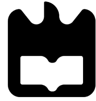

universidade de aveiro
theoria poiesis praxis

Mobile Computing

Final Flutter Project

Concert Experience Logger

An offline-first Flutter mobile application designed to capture, organize, and analyze live concert experiences through media sharing, sensor data integration, and secure backend synchronization.

Authors

Carolina Sofia Pereira da Silva • 113475 • MEI
Luana Carolina Cunha Reis • 131193 • MEI

Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática

January 6, 2026

Contents

Repository Information **5**

1 Introduction **5**

2 Project Context **6**
2.1 The Problem . . . 6
2.2 Our Solution . . . 6

3 Requirements **6**
3.1 Authentication and User Management . . . 7
3.2 Offline-First Architecture . . . 7
3.3 Event Management . . . 7
3.4 Media Capture and Sharing . . . 7
3.5 Sensor Integration . . . 7
3.6 Social Features . . . 8
3.7 Notifications . . . 8
3.8 Statistics and Analytics . . . 8
3.9 Location Features . . . 8

4 Architecture Diagram and Workflow **9**
4.1 System Components . . . 9
4.2 Architectural Principles . . . 9
4.3 Workflow . . . 10
4.3.1 Authentication Flow . . . 10
4.3.2 Data Synchronization Flow . . . 10
4.3.3 Media Upload Flow . . . 10
4.3.4 Sensor Data Flow . . . 11
4.4 Technology Selection Rationale . . . 11

5 Widget Tree **12**
5.1 Main Application Structure . . . 13
5.1.1 Main Widget (main.dart) . . . 13
5.2 Authentication Screens . . . 14
5.2.1 Login Screen (lib/screens/login_screen.dart) . . . 14
5.2.2 Signup Screen (lib/screens/signup_screen.dart) . . . 15
5.3 Core Application Screens . . . 15
5.3.1 Home Screen (lib/screens/home_screen.dart) . . . 15
5.3.2 Events Screen (lib/screens/events_screen.dart) . . . 16
5.3.3 Discover Screen (lib/screens/users_search_screen.dart) . . . 18
5.3.4 Event Detail Screen (lib/screens/event_detail_screen.dart) . . . 20
5.3.5 Profile Screen (lib/screens/profile_screen.dart) . . . 23
5.3.6 Public Profile Screen (lib/screens/public_profile_screen.dart) . . . 26
5.4 Social & Sharing Screens . . . 27

5.4.1 Private Share Screen (lib/screens/private_share_screen.dart) 27
5.4.2 Conversation Screen (lib/screens/conversation_screen.dart) 29
5.4.3 QR Scan Screen (lib/screens/qr_scan_screen.dart) 29
5.5 Statistics Screens 29
5.5.1 Stats Screen (lib/screens/stats_screen.dart) 29
5.6 Reusable Widgets 32
5.6.1 Event Card (lib/widgets/event_card.dart) 32
5.6.2 Custom Button (lib/widgets/custom_button.dart) 32

6 Services and Providers 32
6.1 State Management Providers 32
6.1.1 Events Provider (lib/providers/events_provider.dart) 32
6.1.2 Local Events Provider (lib/providers/local_events_provider.dart) . . 32
6.2 Authentication Service (lib/services/auth_service.dart) 33
6.3 API Services 33
6.3.1 Event API Service (lib/services/event_api_service.dart) 33
6.3.2 Users Service (lib/services/users_service.dart) 33
6.3.3 Storage Service (lib/services/storage_service.dart) 33
6.3.4 Leaderboard Service (lib/services/leaderboard_service.dart) 33
6.4 Database Service (lib/services/local_db.dart) 34
6.5 Sync Services 34
6.5.1 User Sync Service (lib/services/user_sync_service.dart) 34
6.5.2 Sync Service (lib/services/sync_service.dart) 34
6.6 Sensor Services 34
6.6.1 Motion Sensor Service (lib/services/motion_sensor_service.dart) . . 34
6.6.2 Heart Rate Service (lib/services/heart_rate_service.dart) 35
6.7 Communication Services 35
6.7.1 Bluetooth Share Service (lib/services/bluetooth_share_service.dart) 35
6.7.2 Private Messaging Service (lib/services/private_messaging_service.dart) 35
6.7.3 Notification Service (lib/services/notification_service.dart) 35

7 Backend Architecture 36
7.1 Flask Application Structure 36
7.2 Database Models 37
7.2.1 User Model (backend/models/user.py) 37
7.2.2 Event Model (backend/models/event.py) 37
7.2.3 Conversation Model (backend/models/conversation.py) 38
7.2.4 Private Message Model (backend/models/private_message.py) . . . 38
7.2.5 Follow Model (backend/models/follow.py) 39
7.2.6 Notification Model (backend/models/notification.py) 39
7.2.7 User Event Stats Model (backend/models/user_event_stats.py) . . . 39
7.3 API Routes 40
7.3.1 User Routes (backend/routes/users.py) 40
7.3.2 Event Routes (backend/routes/events.py) 40
7.3.3 Private Message Routes (backend/routes/private_messages.py) . . . 40
7.3.4 Storage Routes (backend/routes/storage.py) 41
7.4 Services 41

7.4.1 Badge Service (backend/services/badges.py) 41
7.4.2 Firebase Auth Service (backend/services/firebase_auth.py) 41
7.4.3 Location Service (backend/services/location.py) 41
7.4.4 Stats Service (backend/services/stats.py) 41

8 Dependencies 41
8.1 Flutter Dependencies (pubspec.yaml) 41
8.1.1 Core Dependencies 41
8.1.2 Authentication & Backend 42
8.1.3 Database 42
8.1.4 Connectivity & Sharing 42
8.1.5 Location & Maps 42
8.1.6 Media & Sensors 42
8.1.7 UI & Visualization 43
8.2 Backend Dependencies (requirements.txt) 43

9 Development Challenges and Solutions 43
9.1 Challenge 1: Offline Data Synchronization 43
9.2 Challenge 2: Firebase Auth Integration with Custom Backend 44
9.3 Challenge 3: Peer-to-Peer Connectivity 44
9.4 Challenge 4: Database Connection Pooling 44
9.5 Challenge 5: Media Upload to Cloud Storage 45
9.6 Challenge 6: Real-Time Sensor Data Handling 45
9.7 Challenge 7: Bluetooth File Recovery (Deep Search) 45
9.8 Challenge 8: UI Responsiveness & Optimistic Updates 45
9.9 Challenge 9: Foreground Notifications 46

10 Project Diagrams 46

11 Tutorial and Usage Scenarios 49
11.1 Scenario 1: First-Time User Registration 49
11.2 Scenario 2: Discovering and Joining Events 49
11.3 Scenario 3: Capturing Concert Memories 49
11.4 Scenario 4: Monitoring Sensor Data 50
11.5 Scenario 5: Data Synchronization 50
11.6 Scenario 6: Viewing Statistics 50
11.7 Scenario 7: Social Features 51
11.8 Scenario 8: Creating a New Event 51
11.9 Scenario 9: Event Participation and Live Statistics 51
11.10 Scenario 10: Media Sharing with Privacy Options 52
11.11 Scenario 11: Private Messaging with Online/Offline Options 52
11.12 Scenario 12: Leaderboard and Badge System 52
11.13 Scenario 13: Location Sharing and Map Discovery 53
11.14 Scenario 14: QR Code Sharing and Privacy Settings 53
11.15 Scenario 15: Notification Center 53

12 Improvements since the previous Milestone 54

13 Overall Assessment **54**
13.1 Key Achievements 55
13.2 Technical Learning Outcomes 55
13.3 Areas for Future Enhancement 55

14 Contribution Assessment **56**
14.1 Carolina Silva (Nº 113475) - 50% 56
14.2 Luana Reis (Nº 131193) - 50% 57
14.3 Collaborative Work 58

15 Conclusion **58**

Repository Information

The complete source code and project files for FanZone are available on GitHub:

`https://github.com/luanacarolinareis/FanZone`

This repository includes:

- Flutter mobile application source code
- Flask backend implementation
- Docker configuration files
- Project documentation and setup instructions
- Database schemas and migration scripts

Additionally, the Android application is available for download and testing at the following website:

`https://www.fanzone-app.tech/`

The backend infrastructure is deployed and running on a dedicated Virtual Machine using Docker.

1 Introduction

Mobile applications have become an essential part of everyday life, supporting activities ranging from communication and entertainment to productivity and personal organization. In the context of mobile computing, modern applications are expected to be responsive, reliable, and capable of operating under varying network conditions, particularly in environments with limited connectivity.

Within the scope of the *Computação Móvel* course, this project proposes the development of **FanZone**, a fully functional mobile application built with the Flutter framework. FanZone is designed specifically for music fans who attend live concerts and wish to capture, organize, and relive their experiences in a structured and meaningful way.

Although users frequently record photos and videos during concerts, these memories are often scattered across personal galleries or lost over time. Additionally, connections formed during events tend to fade after the concert ends, and there is no dedicated platform focused on preserving both memories and social interactions associated with live music events.

FanZone addresses these challenges by offering an **offline-first mobile solution** that allows users to capture media, record contextual data through sensors, and synchronize information seamlessly when network connectivity becomes available. The application combines media sharing, social interaction, event tracking, and real-time sensor integration into a single cohesive experience tailored for concert environments.

2 Project Context

FanZone is an offline-first Flutter mobile application designed for music fans attending concerts. The application enables users to:

- Capture and organize concert memories in one dedicated space
- Record photos, videos, and audio from events
- Connect with other attendees and build lasting connections
- Track their concert journey with detailed statistics
- Access all features even without internet connectivity
- Sync data seamlessly when back online

The app targets concert enthusiasts aged 18-35 who attend multiple concerts per year, value experiences over possessions, and want to preserve memories digitally while staying connected with fellow fans.

2.1 The Problem

Concert venues often suffer from poor network coverage, with thousands of users competing for limited bandwidth. Fans take hundreds of photos and videos at concerts, but this media gets lost in camera rolls. People meet amazing individuals at concerts but connections fade after exchanging contact information. Furthermore, there's no way to track concert attendance or visualize one's music fan journey.

2.2 Our Solution

FanZone provides a comprehensive platform that works offline-first, ensuring reliability even in challenging network conditions. The app combines media capture, social networking, statistical tracking, and sensor integration to create an immersive concert experience that extends beyond the event itself.

3 Requirements

Our team defined the following requirements to make the system fully functional:

3.1 Authentication and User Management

- Email/password authentication using Firebase Authentication
- Persistent user sessions across app restarts
- Profile management with display names and photos
- User-friendly error handling with clear messages

3.2 Offline-First Architecture

- Local SQLite database as primary data source
- Some functionality without internet connection
- Automatic synchronization when connectivity is restored
- Network monitoring and adaptive behavior
- No data loss during network outages

3.3 Event Management

- Browse upcoming and past events
- Create new local events with comprehensive details
- View event information
- Join/leave events to track participation

3.4 Media Capture and Sharing

- Upload and share media with event community
- View gallery of media from other attendees
- Cloud storage integration via Supabase

3.5 Sensor Integration

- Accelerometer data capture for movement detection
- Ambient light sensor integration

- Sound level monitoring
- Heart rate data from WearOS devices
- Real-time sensor data visualization

3.6 Social Features

- User search functionality
- Follow/unfollow system
- Mutual follow creates friend relationship automatically
- Public profile viewing
- Follower and following lists
- Private sharing via Bluetooth/WiFi with method selection

3.7 Notifications

- Notifications for important updates via Firebase Cloud Messaging
- Foreground heads-up notifications using local notifications

3.8 Statistics and Analytics

- Track total events attended
- Display follower/following statistics
- Visual charts and graphs using fl_chart
- Concert history (movement, light, sound, heart rate)

3.9 Location Features

- Google Maps integration
- Users' location mapping
- Filtering by users' name
- Share location with all users or with friends

4 Architecture Diagram and Workflow

The architecture of FanZone comprises several essential components working together to provide a seamless user experience:

4.1 System Components

1. **Flutter Frontend**: Cross-platform mobile application built with Flutter/Dart
2. **Firebase Authentication**: Secure user authentication and identity management
3. **Flask Backend**: Custom REST API for business logic and data operations
4. **PostgreSQL Database**: Supabase-hosted relational database for remote storage
5. **SQLite Database**: Local on-device database for offline functionality
6. **Supabase Storage**: Cloud storage for media files
7. **WearOS Integration**: Heart rate monitoring from smartwatches

Figure 1: Key Technologies - Flutter, Firebase, Flask, WearOS, PostgreSQL, SQLite, and Supabase

4.2 Architectural Principles

Offline-First Approach: The application uses local SQLite as the primary data source, with the backend serving as a synchronization target. This ensures some functionality without network dependency and provides complete control over the data model.

Manual Synchronization: Unlike services like Firebase Firestore that handle automatic state synchronization, FanZone implements custom synchronization logic. This decision provides full control over when and how data syncs, optimizing for concert venue scenarios where connectivity is intermittent.

Dual Database Strategy:

- Local: SQLite database on the device for immediate access and offline operation
- Remote: PostgreSQL on Supabase for data persistence and cross-device synchronization
- Sync: Manual synchronization via REST API when connectivity is available

4.3 Workflow

The workflow embedded within our architecture unfolds as follows:

4.3.1 Authentication Flow

1. User registers or logs in through Firebase Authentication
2. Firebase returns a JWT (ID token)
3. Flutter app includes this token in Authorization headers for all API requests
4. Flask backend verifies the token using Firebase Admin SDK
5. Upon successful verification, backend executes the requested operation

4.3.2 Data Synchronization Flow

1. User performs actions offline (create events, capture media, etc.)
2. Data is stored immediately in local SQLite database
3. App monitors network connectivity continuously
4. When connection is detected, synchronization service activates
5. Local changes are pushed to backend via REST API
6. Backend validates and stores data in PostgreSQL
7. Remote changes are pulled and merged with local database
8. Conflict resolution ensures data integrity

4.3.3 Media Upload Flow

The media upload process uses a proxy pattern to maintain security by keeping Supabase service keys on the backend while allowing the mobile app to upload media securely.

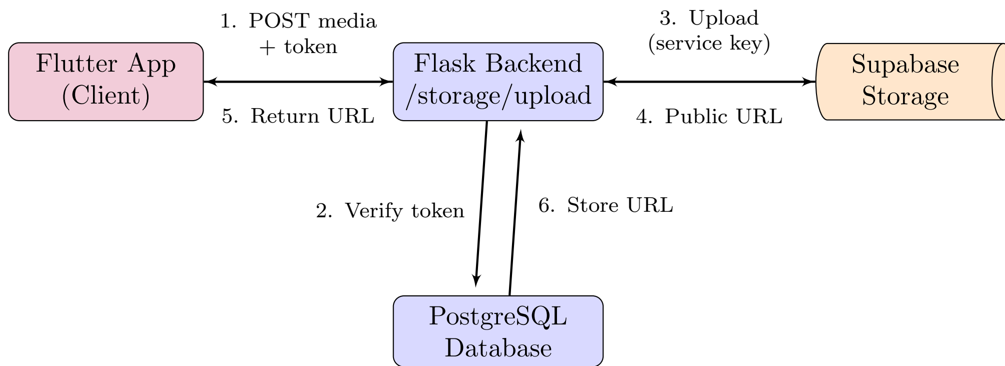

Figure 2: Media upload flow showing secure proxy pattern with Firebase authentication and Supabase Storage integration

The detailed steps are:

1. User uploads photo/video through the app
2. Media is stored locally for immediate access
3. When online, media is uploaded to Supabase Storage via backend proxy
4. Backend returns public URL for the media
5. URL is stored in both local and remote databases
6. Other users can fetch and view the media

4.3.4 Sensor Data Flow

1. Sensor streams (accelerometer, etc.) are initialized
2. High-frequency data is throttled to prevent UI performance issues
3. Aggregated sensor data is stored periodically
4. WearOS devices transmit heart rate data
5. All sensor data is visualized in real-time
6. Data is associated with specific events and stored locally

4.4 Technology Selection Rationale

Flutter was chosen for its cross-platform capabilities, allowing development for both iOS and Android from a single codebase. Its rich widget library and hot-reload feature significantly accelerated development.

Firebase Authentication provides secure, scalable authentication without the complexity of implementing our own system. It handles token generation and security best practices.

Flask Backend was selected for its lightweight nature and flexibility. It provides full control over business logic and data validation while remaining easy to deploy and maintain.

PostgreSQL/Supabase offers a reliable relational database with excellent performance. Supabase's Session Pooler handles connection management efficiently, crucial for mobile applications with many concurrent users.

SQLite is the ideal choice for local storage: it's fast, lightweight, requires no server, and is perfectly suited for mobile devices.

5 Widget Tree

The FanZone application is organized into a structured widget hierarchy that separates concerns and promotes code reusability. The widget tree is divided into several major sections based on functionality.

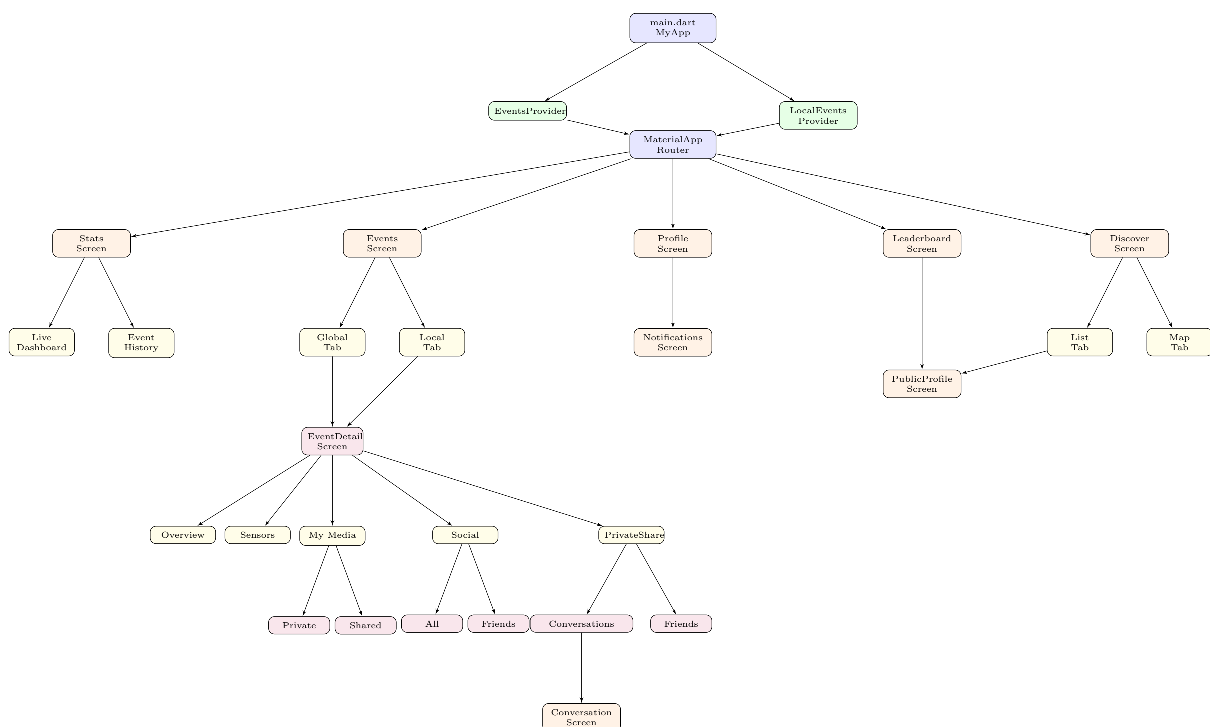

Figure 3: Widget tree showing tab-based navigation structure with nested views

5.1 Main Application Structure

5.1.1 Main Widget (main.dart)

The root widget initializes the Flutter application and sets up global providers for state management. It configures Firebase, Supabase, and initializes the local database and sensor services.

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp(
    options: DefaultFirebaseOptions.currentPlatform,
  );
  await HeartRateService.instance.init();
  await Supabase.initialize(url: supabaseUrl, anonKey: supabaseAnonKey);
  await LocalDb.database;

  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (_) => MotionSensorService()),
        ChangeNotifierProvider(create: (_) => EventsProvider()..
  loadEvents()),
        ChangeNotifierProxyProvider<EventsProvider, LocalEventsProvider
  >(
          create: (context) => LocalEventsProvider(eventsProvider:
  context.read<EventsProvider>()),
          update: (context, events, previous) {
            final p = LocalEventsProvider(eventsProvider: events);
            p.initIfNeeded();
            return p;
          },
        ),
      ],
      child: const FanZoneApp(),
    ),
  );
}
```

Listing 1: Main Widget Structure

5.2 Authentication Screens

5.2.1 Login Screen (lib/screens/login_screen.dart)

Provides email/password authentication with:

- Text input fields with validation
- Password visibility toggle
- Loading indicator during authentication
- Navigation to signup screen
- Persistent Login - Credentials are saved securely, allowing users to bypass the login screen on subsequent app launches.

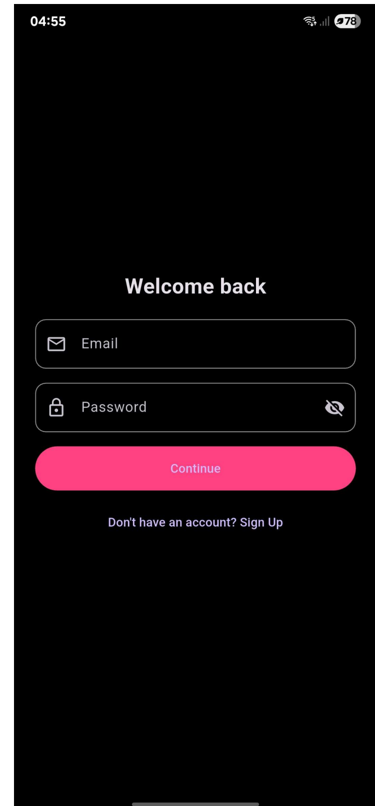

Figure 4: Login Screen

5.2.2 Signup Screen (lib/screens/signup_screen.dart)

Collects user information for registration:

- First name and last name fields
- Email and password inputs with validation
- Creates user profile with combined display name

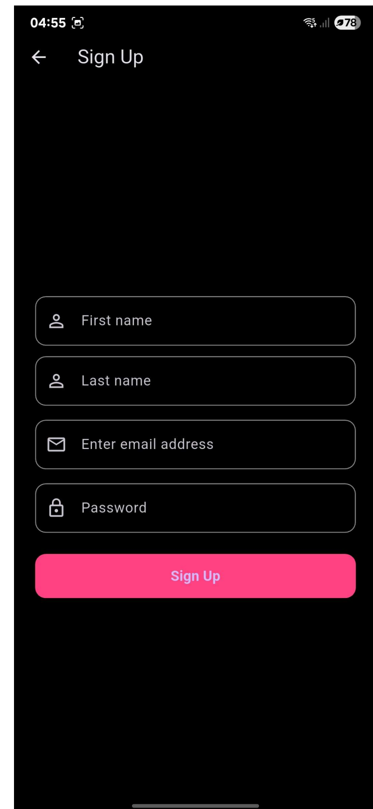

Figure 5: Signup Screen

5.3 Core Application Screens

5.3.1 Home Screen (lib/screens/home_screen.dart)

The main dashboard displaying the bottom navigation bar with five main sections:

- **Events**: Global event feed and user-created local events
- **Stats**: Personal analytics and concert history
- **Leaderboard**: Global user rankings based on event participations
- **Discover**: User search, list and map
- **Profile**: User settings, notifications and history

5.3.2 Events Screen (lib/screens/events_screen.dart)

The central hub for event discovery and management, organized into two distinct tabs to separate major official concerts from community gatherings.

Global Tab:

- Displays Global Events: Pre-defined by application administrators for major concerts and festivals. This ensures critical event information is accurate and prevents misinformation that could impact large audiences.
- List view of official events with key information cards (Title, Description, Date, Image).
- Search functionality by name or location.
- Filter by active/inactive status.

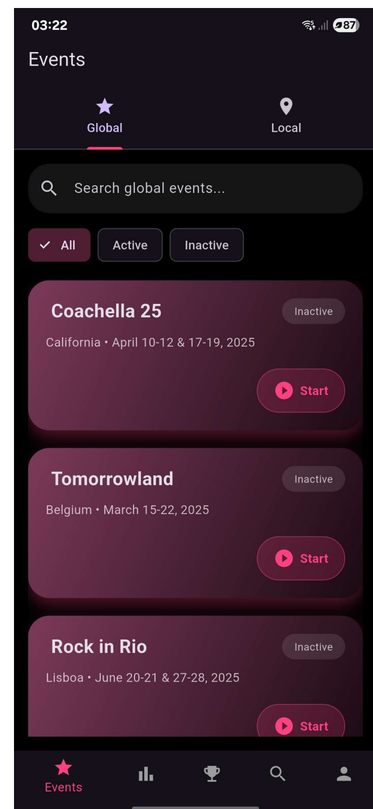

Figure 6: Global Events Tab showing official concerts

Local Tab:

- Displays Local Events: Can be created by administrators or any user. Focused on smaller gatherings such as local parties, fan meetups, or community celebrations.
- Creation Options:
  - **Manual Creation**: Form input for title, description, date, and location.
  - **QR Code Scan**: Instantly create an event by scanning a pre-generated JSON QR code containing event details.
- Management:
  - **Edit/Delete**: Users can only edit or delete events they manually created themselves. Events created by others are read-only.

– **Database Cleanup**: A dedicated button allows users to clear local database changes that haven't been synced and force a resync from Supabase.

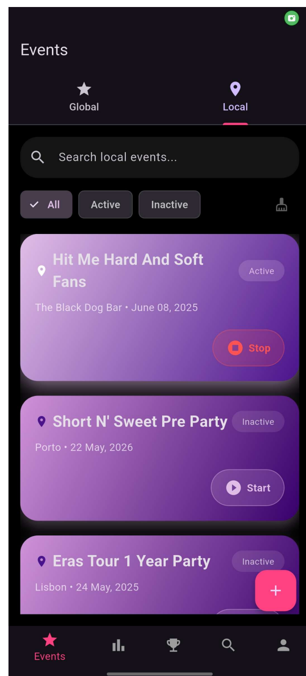

Figure 7: Local Events Tab

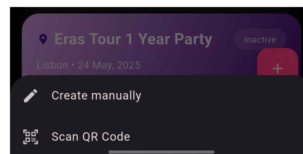

Figure 8: Event Creation Options

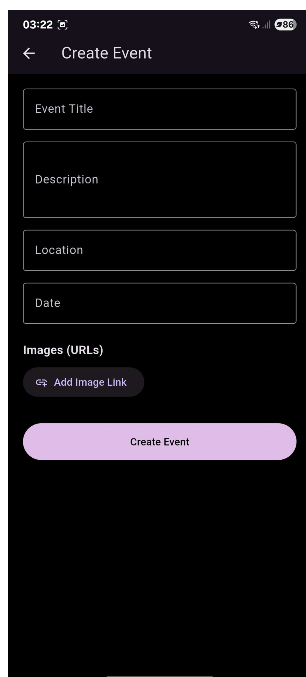

Figure 9: Manual Event Creation Form

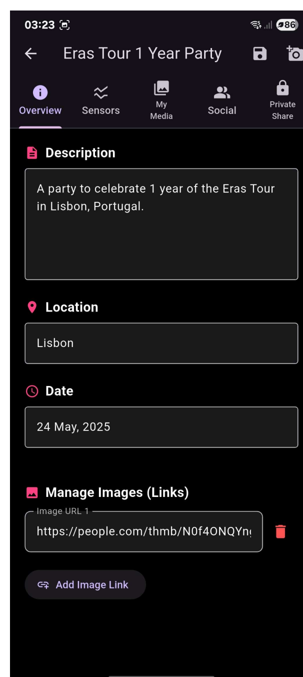

Figure 10: Edit Event Screen

Both event types support the same rich feature set (participation, sensors, media sharing, social features) once entered.

5.3.3 Discover Screen (lib/screens/users_search_screen.dart)

Comprehensive user discovery platform with two main tabs:

List Tab:

- Displays all users in the system with search functionality.
- Relationship Status: Each user card displays a button indicating the current relationship:
  - **Friends**: Mutual friendship established.
  - **Following**: User is following the target but not friends yet.
  - **Follow**: Available to follow.
- Clicking any user opens their detailed public profile.
- Refresh button to update the user list from the server.

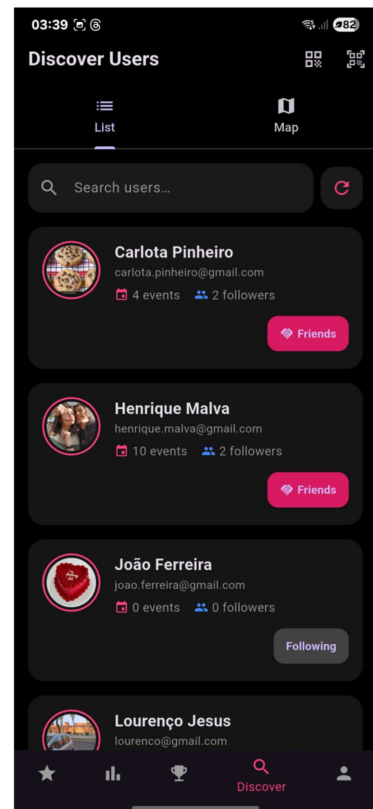

Figure 11: Discover Users List

Map Tab:

- Interactive Google Maps interface showing user locations.
- Location Sharing: Toggle switch allows users to enable/disable sharing their own location with others (Friends or All, based on privacy settings).
- User Pins: Users sharing their location appear as pins featuring their profile picture. Clicking a pin reveals their name and email.
- Search & Zoom: Searching for a user highlights their location and automatically zooms the map to their position (if they are sharing location).
- Refresh button to update location data from all users.

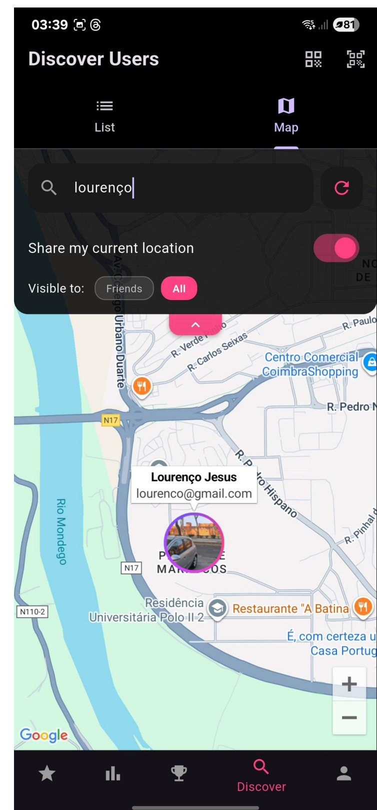

Figure 12: User Discovery Map with Profile Pins

QR Code Features:

- My QR Code: Displays the user's personal QR code for others to scan.
- Scan QR: Allows scanning another user's QR code to instantly follow them.

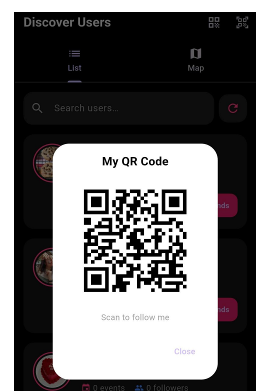

Figure 13: Personal QR Code for Quick Connection

5.3.4 Event Detail Screen (lib/screens/event_detail_screen.dart)

Displays comprehensive event information with multiple tabs and event participation controls:

Event Participation:

- Start/Stop Participation: Users can start and stop participating in an event.
- Live Statistics: While participating, users gain access to the Live Dashboard (in Stats Screen) showing real-time graphs for movement, ambient light, sound level, and heart rate.
- These metrics are recorded to calculate leaderboard scores and award badges.

Event Tabs:

- Overview: Event title, date, location, description, and event images (expandable to full screen). Future enhancements may include a rules section and a text forum.
- Sensors: Displays the history of sensor data for this specific event. This tab becomes available only after the user stops participating and if sensor records exist.
- My Media: Personal media diary with two subdivisions:
  - *Private*: Personal photos and videos. Users can long-press to delete items here.
  - *Shared*: History of media shared with others. Deletion is not permitted to preserve the sharing record. Each item includes an icon and description indicating its scope (Shared with All or Friends).
- Social: Media shared within the event with two subdivisions:
  - *All*: Media shared by everyone in the event (including the user's own public shares). Items display the sharer's profile picture, name, and scope icon.
  - *Friends*: Media shared by friends (and the user's own shares to friends).
- Private Share: detailed in Private Share section.

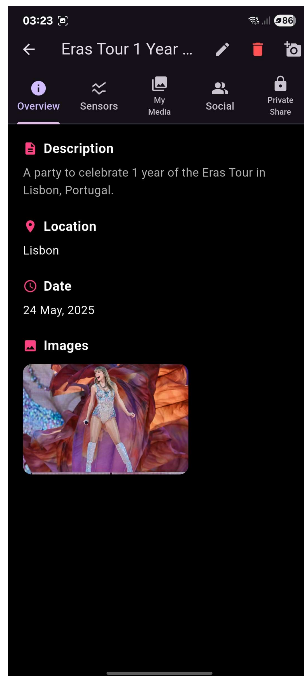

Figure 14: Event Overview Tab

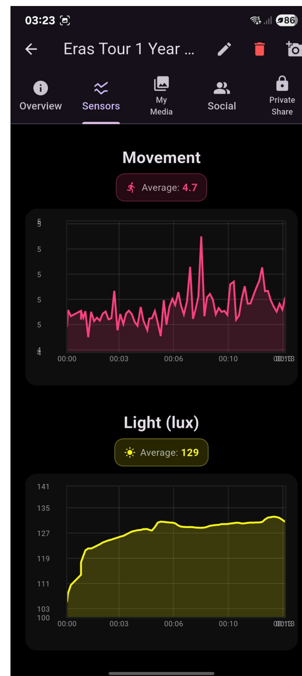

Figure 15: Event Sensor History

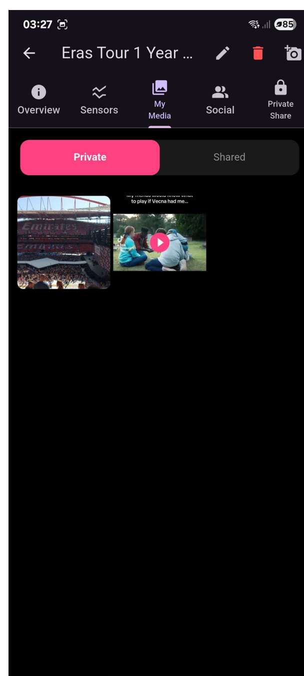

Figure 16: My Media (Private)

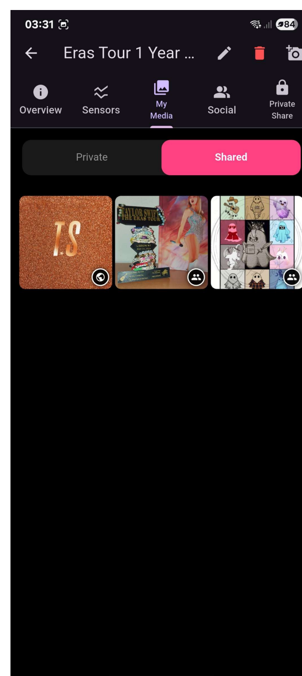

Figure 17: My Media (Shared History)

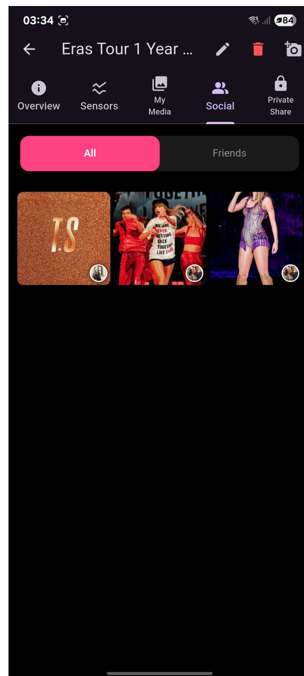

Figure 18: Social Media (All Participants)

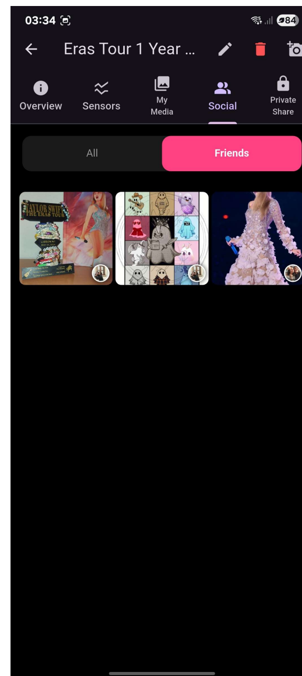

Figure 19: Social Media (Friends Only)

Media Upload:

- Camera button in top-right corner provides three sharing options:
  - Share with all
  - Share with friends
  - Don't share with anyone (private only)
- Supports photos and videos up to 10MB.
- Viewing: Photos open in full-screen. Videos generate thumbnails and include a player with pause and seek controls.
- Media is automatically organized into the appropriate tab based on the chosen option.

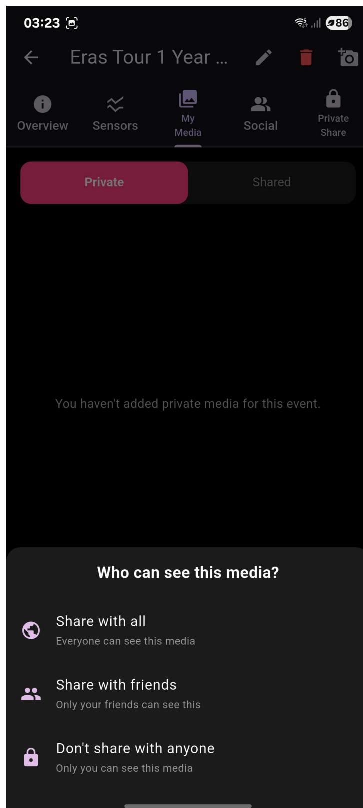

Figure 20: Upload Options

Figure 21: Fullscreen Photo

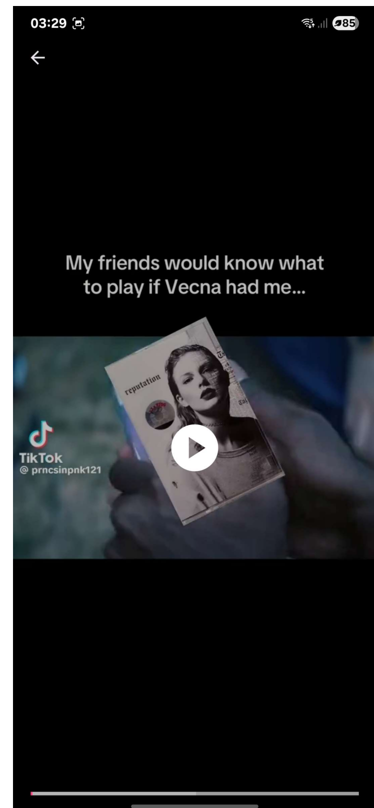

Figure 22: Video Player

5.3.5 Profile Screen (lib/screens/profile_screen.dart)

Comprehensive user profile and settings management:

Profile Information:

- Profile Picture: Editable with photo upload capability.
- Display Name: Editable first and last name.
- Email: Display of account email.
- Social Connections: Clickable counters for Followers, Following, and Friends.

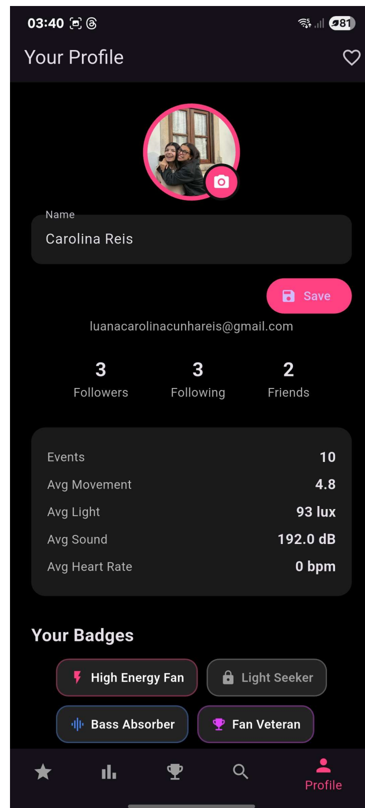

Figure 23: Profile Overview

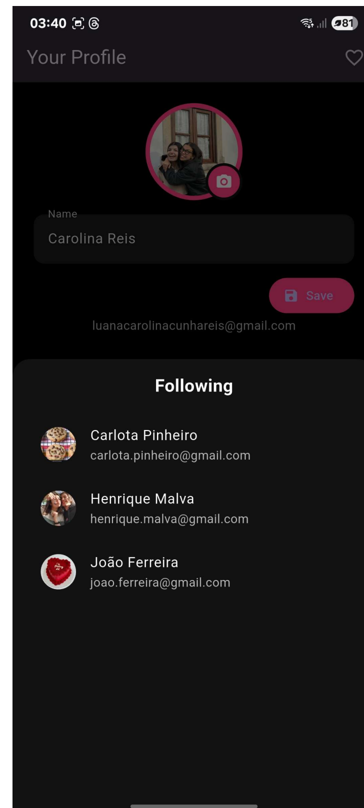

Figure 24: Following List

Badge Display:

- Visual display of all badges (locked and unlocked).
- Clicking each badge shows requirements and current status.
- Unlocked badges contribute to leaderboard score.

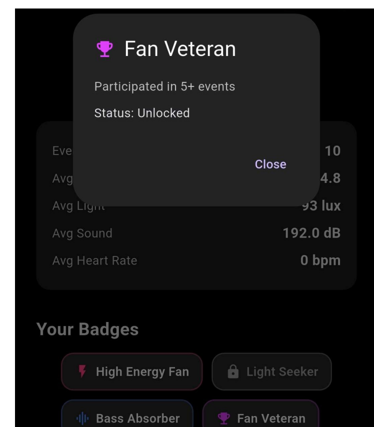

Figure 25: Badge Details

Privacy Settings:

- Control visibility (All, Friends, None) for:
  - Profile Picture
  - Email
  - Following/Followers Lists

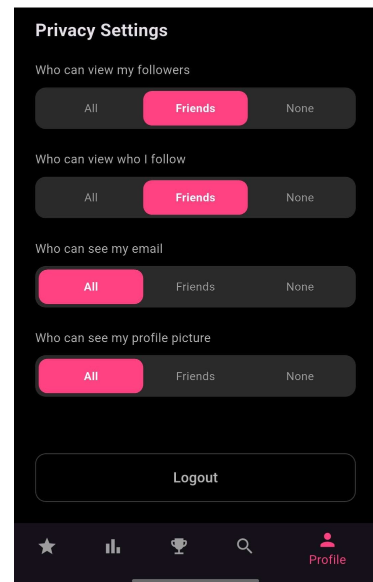

Figure 26: Privacy Settings

Notifications:

- Heart Icon: Opens notification center.
- Triggers: Leaderboard changes, friends nearby (500m), location sharing starts, new followers, badges unlocked.
- Notifications are delivered as local push notifications when the app is in the background or foreground (and in the form of a list).

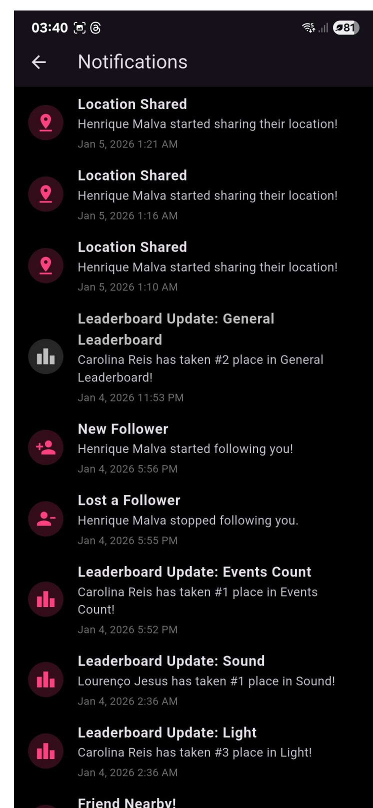

Figure 27: Notification Center

Account Management:

- Logout button.
- Persistent login saves credentials after successful login.

5.3.6 Public Profile Screen (lib/screens/public_profile_screen.dart)

View other users' profiles with respect for privacy settings:

Visible Information (respecting privacy settings):

- Profile Picture: Replaced with placeholder if private.
- Name: Always visible.
- Email: Shows "Hidden" if private.
- Social Lists: Shows "Information is private" if restricted.
- Statistics & Badges: Summary of activity and achievements.

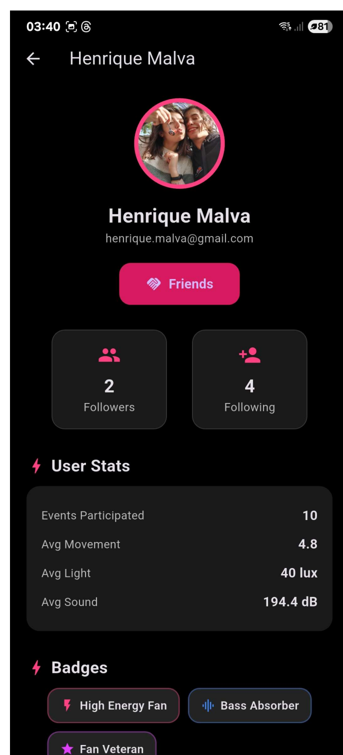

Figure 28: Public Profile View

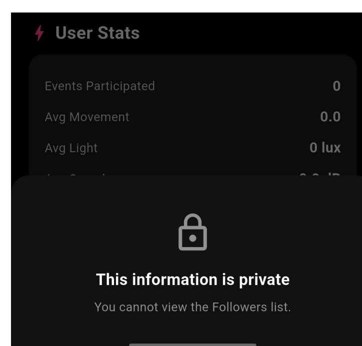

Figure 29: Restricted Information View

Interaction Options:

- Follow/Unfollow button.
- View detailed followers/following information.

5.4 Social & Sharing Screens

5.4.1 Private Share Screen (lib/screens/private_share_screen.dart)

Facilitates offline and private content sharing with two main tabs:

Conversations Tab:

- Displays all active private conversations.
- Shows conversation history with sent photos.
- Message Status: Indicators for "Sent" and "Seen" (with timestamp).
- Currently supports photo sharing only.

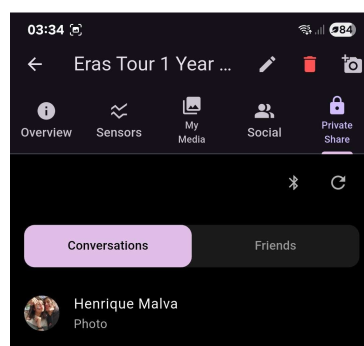

Figure 30: Conversations List

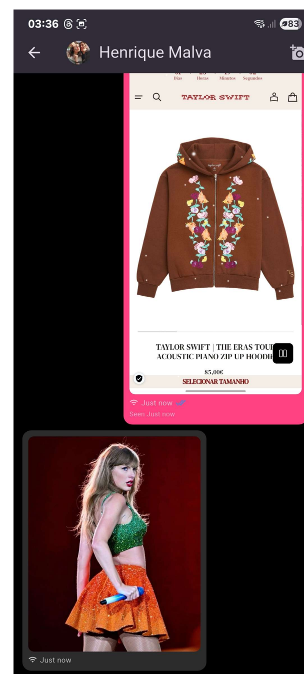

Figure 31: Chat Interface

Friends Tab:

- Lists all friends for initiating new conversations.
- Sending Process: Click "Send" on a friend - Select photo - Choose method.
- Send Online (WiFi): Fully functional. Uploads to Supabase, creates conversation, and syncs with recipient.
- Send Offline (Bluetooth): Requires both users to enable advertising/discovery via the Bluetooth icon. File transfer succeeds, but the recipient currently sees a broken filepath in the conversation view (known limitation to be resolved).

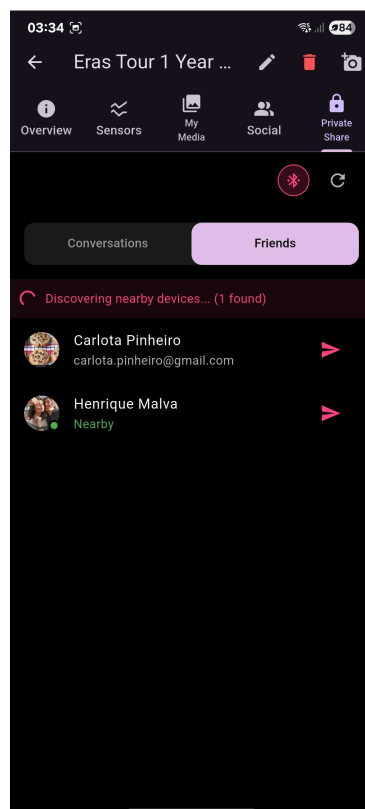

Figure 32: Friends List for Sharing

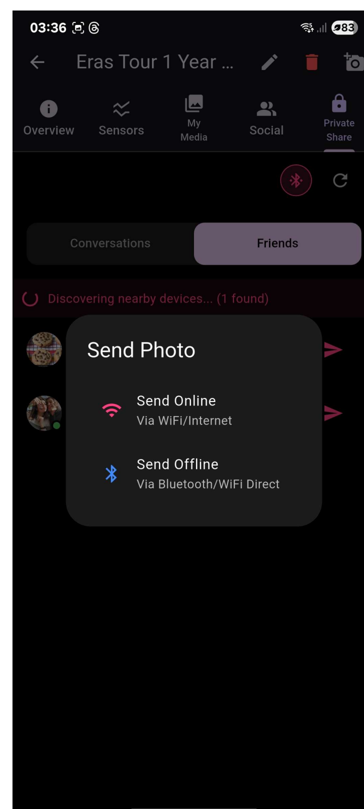

Figure 33: Sending Options (Online/Offline)

Functionality:

- Real-time message delivery status tracking
- View timestamp when message is seen by recipient
- Persistent conversation history stored locally
- Seamless integration with friend list for quick sharing

5.4.2 Conversation Screen (lib/screens/conversation_screen.dart)

Handles private messaging between users:

- Real-time chat interface
- Image sharing support
- Message history stored locally

5.4.3 QR Scan Screen (lib/screens/qr_scan_screen.dart)

Allows users to quickly join events or connect with others:

- Camera interface for scanning QR codes
- Automatic parsing of event data
- Immediate action execution (e.g., joining an event)

5.5 Statistics Screens

5.5.1 Stats Screen (lib/screens/stats_screen.dart)

Comprehensive analytics platform with two main sections:

Live Dashboard:

- Availability: Only accessible when actively participating in an event.
- Real-time Metrics: Displays live graphs for:
  - Movement (Accelerometer)
  - Ambient Light (Camera)
  - Sound Level (Microphone)
  - Heart Rate (if connected to WearOS)

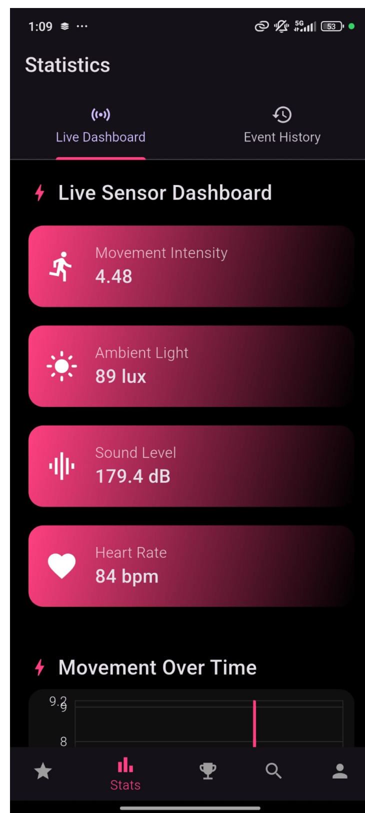

Figure 34: Live Dashboard

Figure 35: Real-time Sensor Graphs

Event History:

- Availability: Accessible when not participating in an event.
- Displays a list of all past concerts attended.
- Shows personal statistics for each past event.
- Search functionality to find specific past events.

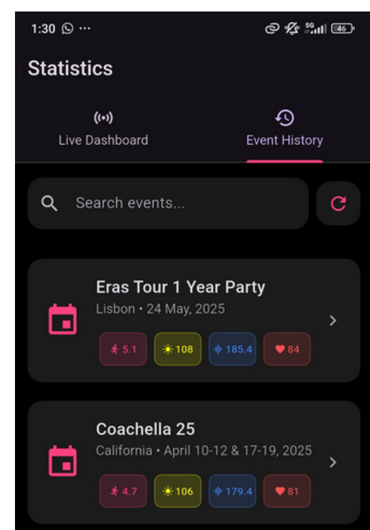

Figure 36: Event History Timeline

Leaderboard:

- Global ranking system based on participation scores.
- Podium: Top 3 users prominently featured. Can be expanded/collapsed.
- Full Ranking: List of all users with their scores.
- Filters: Filter rankings by category (All, Events, Movement, Light, Sound, Heart, Badges).
- User Details: Clicking a user opens their public profile.

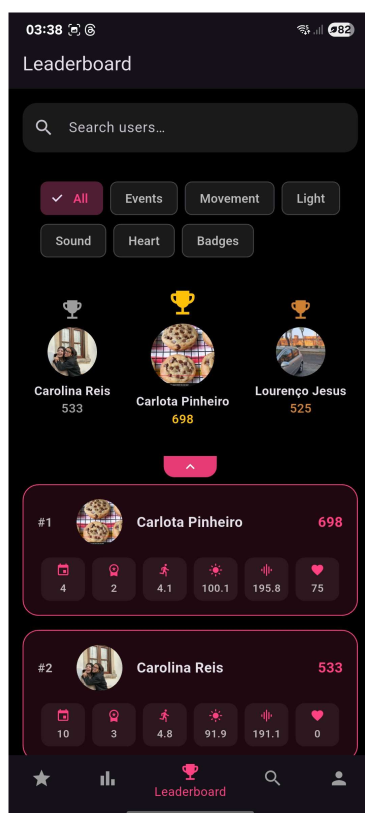

Figure 37: Leaderboard Podium

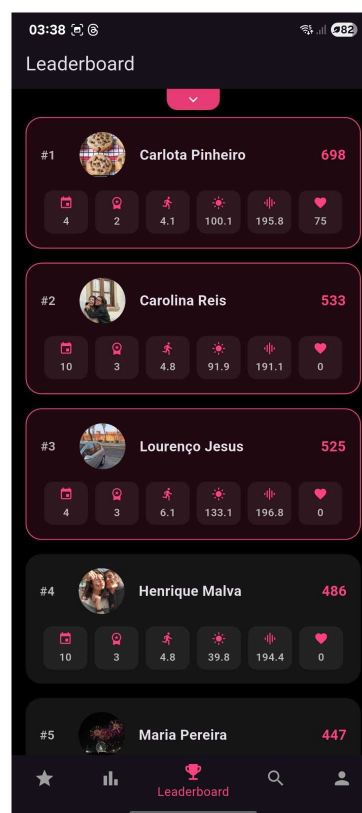

Figure 38: Ranking List

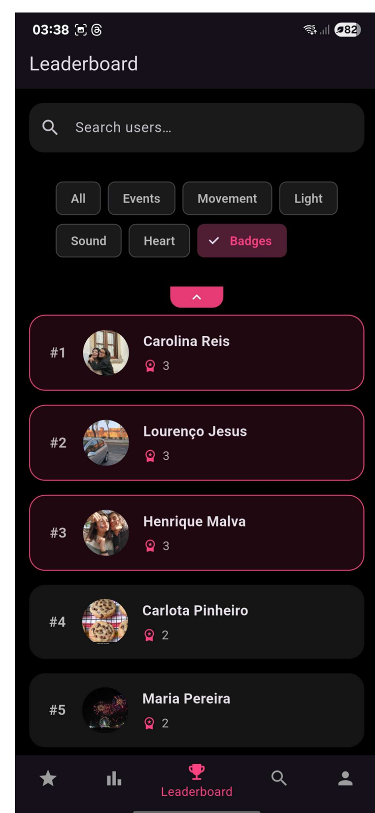

Figure 39: Badges View

Badge System:

- Achievement system rewarding various milestones
- Badges contribute to leaderboard score
- Visible in user profiles (both own and others)
- Each badge shows unlock criteria and current status (locked/unlocked)

5.6 Reusable Widgets

5.6.1 Event Card (lib/widgets/event_card.dart)

Displays event preview information in a card format with title, location, and date.

5.6.2 Custom Button (lib/widgets/custom_button.dart)

A customizable button widget used throughout the application for consistent styling.

6 Services and Providers

6.1 State Management Providers

The application uses the Provider pattern for state management, separating business logic from UI components.

6.1.1 Events Provider (lib/providers/events_provider.dart)

Centralizes the management of all event data:

- Loads events from the local SQLite database via `DatabaseHelper`
- Manages the currently active event state
- Filters events by source (global vs. local)
- Orchestrates synchronization by calling `SyncService` to push pending changes and pull updates

6.1.2 Local Events Provider (lib/providers/local_events_provider.dart)

A proxy provider that depends on `EventsProvider` to manage location-specific logic:

- Filters and manages events created locally on the device
- Handles the creation of new local events

6.2 Authentication Service (lib/services/auth_service.dart)

Manages all Firebase Authentication operations:

- `signUp`: Creates new user account and registers them in the backend
- `signIn`: Authenticates existing user
- `signOut`: Logs out current user
- `authStateChanges`: Stream of authentication state
- `getCurrentUser`: Returns current Firebase user

6.3 API Services

6.3.1 Event API Service (lib/services/event_api_service.dart)

Handles all HTTP communication with the backend related to events:

- `startEvent`: Creates a new event on the backend
- Manages authentication headers for requests

6.3.2 Users Service (lib/services/users_service.dart)

Manages user-related API calls:

- `searchUsers`: Searches for users by query
- `listUsers`: Fetches a list of users for discovery

6.3.3 Storage Service (lib/services/storage_service.dart)

Handles file uploads to Supabase Storage via the backend:

- `uploadEventMedia`: Uploads event photos/videos with progress tracking

6.3.4 Leaderboard Service (lib/services/leaderboard_service.dart)

Fetches leaderboard data and user statistics from the backend.

6.4 Database Service (lib/services/local_db.dart)

Handles specific local SQLite database operations related to user synchronization:

- `insertUser`: Stores user data locally
- `queryPending`: Retrieves users waiting to be synced
- `updateStatus`: Updates the sync status of users

Note: Core event and sensor data storage is handled by `DatabaseHelper` *in the* `database` *package.*

6.5 Sync Services

6.5.1 User Sync Service (lib/services/user_sync_service.dart)

Manages synchronization of user profiles with the backend:

- Monitors network connectivity
- Syncs "pending" locally created users to the backend when online

6.5.2 Sync Service (lib/services/sync_service.dart)

General synchronization logic for events:

- `pushPending`: Pushes locally created events to the backend
- Handles creation of remote events for local-only events

6.6 Sensor Services

6.6.1 Motion Sensor Service (lib/services/motion_sensor_service.dart)

A comprehensive service that manages multiple device sensors:

- Accelerometer: Captures movement intensity (X, Y, Z axes)
- Light Sensor: Uses the camera to detect ambient light levels
- Sound: Uses the microphone to measure ambient noise levels (dB)

6.6.2 Heart Rate Service (lib/services/heart_rate_service.dart)

Manages connection to WearOS devices:

- Configures Wearable API
- Listens for heart rate data messages from the watch

6.7 Communication Services

6.7.1 Bluetooth Share Service (lib/services/bluetooth_share_service.dart)

Manages peer-to-peer connections using the `nearby_connections` package:

- Advertising device presence to nearby users
- Discovering other devices
- Establishing secure connections
- Transferring files (photos) directly between devices
- Deep Search: A recovery mechanism to find received files even if metadata transfer fails

6.7.2 Private Messaging Service (lib/services/private_messaging_service.dart)

Handles the logic for private conversations:

- Storing messages in the local SQLite database
- Retrieving conversation history
- Managing message states (read/unread)
- Syncing messages with the backend

6.7.3 Notification Service (lib/services/notification_service.dart)

Manages system notifications:

- Initializes Firebase Messaging for push notifications
- Configures `flutter_local_notifications` for foreground alerts
- Manages Android notification channels (High Importance)
- Streams notification events to update UI automatically

7 Backend Architecture

The backend is organized into a modular Flask application with clear separation of concerns.

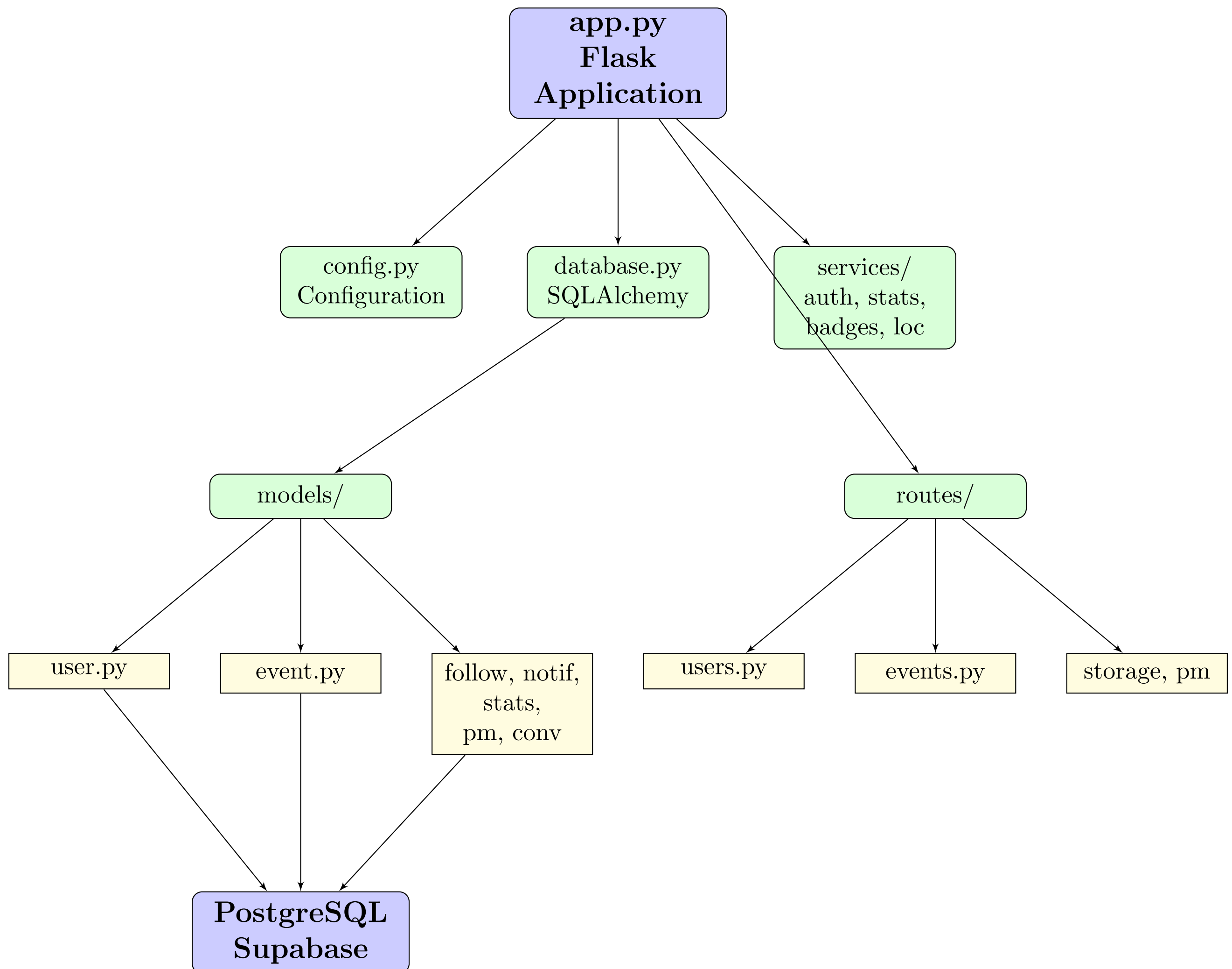

Figure 40: Backend architecture showing Flask application structure, models, routes, and database connection

7.1 Flask Application Structure

The backend structure is organized as follows:

```
backend/
|-- app.py                  # Main entry point
|-- config.py               # Configuration management
|-- database.py             # SQLAlchemy initialization
|-- models/                 # Database models
|   |-- user.py
```

```
|    |-- event.py
|    |-- private_message.py
|    |-- conversation.py
|    |-- follow.py
|    |-- notification.py
|    `-- user_event_stats.py
|-- routes/                # API endpoints
|    |-- users.py
|    |-- events.py
|    |-- private_messages.py
|    `-- storage.py
`-- services/              # Business logic
     |-- firebase_auth.py  # Token verification
     |-- badges.py         # Badge logic
     |-- location.py       # Location services
     `-- stats.py          # Statistics calculation
```

Listing 2: Backend Structure

7.2 Database Models

7.2.1 User Model (backend/models/user.py)

Represents user accounts with fields:

- `id`: Integer primary key
- `uid`: Firebase User ID (unique)
- `client_id`: Client-side generated UUID for idempotency
- `name`: Display name
- `email`: Email address
- `user_type`: Role ('client' or 'admin')
- `photo_url`: Profile picture URL
- `location_shared`: Boolean flag for location sharing
- `privacy_*`: Various privacy settings (followers, email, etc.)

7.2.2 Event Model (backend/models/event.py)

Stores concert event information:

- `id`: Auto-increment primary key
- `title`: Event name

- `description`: Event details
- `location`: Venue location
- `date`: Event date string (supports ranges)
- `organizer_uid`: Creator's Firebase UID
- `source`: Origin of the event ('global' or 'local')
- `media`: JSON fields for photos, videos, and social media links

7.2.3 Conversation Model (backend/models/conversation.py)

Manages chat sessions between two users:

- `id`: Primary key
- `user1_uid`, `user2_uid`: Participants
- `last_message_at`: Timestamp for sorting
- `unread_counts`: Tracks unread messages for each user

7.2.4 Private Message Model (backend/models/private_message.py)

Stores media messages exchanged between users:

- `id`: Primary key
- `conversation_id`: Foreign key to Conversation
- `sender_uid`, `recipient_uid`: Participants
- `media_url`: URL to the media in storage
- `media_type`: Type of media ('photo' or 'video')
- `transfer_method`: How it was sent ('online', 'bluetooth')
- `is_read`: Read status

7.2.5 Follow Model (backend/models/follow.py)

Manages user relationships:

- `follower_uid`: UID of the user following
- `target_uid`: UID of the user being followed
- `created_at`: Timestamp of the follow action

7.2.6 Notification Model (backend/models/notification.py)

Stores user notifications:

- `user_uid`: Recipient UID
- `type`: Notification type (e.g., 'follow', 'event_invite')
- `title`, `body`: Notification content
- `data`: JSON payload for navigation/actions
- `is_read`: Read status

7.2.7 User Event Stats Model (backend/models/user_event_stats.py)

Stores sensor data and stats for a user at an event:

- `user_uid`: User UID
- `event_id`: Event ID
- `movement_history`: Accelerometer data series
- `light_history`: Ambient light data series
- `sound_history`: Noise level data series
- `heart_rate_history`: Heart rate data (from Wear OS)
- `media`: List of media captured during the event

7.3 API Routes

7.3.1 User Routes (backend/routes/users.py)

- `POST /users/`: Create or update user profile
- `POST /users/register`: Idempotent registration (supports offline sync)
- `GET /users/search`: Search users by name or email
- `GET /users/list`: List users for discovery
- `GET /users/autocomplete`: Autocomplete user search
- `POST /users/follow`: Follow a user

7.3.2 Event Routes (backend/routes/events.py)

- `GET /events/`: List all events (with filters)
- `POST /events/start`: Create a new event
- `PUT /events/<id>`: Update an existing event
- `DELETE /events/<id>`: Delete an event
- `POST /events/stop/<id>`: Mark an event as finished

7.3.3 Private Message Routes (backend/routes/private_messages.py)

- `GET /private-messages/conversations`: Get all conversations
- `GET /private-messages/conversations/<id>/messages`: Get messages in a conversation
- `POST /private-messages/conversations/<id>/read`: Mark conversation as read
- `POST /private-messages/send`: Send a new message
- `POST /private-messages/sync`: Sync offline messages metadata
- `POST /private-messages/media`: Upload media for messages

7.3.4 Storage Routes (backend/routes/storage.py)

- `POST /upload/avatar`: Upload user avatar to Supabase Storage

7.4 Services

7.4.1 Badge Service (backend/services/badges.py)

Logic for awarding badges based on user statistics (e.g., "High Energy Fan", "Light Seeker").

7.4.2 Firebase Auth Service (backend/services/firebase_auth.py)

Handles Firebase Admin initialization and token verification, including a bypass mechanism for local development.

7.4.3 Location Service (backend/services/location.py)

Utilities for calculating distances between coordinates (Haversine formula).

7.4.4 Stats Service (backend/services/stats.py)

Aggregates user data from multiple events to calculate averages and generate profile statistics and leaderboards.

8 Dependencies

8.1 Flutter Dependencies (pubspec.yaml)

8.1.1 Core Dependencies

- `flutter`: SDK version ^3.9.2
- `provider`: State management
- `http`: HTTP client for API calls
- `uuid`: Unique identifier generation

- `permission_handler`: Runtime permission management
- `path_provider`: File system path access
- `device_info_plus`: Device information access
- `intl`: Internationalization and date formatting

8.1.2 Authentication & Backend

- `firebase_core`: Firebase initialization
- `firebase_auth`: Firebase Authentication
- `firebase_messaging`: Push notifications
- `supabase_flutter`: Supabase client

8.1.3 Database

- `sqflite`: SQLite database
- `path`: File path utilities

8.1.4 Connectivity & Sharing

- `nearby_connections`: Peer-to-peer discovery and data transfer
- `flutter_blue_plus`: Bluetooth Low Energy support
- `connectivity_plus`: Network monitoring

8.1.5 Location & Maps

- `google_maps_flutter`: Google Maps integration
- `geolocator`: Geolocation services

8.1.6 Media & Sensors

- `image_picker`: Image/video selection
- `camera`: Camera access
- `video_player`: Video playback
- `flutter_sound`: Audio recording and playback
- `sensors_plus`: Accelerometer, gyroscope

- `mobile_scanner`: QR code scanning
- `flutter_wear_os_connectivity`: WearOS communication
- `screen_brightness`: Screen brightness control
- `qr_flutter`: QR code generation

8.1.7 UI & Visualization

- `fl_chart`: Charts and graphs
- `video_thumbnail`: Generate video thumbnails

8.2 Backend Dependencies (requirements.txt)

- `Flask`: Web framework
- `Flask-CORS`: Cross-origin resource sharing
- `Flask-SQLAlchemy`: ORM integration
- `psycopg2-binary`: PostgreSQL driver
- `firebase-admin`: Firebase token verification
- `supabase`: Supabase Python client
- `python-dotenv`: Environment variables
- `python-dateutil`: Date parsing utilities

9 Development Challenges and Solutions

9.1 Challenge 1: Offline Data Synchronization

Problem: Concert venues typically have poor or no network connectivity, with thousands of users competing for limited bandwidth. Users expect the app to work seamlessly regardless of network conditions, and data loss is unacceptable.

Solution: We implemented an offline-first architecture where the local SQLite database serves as the source of truth. All operations work offline by default, and synchronization happens in the background when connectivity is restored. We implemented:

- Optimistic UI updates

- Queue-based sync with retry logic
- Conflict resolution strategies
- Network state monitoring

9.2 Challenge 2: Firebase Auth Integration with Custom Backend

Problem: Firebase Authentication doesn't integrate directly with Flask backends. We needed a way to verify user identity for every API request without compromising security.

Solution: Implemented JWT token-based authentication flow:

1. Flutter app obtains Firebase ID token after login
2. Token is included in Authorization header for all requests
3. Flask middleware verifies token using Firebase Admin SDK
4. User UID is extracted and used for authorization
5. Tokens are refreshed automatically when expired

9.3 Challenge 3: Peer-to-Peer Connectivity

Problem: Enabling users to share content directly without internet access required a robust peer-to-peer solution that works across different Android devices.

Solution: We utilized the `nearby_connections` package to implement a discovery and advertising mechanism. This allows devices to find each other using a combination of Bluetooth and WiFi Direct, establishing a secure channel for transferring photos even when the internet is unavailable.

9.4 Challenge 4: Database Connection Pooling

Problem: Supabase Session Pooler has connection limits, and improper pooling led to connection leaks and "too many connections" errors.

Solution: Implemented environment-specific database pooling strategies:

- Development: Small QueuePool for local PostgreSQL
- Production: NullPool for Supabase Pooler (which handles pooling)
- Proper connection cleanup in finally blocks
- Connection timeout configurations

9.5 Challenge 5: Media Upload to Cloud Storage

Problem: Direct client-side uploads to Supabase Storage would expose service role keys in the mobile app, creating a security vulnerability.
Solution: Implemented a proxy upload system:

1. Mobile app sends media to backend endpoint
2. Backend authenticates user via Firebase token
3. Backend uploads to Supabase using secure service role key
4. Backend returns public URL to mobile app
5. Mobile app stores URL in local and remote databases

9.6 Challenge 6: Real-Time Sensor Data Handling

Problem: High-frequency sensor data was overwhelming the UI thread, causing stuttering animations and poor battery life.
Solution: We optimized the data handling by:

- Using efficient stream listeners
- Aggregating data points before visualization
- Offloading heavy computations where possible
- Storing raw data asynchronously in batches to avoid blocking the main thread

9.7 Challenge 7: Bluetooth File Recovery (Deep Search)

Problem: In peer-to-peer transfers, sometimes the file payload would arrive but the metadata (filename, sender info) would be lost due to connection interruptions, leaving "orphan" files in the device storage.
Solution: We implemented a "Deep Search" algorithm in the `BluetoothShareService`. When a transfer is incomplete, the system scans the specific download directory for files matching the expected pattern or timestamp, recovering the content and associating it with the correct conversation context.

9.8 Challenge 8: UI Responsiveness & Optimistic Updates

Problem: Operations like "Start Event" or "Follow User" felt sluggish because the UI waited for the backend response before updating the state.

Solution: We adopted an "Optimistic UI" approach. The app updates the local state and UI immediately upon user interaction, while the network request happens in the background. If the request fails, the state is rolled back, ensuring a snappy user experience without compromising data integrity.

9.9 Challenge 9: Foreground Notifications

Problem: Firebase Cloud Messaging (FCM) notifications do not show as "heads-up" pop-ups when the app is in the foreground on Android, making it easy for users to miss messages while using the app.

Solution: We integrated `flutter_local_notifications` to intercept FCM messages when the app is open. We manually construct and display a high-priority local notification, ensuring users are alerted immediately regardless of the app state.

10 Project Diagrams

This section visually represents the application's architecture, data flow, structure, and navigation paths.

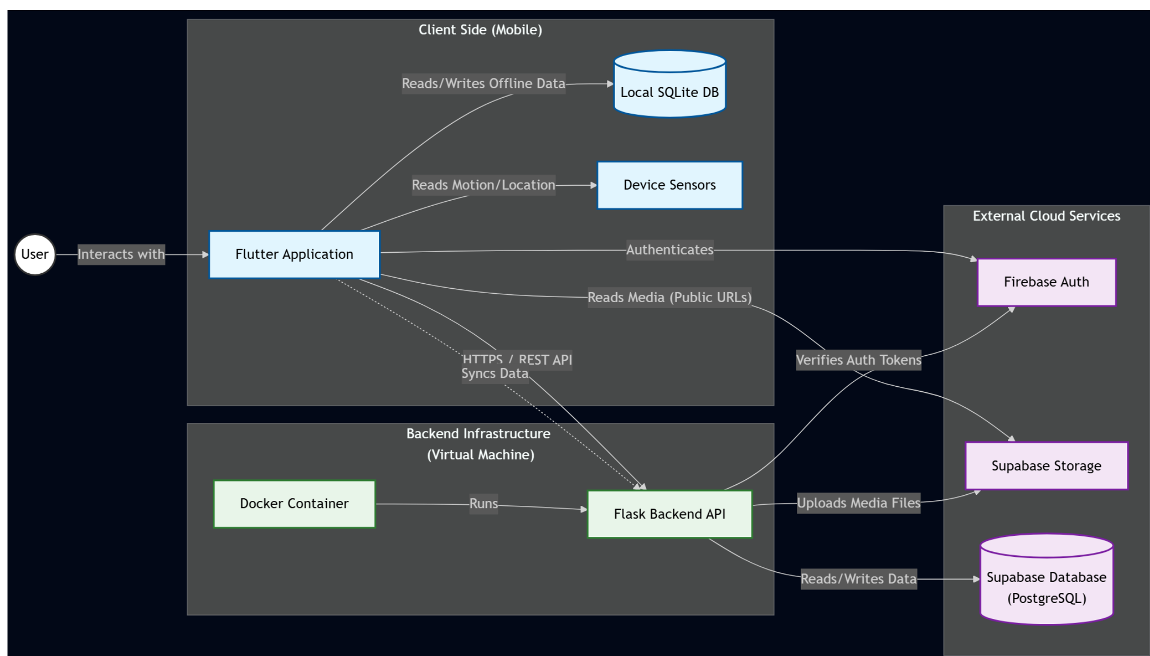

Figure 41: General Architecture Diagram

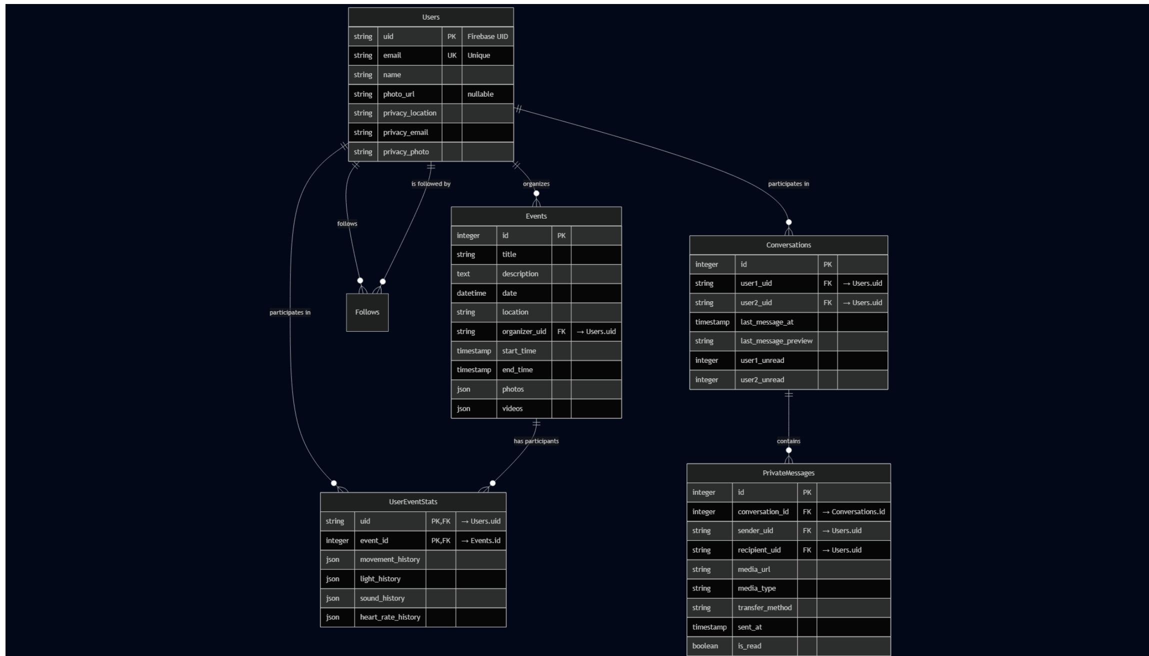

Figure 42: Entity Relationship Diagram (ERD)

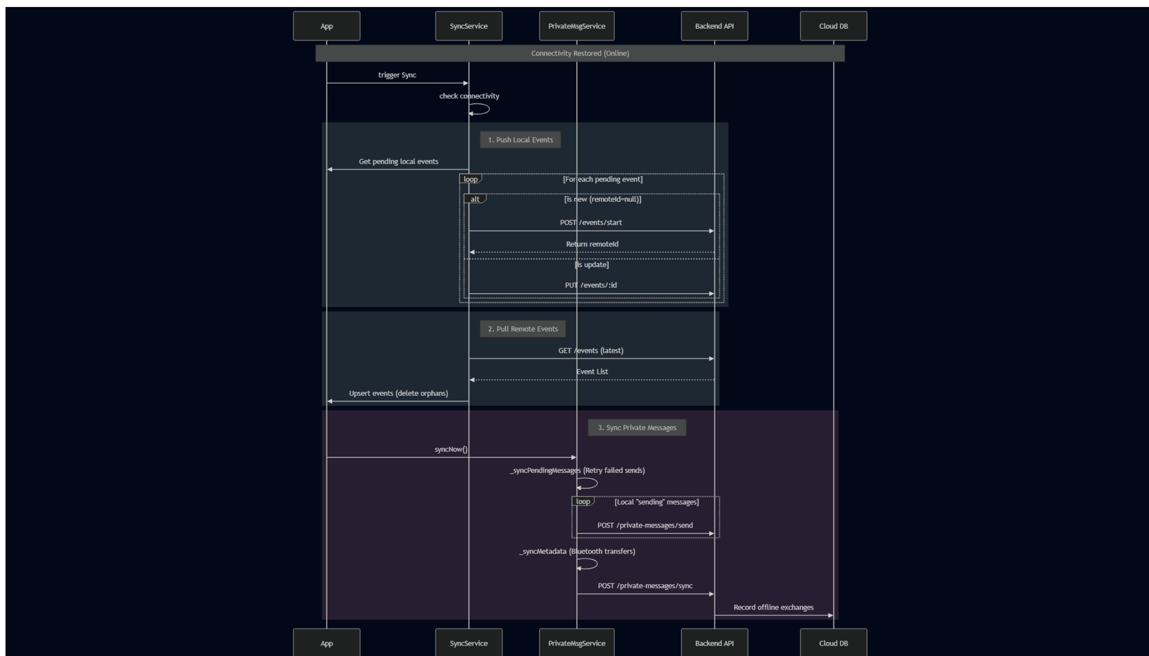

Figure 43: Sync Sequence Diagram

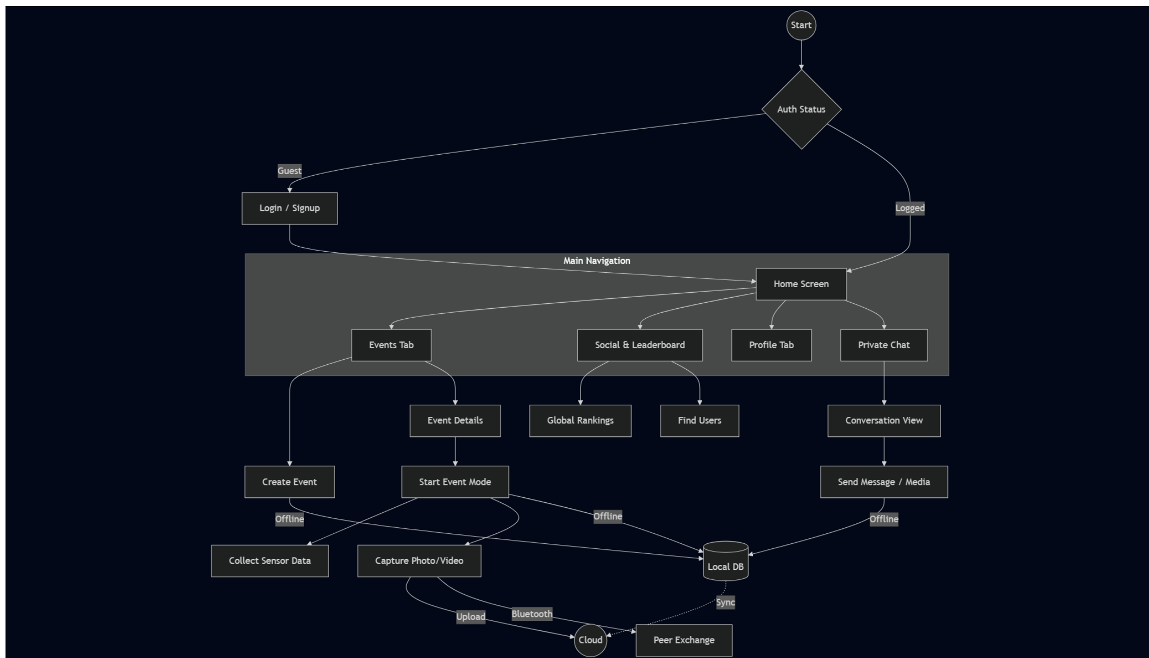

Figure 44: User Flow

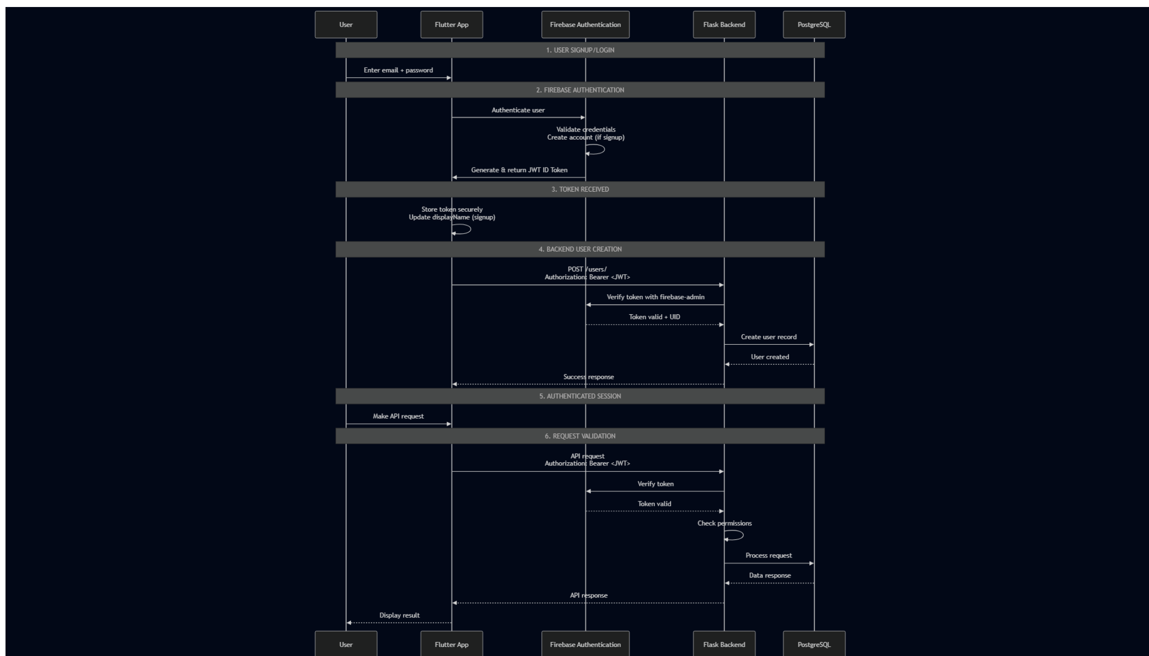

Figure 45: Authentication Flow

11 Tutorial and Usage Scenarios

This section demonstrates FanZone's key features through practical usage scenarios. Each scenario showcases the application's intuitive interface and comprehensive functionality.

11.1 Scenario 1: First-Time User Registration

A new user downloads FanZone and begins the registration process:

1. User opens the app and sees the welcome screen
2. Taps "Sign Up" to create an account
3. Fills in personal information (first name, last name, email, password)
4. Accepts permissions for camera, location, and sensors
5. Completes profile setup with optional profile picture
6. Lands on the home screen ready to explore events

11.2 Scenario 2: Discovering and Joining Events

An existing user browses and joins an upcoming concert:

1. User navigates to Events tab
2. Searches for events by artist name or venue
3. Views event details including date, location, and attendees
4. Taps "Join Event" to register attendance
5. Receives confirmation and event is added to profile

11.3 Scenario 3: Capturing Concert Memories

During a concert, a user captures and shares media:

1. User opens camera interface within the app
2. Captures photos and videos of the performance
3. Media is saved locally and marked for sync
4. User views captured media in gallery
5. When connectivity is restored, media uploads to cloud storage

11.4 Scenario 4: Monitoring Sensor Data

A user checks real-time sensor data during an event:

1. User navigates to the Sensors tab within an Event
2. Views real-time accelerometer data showing movement intensity
3. Monitors sound level meter displaying concert volume
4. Checks heart rate from connected WearOS device
5. Sensor data is associated with the current event

11.5 Scenario 5: Data Synchronization

After leaving the venue, the user's data syncs automatically:

1. App detects WiFi connection
2. Sync notification appears
3. Progress indicators show upload status
4. Events, media, and profile updates sync to backend
5. Sync completion notification confirms success

11.6 Scenario 6: Viewing Statistics

A user reviews their concert activity:

1. User navigates to Profile screen
2. Views statistics: events attended, media shared, followers
3. Browses concert history timeline
4. Explores event memories and photos from past concerts
5. Charts visualize concert attendance over time

11.7 Scenario 7: Social Features

A user connects with other concert attendees:

1. User navigates to Discover tab
2. Searches for other users by name
3. Views user profile showing shared concert attendance
4. Taps "Follow" to connect
5. Receives follow confirmation
6. Follows user's concert activity in feed

11.8 Scenario 8: Creating a New Event

An event organizer creates a new concert event:

1. User taps "Create Event" button
2. Fills event form (title, description, date, location)
3. Uses location picker to select venue on map
4. Optionally adds event banner image
5. Submits event creation
6. Event appears in public event list and organizer's profile

11.9 Scenario 9: Event Participation and Live Statistics

A user participates in an event and monitors live metrics:

1. User navigates to Event Detail screen
2. Taps "Start Participation" button
3. Views live dashboard with real-time graphs (movement, light, sound, heart rate)
4. Monitors participation intensity during the concert
5. Taps "Stop Participation" to end tracking
6. Views sensor history in Sensors tab

11.10 Scenario 10: Media Sharing with Privacy Options

A user captures and shares media during an event:

1. User taps camera button in Event Detail screen
2. Selects sharing option: "Share with all", "Share with friends", or "Don't share"
3. Captures photo or video (up to 10MB)
4. Media appears in appropriate tab (Private, Shared, or Social)
5. Other users can view shared media with sharer's profile picture and name

11.11 Scenario 11: Private Messaging with Online/Offline Options

A user sends a private photo to a friend:

1. User navigates to Private Share screen
2. Selects Friends tab
3. Taps "Send" button next to friend's name
4. Chooses photo from gallery
5. Selects sending method: "Send Online (WiFi)" or "Send Offline (Bluetooth)"
6. For Bluetooth: Both users enable advertising/discovery
7. Photo appears in Conversations tab with sent/seen status

11.12 Scenario 12: Leaderboard and Badge System

A user checks their ranking and badge progress:

1. User navigates to Stats screen, Leaderboard tab
2. Views podium with top 3 users
3. Applies category filter (e.g., "Movement" or "Badges")
4. Searches for specific user in leaderboard
5. Clicks on user's profile picture to view detailed profile
6. Navigates to own Profile to check badge progress
7. Clicks on locked badge to see unlock requirements

11.13 Scenario 13: Location Sharing and Map Discovery

A user shares location and finds nearby friends:

1. User navigates to Discover screen, Map tab
2. Enables location sharing toggle (share with friends or all)
3. Views map with profile picture pins for users sharing location
4. Searches for specific friend by name
5. Map automatically zooms to friend's location
6. Clicks on profile picture pin to view name and email
7. Uses refresh button to update locations

11.14 Scenario 14: QR Code Sharing and Privacy Settings

A user connects with others via QR code and manages privacy:

1. User navigates to Discover screen
2. Taps QR code icon to display personal QR code
3. Friend scans QR code, automatically following the user
4. User navigates to Profile screen
5. Accesses Privacy Settings
6. Configures visibility for profile picture, email, followers, and following lists
7. Sets each to "All", "Friends only", or "No one"

11.15 Scenario 15: Notification Center

A user checks recent notifications:

1. User navigates to Profile screen
2. Taps heart icon in top-right corner
3. Views list of notifications:
    - Leaderboard position changes
    - Friends within 500m proximity
    - New follower gained
    - Badge unlocked
4. Taps notification to view relevant details

12 Improvements since the previous Milestone

Following the previous milestone, FanZone was significantly expanded both functionally and structurally. Several core features were completed, refined, or newly introduced, resulting in a more mature, cohesive, and engaging application. The most relevant improvements are summarized below, ordered by their overall impact.

1. **Leaderboard and Gamification Expansion.**
   The leaderboard system was extended with a podium view for the top three users, expandable rankings, and category-based filters. Unlocked badges now contribute directly to leaderboard scores, strengthening competition and engagement.
2. **Centralized Notification System.**
   A notification center was introduced, aggregating events such as leaderboard changes, nearby friends, new followers, location-sharing activity, and badge unlocks.
3. **Comprehensive Privacy Settings.**
   A dedicated privacy module enables granular visibility control for profile pictures, email addresses, and followers/following lists using three levels (All, Friends, None).
4. **Media Organization and Sharing Refinement.**
   Media content was reorganized by separating personal and shared media. The *My Media* and *Social* sections were split to improve clarity and privacy.
5. **Enhanced Media Capture and Viewing Experience.**
   The camera interface now supports explicit sharing options, with improved media visualization through full-screen photos and enhanced video playback.
6. **Location Sharing with Privacy-Aware Visualization.**
   Location sharing was redesigned with selective visibility and interactive map pins displaying user profile pictures and details.
7. **Social Relationships and Discovery Improvements.**
   User discovery now distinguishes followers from mutual friendships, with consistent representation across social features and manual refresh options.
8. **QR Code-Based Social Connectivity.**
   QR codes enable instant user connections through quick scanning and automatic follow actions.

13 Overall Assessment

The FanZone project successfully demonstrates the application of core mobile computing principles in a real-world context. The development process provided valuable hands-on experience with modern mobile development technologies and architectural patterns.

13.1 Key Achievements

Offline-First Architecture: The implementation of a fully functional offline-first system proved to be one of the project's strongest features. Users can perform core operations without network connectivity, addressing the primary challenge of poor connectivity in concert venues.

Cross-Platform Development: Using Flutter enabled efficient development for both iOS and Android platforms from a single codebase, significantly reducing development time while maintaining consistent user experience across platforms.

Comprehensive Sensor Integration: Successfully integrating multiple device sensors (accelerometer, gyroscope, magnetometer, ambient light, sound meter) and WearOS heart rate monitoring provides rich contextual data that enhances the concert experience.

Secure Authentication Flow: The integration of Firebase Authentication with a custom Flask backend demonstrates proper security practices, including JWT token verification and secure media upload proxying.

Scalable Backend Architecture: The Flask backend with PostgreSQL/Supabase provides a solid foundation for growth, with proper database connection pooling, RESTful API design, and environment-specific configurations.

13.2 Technical Learning Outcomes

Throughout the development of FanZone, we gained substantial expertise in:

- State management using Provider pattern
- Asynchronous programming with Dart Futures and Streams
- Local data persistence with SQLite
- RESTful API design and consumption
- Firebase services integration (Authentication, Cloud Storage)
- Real-time sensor data processing
- Cross-platform mobile development best practices
- Backend development with Flask and SQLAlchemy
- Docker containerization for development and deployment

13.3 Areas for Future Enhancement

While FanZone successfully meets its core objectives, several areas have been identified for future improvement and extension. These enhancements aim to increase robustness, user engagement, and the overall richness of the concert experience:

- **Improved Bluetooth File Transfer**: Resolve existing limitations in offline Bluetooth media sharing, particularly the filepath recovery issue, to enable fully reliable peer-to-peer photo and video exchange without internet connectivity.
- **Extended Media Sharing in Private Messaging**: Expand private conversations to support video sharing in addition to photos, ensuring feature parity between public and private media interactions.
- **Event Rules and Guidelines Section**: Introduce a dedicated section within each event for displaying rules, restrictions, or important guidelines defined by organizers, improving clarity and participant awareness.
- **Text-Based Event Forums**: Implement discussion forums within events to allow attendees to share information, coordinate activities, ask questions, or exchange experiences before, during, and after concerts.
- **Interactive Concert Quizzes**: Add optional quizzes related to concerts, artists, or event history to enhance user engagement and provide additional gamification opportunities tied to badges or leaderboard rewards.
- **Event Timeline and Highlight Markers**: Allow users to create timestamps for notable moments during concerts (e.g., favorite songs or memorable interactions), enriching the event timeline and improving post-event recall.
- **Advanced Event Discovery and Filtering**: Enhance event browsing by introducing filters based on date, location, and event status, allowing users to more easily discover relevant concerts.
- **Real-Time Event Participation Visibility**: Enable users to see, in real time, who is currently participating in an event, increasing the sense of presence and social awareness during live concerts.
- **Advanced Analytics and Insights**: Provide deeper analytical insights into participation patterns, sensor data trends, and long-term user behavior through richer visualizations and aggregated statistics.
- **Multi-Device Synchronization**: Extend the synchronization architecture to support multiple devices per user account, ensuring a seamless experience across phones and wearables.

14 Contribution Assessment

14.1 Carolina Silva (Nº 113475) - 50%

Frontend Development:

- Implemented Event Detail Screen with complex tab navigation
- Integrated Sensor and Wear OS (Heart Rate) sensor connectivity and data acquisition
- Developed Sensor Data Visualization using `fl_chart`
- Created Private Share Screen and Conversation logic
- Implemented Bluetooth "Deep Search" recovery mechanism
- Integrated `flutter_local_notifications` for push notifications

Backend Development:

- Designed Event and Private Message database models
- Implemented Event API endpoints (CRUD operations)
- Created Private Messaging API routes
- Configured Firebase Admin SDK integration

Report Contributions:

- Sections 1-4: Introduction, Project Context, Requirements, Architecture
- Section 9: Development Challenges and Solutions
- Section 12: Tutorial and Usage Scenarios (Scenarios 1-4)
- Section 14: Contribution Assessment

14.2 Luana Reis (Nº 131193) - 50%

Frontend Development:

- Developed Home, Events, Leaderboard and Profile screens
- Implemented Authentication flow (Login/Signup)
- Created "Optimistic Update" logic for responsive UI
- Implemented User Search and Follow/Unfollow logic
- Integrated Google Maps for event location

Backend Development:

- Designed User database model and authentication logic
- Implemented User API endpoints (Profile, Search, Follow)

- Created Storage proxy routes for secure media upload
- Managed Supabase database connection pooling

Report Contributions:

- Sections 5-8: Widget Tree, Services/Providers, Backend Architecture, Dependencies
- Section 10-11: Project Diagrams, Development Challenges and Solutions
- Section 15: Conclusion

14.3 Collaborative Work

Several aspects of the project required close collaboration:

- API contract definition and endpoint design
- Database schema design for both SQLite and PostgreSQL
- Synchronization logic and conflict resolution strategies
- Testing across physical devices and platforms
- Code reviews and debugging sessions
- Project documentation and README maintenance

15 Conclusion

The FanZone project represents a comprehensive exploration of modern mobile application development, successfully addressing the challenge of creating a reliable, feature-rich application for concert attendees. By implementing an offline-first architecture, we ensured the app remains functional even in the challenging network conditions typical of concert venues.

Throughout the development process, we encountered and overcame significant technical challenges, from implementing secure authentication flows to managing real-time sensor data and ensuring seamless data synchronization. Each challenge provided valuable learning opportunities and deepened our understanding of mobile computing principles.

The choice of Flutter for cross-platform development proved highly effective, allowing us to build a consistent user experience across iOS and Android while maintaining a single codebase. The integration of Firebase Authentication, custom Flask backend, and Supabase services demonstrates our ability to architect a complete full-stack mobile solution with proper separation of concerns.

The sensor integration aspect of the project, including accelerometer, gyroscope, sound meter, ambient light, and WearOS heart rate monitoring, showcases the potential for mobile devices to capture rich contextual data that enhances user experiences. This data transforms FanZone from a simple media gallery into an immersive concert companion that preserves not just visual memories but the complete atmosphere of live events.

From a technical perspective, the project successfully demonstrates mastery of key mobile development concepts: state management, local data persistence, RESTful API consumption, asynchronous programming, and responsive UI design. The offline-first architecture, complemented by peer-to-peer sharing capabilities using Bluetooth and WiFi Direct, represents a robust solution to real-world connectivity challenges.

Beyond the technical accomplishments, FanZone addresses a genuine need in the concert-going community. By providing a dedicated platform for organizing concert memories, tracking attendance, and connecting with fellow music fans, the application creates lasting value that extends far beyond the duration of individual events.

The collaborative nature of this project fostered teamwork, code review practices, and effective communication—skills essential for professional software development. The even distribution of responsibilities ensured both team members gained broad experience across the full development stack, from mobile UI to backend services.

In conclusion, FanZone stands as a testament to the power of modern mobile development frameworks and cloud services to create sophisticated, user-centric applications. The project has equipped us with practical skills and theoretical knowledge that will prove invaluable in future mobile computing endeavors, whether in academic research or professional software development.