text",
  "recipientId": "user_id"  # Optional
}
```

Receive Message

```
Listen: receive_message
{
  "_id": "message_id",
  "senderId": "user_id",
  "chatId": "chat_id",
  "content": "message_content",
  "type": "text",
  "createdAt": "2024-02-11T10:30:00Z",
  "isEncrypted": true
}
```

Online Users

```
Listen: online_users
["user_id_1", "user_id_2", ...]
```

Typing Indicator

```
Emit: typing
{
  "chatId": "chat_id"
}

Listen: typing
```

"user_id"

Detailed Code Explanation

1. Main Entry Point (`main.dart`)

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

  // Initialize storage service before running app
  final storageService = StorageService();
  await storageService.init();

  runApp(
    ProviderScope(
      overrides: [
        storageServiceProvider.overrideWithValue(storageService),
      ],
      child: const MyApp(),
    ),
  );
}
```

Purpose: Entry point of the Flutter application.

What it does: 1. `WidgetsFlutterBinding.ensureInitialized()` - Ensures Flutter binding is initialized before async operations 2. Initializes `StorageService` to set up shared preferences and secure storage 3. Wraps app with `ProviderScope` for Riverpod state management 4. Overrides the storage provider with the initialized instance 5. Builds the Material app with routing configuration

2. Router Configuration (`router.dart`)

```
final routerProvider = Provider((ref) {
  final authState = ref.watch(currentUserProvider);

  return GoRouter(
    initialLocation: '/splash',
    redirect: (context, state) {
      // Public routes (no auth required)
      if (state.fullPath == '/splash' ||
          state.fullPath == '/login' ||
          state.fullPath == '/register') {
        return null;
      }
      // Check authentication for protected routes
      final isAuthenticated = authState.whenData((user) => user != null).value ?? false;

      if (!isAuthenticated) {
        return '/splash';
      }
      return null;
    },
    routes: [
      // All route definitions...
    ],
  );
});
```

Purpose: Manages application navigation and routing.

Key features: - **Initial Route**: Starts at `/splash` screen - **Redirect Logic**: Protects routes requiring authentication - **Route Types**: - Public routes: `/splash`, `/login`, `/register` - Protected routes: `/home`, `/chat/:userId`, `/group-chat/:groupId`, `/profile` - Uses `GoRouter` for declarative navigation with deep linking support

3. Authentication Provider (`providers/auth_provider.dart`)

```
final currentUserProvider =
    StateNotifierProvider<CurrentUserNotifier, AsyncValue<User?>>((ref) {
  return CurrentUserNotifier(ref);
});

class CurrentUserNotifier extends StateNotifier<AsyncValue<User?>> {
  // Login method
  Future<void> login(String username, String password) async {
    state = const AsyncValue.loading();
    state = await AsyncValue.guard(() async {
      final api = ref.read(apiServiceProvider);
      final storage = ref.read(storageServiceProvider);

      // 1. Call login API
      final response = await api.login(username, password);

      // 2. Save token and user data to secure storage
      await storage.saveToken(response.token);
      await storage.saveUserId(response.user.id);
      await storage.saveUsername(response.user.username);

      // 3. Connect WebSocket
      await ref.read(socketServiceProvider.notifier).connect(response.token);

      // 4. Refresh users list
      ref.invalidate(usersListProvider);

      return response.user;
    });
  }

  // Register method
  Future<void> register(String username, String password) async {
    // Similar flow to login
  }

  // Logout method
  Future<void> logout() async {
    // Clear storage and disconnect socket
  }
}
```

Purpose: Manages user authentication state (login, register, logout).

Key operations: 1. **Login**: Validates credentials, saves JWT token, connects WebSocket, refreshes user list 2. **Register**: Creates new account, saves token, initializes socket connection 3. **Logout**: Clears all stored data, disconnects socket, returns to login

4. Chat Provider (`providers/chat_provider.dart`)

```
final messagesProvider =
    StateNotifierProvider.family<
      MessagesNotifier,
      AsyncValue<List<Message>>,
      (String, bool)  // (chatId, isGroupChat)
    >((ref, params) {
      final (chatId, isGroupChat) = params;
      return MessagesNotifier(ref, chatId, isGroupChat: isGroupChat);
    });

class MessagesNotifier extends StateNotifier<AsyncValue<List<Message>>> {
  void _setupSocketListeners() {
    // Listen for incoming messages
    socket.onMessageReceived((message) {
      // Check if message is relevant to current chat
      if (_isRelevant(message)) {
        // Decrypt if encrypted
        if (message.isEncrypted) {
          message = decryptMessage(message);
        }
        // Add to state
        state.whenData((messages) {
          state = AsyncValue.data([...messages, message]);
        });
      }
    });
  }

  Future<void> _loadMessages() async {
    state = const AsyncValue.loading();
    state = await AsyncValue.guard(() async {
      final api = ref.read(apiServiceProvider);

      // Fetch messages from API
      final messages = isGroupChat
          ? await api.getGroupChatMessages(chatId)
          : await api.getPrivateChatMessages(currentUserId, chatId);

      // Decrypt all encrypted messages
      return messages.map((msg) {
        if (msg.isEncrypted) {
          return msg.copyWith(
            content: _encryptionService.decryptMessage(msg.content, msg.chatId)
          );
        }
        return msg;
      }).toList();
    });
  }
}
```

Purpose: Manages chat messages state with real-time updates.

Key features: 1. **Parameterized Provider**: Uses tuple (chatId, isGroupChat) for multiple chats 2. **Socket Integration**: Listens for real-time messages via WebSocket 3. **Encryption Handling**: Automatically decrypts encrypted messages 4. **Message Relevance**: Only adds messages relevant to current chat 5. **State Management**: Maintains asyncValue for loading/error/data states

5. API Service (`services/api_service.dart`)

```
class ApiService {
  late Dio _dio;
```

```dart
void _initializeDio() {
  _dio = Dio(BaseOptions(
    baseUrl: ApiConstants.apiBaseUrl,
    connectTimeout: ApiConstants.apiTimeout,
  ));

  // Add authentication interceptor
  _dio.interceptors.add(InterceptorsWrapper(
    onRequest: (options, handler) async {
      final token = await _storageService.getToken();
      if (token != null) {
        options.headers['Authorization'] = 'Bearer $token';
      }
      return handler.next(options);
    },
  ));
}

// Auth endpoints
Future<AuthResponse> login(String username, String password) async {
  final response = await _dio.post(
    ApiConstants.authLogin,
    data: {'username': username, 'password': password},
  );
  return AuthResponse.fromJson(response.data);
}

// Message endpoints
Future<List<Message>> getPrivateChatMessages(
  String currentUserId,
  String otherUserId, {
  int limit = 20,
  int offset = 0,
}) async {
  final response = await _dio.get(
    '${ApiConstants.chatsPrivate}/$currentUserId/$otherUserId',
    queryParameters: {'limit': limit, 'offset': offset},
  );
  return (response.data as List)
      .map((msg) => Message.fromJson(msg))
      .toList();
}

Future<Message> sendMessage(
  String chatId,
  String message, {
    String type = 'text',
    bool isGroupChat = false,
  }) async {
    final response = await _dio.post(
      ApiConstants.chatsSend,
      data: {'chatId': chatId, 'content': message, 'type': type},
    );
    return Message.fromJson(response.data);
  }
}
```

Purpose: Handles all HTTP requests to backend API.

Key features: 1. **Dio HTTP Client**: Powerful HTTP client with request/response interceptors 2. **Auto Authentication**: Automatically adds Bearer token to all requests 3. **Timeout Management**: 30-second timeout for all requests 4. **Error Handling**: Catches and logs Dio exceptions 5. **JSON Serialization**: Automatically converts responses to model objects

6. Socket Service (`services/socket_service.dart`)

```
class SocketService {
  late IO.Socket _socket;
  final String _token;

  Future<void> connect() async {
    _socket = IO.io(
      ApiConstants.socketUrl,
      IO.OptionBuilder()
        .setTransports(['websocket'])
        .setAuth({'token': _token})
        .build(),
    );

    _socket.on('connect', (_) {
      _isConnected = true;
      _notifyConnectionChanged(true);
    });

    // Listen for messages from server
    _socket.on(SocketEvents.receiveMessage, (data) {
      var message = Message.fromJson(data);

      // Decrypt if encrypted
      if (message.isEncrypted) {
        message = message.copyWith(
          content: _encryptionService.decryptMessage(
            message.content,
            message.chatId,
          ),
        );
      }

      _notifyMessageReceived(message);
    });

    _socket.on(SocketEvents.onlineUsers, (data) {
      _notifyUsersOnline(List<String>.from(data));
    });
  }

  void joinChat(String chatId, {String? recipientId}) {
    if (_isConnected) {
      _socket.emit(SocketEvents.join, {
        'chatId': chatId,
        if (recipientId != null) 'recipientId': recipientId,
      });
    }
  }

  Future<void> sendMessage(String chatId, String message) async {
    if (_isConnected) {
      _socket.emit(SocketEvents.sendMessage, {
        'chatId': chatId,
        'message': message,
      });
    }
  }

  void onMessageReceived(OnMessageReceived callback) {
    _messageListeners.add(callback);
  }
}
```

Purpose: Manages WebSocket connection and real-time events.

Key features: 1. **Socket.IO Integration**: Uses socket_io_client for WebSocket communication 2. **Auto Authentication**: Sends JWT token on connection 3. **Event Management**: Handles connect, disconnect, messages, online users, typing 4. **Message Decryption**: Decrypts messages before notifying listeners 5. **Listener Pattern**: Multiple listeners can subscribe to socket events 6. **Connection State**: Tracks connection status and auto-reconnects

7. Encryption Service (`services/encryption_service.dart`)

```dart
class EncryptionService {
  static const String ENCRYPTION_SECRET = 'collab-chat-encryption-key-32b!';

  // Generate unique key for each chat
  encrypt.Key _generateChatKey(String chatId) {
    final bytes = utf8.encode(chatId + ENCRYPTION_SECRET);
    final hash = sha256.convert(bytes);
    return encrypt.Key(Uint8List.fromList(hash.bytes));
  }

  String encryptMessage(String text, String chatId) {
    final key = _generateChatKey(chatId);
    final iv = encrypt.IV.fromSecureRandom(16);

    final encrypter = encrypt.Encrypter(
      encrypt.AES(key, mode: encrypt.AESMode.cbc, padding: 'PKCS7'),
    );

    final encrypted = encrypter.encrypt(text, iv: iv);
    return '${iv.base16}:${encrypted.base16}';  // Format: IV:EncryptedData
  }

  String decryptMessage(String encryptedText, String chatId) {
    final parts = encryptedText.split(':');
    final key = _generateChatKey(chatId);
    final iv = encrypt.IV.fromBase16(parts[0]);
    final encrypted = encrypt.Encrypted.fromBase16(parts[1]);

    final encrypter = encrypt.Encrypter(
      encrypt.AES(key, mode: encrypt.AESMode.cbc, padding: 'PKCS7'),
    );

    return encrypter.decrypt(encrypted, iv: iv);
  }
}
```

Purpose: Provides end-to-end encryption/decryption for messages.

Key features: 1. **AES-256 Encryption**: Uses Advanced Encryption Standard with 256-bit keys 2. **Chat-Specific Keys**: Each chat has unique key derived from chatId + secret 3. **IV (Initialization Vector)**: Random 16-byte IV for each message 4. **CBC Mode**: Uses CBC (Cipher Block Chaining) mode for security 5. **PKCS7 Padding**: Standard padding for block cipher

Encryption Format: `iv_hex_string:encrypted_data_hex_string`

8. Storage Service (services/storage_service.dart)

```
class StorageService {
  static const _secureStorage = FlutterSecureStorage();
  late SharedPreferences _prefs;

  Future<void> init() async {
    _prefs = await SharedPreferences.getInstance();
  }

  // Secure storage (for JWT token)
  Future<void> saveToken(String token) async {
    await _secureStorage.write(key: 'auth_token', value: token);
  }

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

  // Regular storage (for user info)
  Future<void> saveUsername(String username) async {
    await _prefs.setString('username', username);
  }

  String? getUsername() {
    return _prefs.getString('username');
  }

  Future<void> clearAll() async {
    await _secureStorage.delete(key: 'auth_token');
    await _prefs.clear();
  }
}
```

Purpose: Manages persistent local storage.

Storage Strategy: - **Secure Storage**: Sensitive data (JWT token) in encrypted keychain/keystore - **Shared Preferences**: User metadata (username, userId) in regular storage

9. Data Models

User Model

```
@freezed
class User with _$User {
  const factory User({
    @JsonKey(name: '_id') required String id,
    required String username,
    @Default(false) bool isOnline,
    @Default(null) DateTime? lastSeen,
    @JsonKey(name: 'createdAt') @Default(null) DateTime? createdAt,
    @JsonKey(name: 'updatedAt') @Default(null) DateTime? updatedAt,
  }) = _User;

  factory User.fromJson(Map<String, dynamic> json) => _$UserFromJson(json);
}
```

Uses Freezed for: - Immutability - Value equality - Copy mechanisms - JSON serialization/deserialization

Message Model

```
@Freezed(fromJson: false)
class Message with _$Message {
  const factory Message({
    @JsonKey(name: '_id') required String id,
    @SenderIdConverter() required String senderId,
    required String chatId,
    required String content,
    @Default('text') String type,
    required DateTime createdAt,
    @Default(null) User? sender,
    // File-specific fields
    @Default(null) String? fileUrl,
    // Encryption flag
    @Default(false) bool isEncrypted,
  }) = _Message;

  factory Message.fromJson(Map<String, dynamic> json) {
    // Custom deserialization logic for complex fields
  }
}
```

Features: - Custom converter for `senderId` (handles both String and Object) - File support with Cloudinary URL - Encryption flag for client-side decryption - Sender user object population

Group Model

```
@freezed
class Group with _$Group {
  const factory Group({
    @JsonKey(name: '_id') required String id,
    required String name,
    @MembersConverter() required List<String> members,
    @AdminIdConverter() required String adminId,
    required DateTime createdAt,
    @Default(null) String? description,
    @Default(null) String? chatId,
  }) = _Group;

  factory Group.fromJson(Map<String, dynamic> json) => _$GroupFromJson(json);
}
```

Custom Converters: - `AdminIdConverter`: Converts adminId from String or Object - `MembersConverter`: Converts members list from String array or Object array

10. UI Screens

Login Screen

- Username/password input fields
- Form validation using `Validators`
- Error messages display
- Loading indicator during authentication
- "Register" link to navigation

Home Screen

- **Two tabs**: Chats (private) and Groups
- **Search functionality** to filter users/groups
- Real-time online status indicator
- Floating action button to create groups
- Profile and logout options

Chat Screen (Private)

- Message list with scroll-to-bottom on new messages
- Message bubbles with sender distinction (left/right)
- Message timestamps in relative format (e.g., "5m ago")
- File message display with download link
- Online/offline status of other user
- Message input with file attachment
- Infinite scroll to load more messages

Group Chat Screen

- Similar to chat screen but for groups
- Multiple members in conversation
- Message sender name displayed above non-current-user messages
- Group info and member management

Create Group Screen

- Group name input field
- Multi-select user picker
- Validation before group creation

11. Widgets

ChatBubble Displays individual messages with: - Different styling for current user vs. others - File preview/download button for file messages - Timestamp display - Message type indicator

MessageInput Input field for sending messages with: - File attachment button (FilePicker integration) - Text input for message content - Send button - Typing indicator callback

12. Utilities

Validators Validates form inputs: - `validateUsername`: 3-20 chars, alphanumeric + _ - - `validatePassword`: Minimum 6 characters - `validateGroupName`: 2-50 characters - `validateMessage`: Maximum 5000 characters

Extensions Adds convenience methods: - **StringExtensions**: capitalize(), truncate() - **DateTimeExtensions**: - `toIST()`: Convert UTC to Indian Standard Time - `toFormattedTime()`: Format as "hh:mm AM/PM" - `toFormattedDate()`: Format as "MMM dd, yyyy" - `toRelativeTime()`: Display as "5m ago", "Yesterday", etc.

13. Theme Configuration (`constants/app_theme.dart`)

Color Palette: - **Primary**: Terracotta Brown (#A8553F) - **Accent**: Warm Gold (#D4A574) - **Background**: Cream Beige (#F5F1EB) - **Text Primary**: Dark Brown (#2C1810) - **Error**: Red (#D84315) - **Success**: Green (#689F38)

Material3 Theme with: - Rounded corners (12-16px border radius) - Gradient backgrounds for interactive elements - Soft shadows for depth - Smooth transitions

---

Application Flow

User Authentication Flow

1. **Splash Screen** → Check if user has valid token in storage
2. **Login/Register** → Authenticate with backend
3. **Save Credentials** → Store JWT token securely
4. **Connect Socket** → Establish WebSocket connection

5. **Home Screen** → Display chats and groups

Message Flow (Private Chat)

1. User opens chat with another user
2. **Load Messages** → Fetch chat history from API
3. **Decrypt Messages** → Use chat-specific encryption key
4. **Join Chat** → Emit socket join event with recipient ID
5. **Listen for Messages** → Socket listener receives real-time messages
6. **Send Message** → Encrypt locally, send via Socket.IO
7. **Receive Message** → Socket emits, listener decrypts, updates state

Message Flow (Group Chat)

Similar to private but: - Use group's chatId instead of recipient ID - Display sender name for each message - Manage multiple participant encryption keys

---

State Management Flow

Using **Riverpod Providers**:

```
currentUserProvider (StateNotifier)
- Manages logged-in user
- Handles auth state

       ↓

socketServiceProvider (StateNotifier)
- Manages WebSocket connection
- Emits/Listens to events

       ↓

messagesProvider.family (StateNotifier)
- Manages messages per chat
- Listens to socket events
- Decrypts messages

       ↓

UI Screens (ConsumerWidget)
- Watch providers
- Rebuild on state changes
```

Key Technologies & Patterns

1. **Freezed**: Immutable data models with JSON serialization
2. **Riverpod**: Reactive dependency injection and state management
3. **Socket.IO**: Real-time bidirectional WebSocket communication
4. **AES Encryption**: End-to-end message encryption/decryption
5. **Dio**: HTTP client with interceptors
6. **GoRouter**: Declarative routing with protection
7. **Material Design 3**: Modern UI framework

Security Considerations

1. **JWT Token Storage**: Stored in secure keychain/keystore (not regular storage)
2. **Message Encryption**: All messages encrypted with AES-256 before transmission
3. **HTTPS Only**: All API requests use HTTPS
4. **Chat-Specific Keys**: Encryption keys derived from chat ID + secret
5. **Token Refresh**: Token included in all authenticated requests
6. **Secure Storage**: Sensitive data never exposed in logs

Error Handling

- **API Errors**: Caught by Dio, logged, displayed in UI
- **Socket Errors**: Attempted reconnection with backoff
- **Encryption Errors**: Fallback to plain text if decryption fails
- **Navigation Errors**: Caught by GoRouter, fallback UI displayed
- **Form Validation**: Client-side validation before submission

---

Future Enhancements

1. Message reactions/emojis
2. Message editing/deletion
3. Voice/video calling
4. Message search functionality
5. Group chat notifications
6. Profile pictures/avatars
7. Message reactions
8. Document support for file sharing
9. Offline message queue

10. End-to-end encryption key exchange

Deployment

Frontend URL: Built as Flutter web/mobile app **Backend URL**: https://collabchatbackend.onrender.com **Deployment**: Can be deployed to: - Google Play Store (Android) - Apple App Store (iOS) - Firebase Hosting (Web) - AWS/Azure (Cloud services)

Build & Run

```
# Install dependencies
flutter pub get

# Generate code (Freezed, JSON serialization)
flutter pub run build_runner build

# Run development
flutter run

# Build for production
flutter build apk      # Android
flutter build ipa      # iOS
flutter build web      # Web
```

Last Updated: February 11, 2026 **Version**: 1.0.0+1 **Flutter SDK**: ^3.10.7