BeaconIO (BeaconFit) Developer Manual

Spencer Evenson
CSCI 460
2025

Overview

BeaconIO is a comprehensive fitness tracking application built with a Swift/SwiftUI frontend and a Node.js/Express/MongoDB backend. The application uses Bluetooth Low Energy (BLE) to detect when users are near workout stations, allowing them to track their fitness activities with minimal manual input.

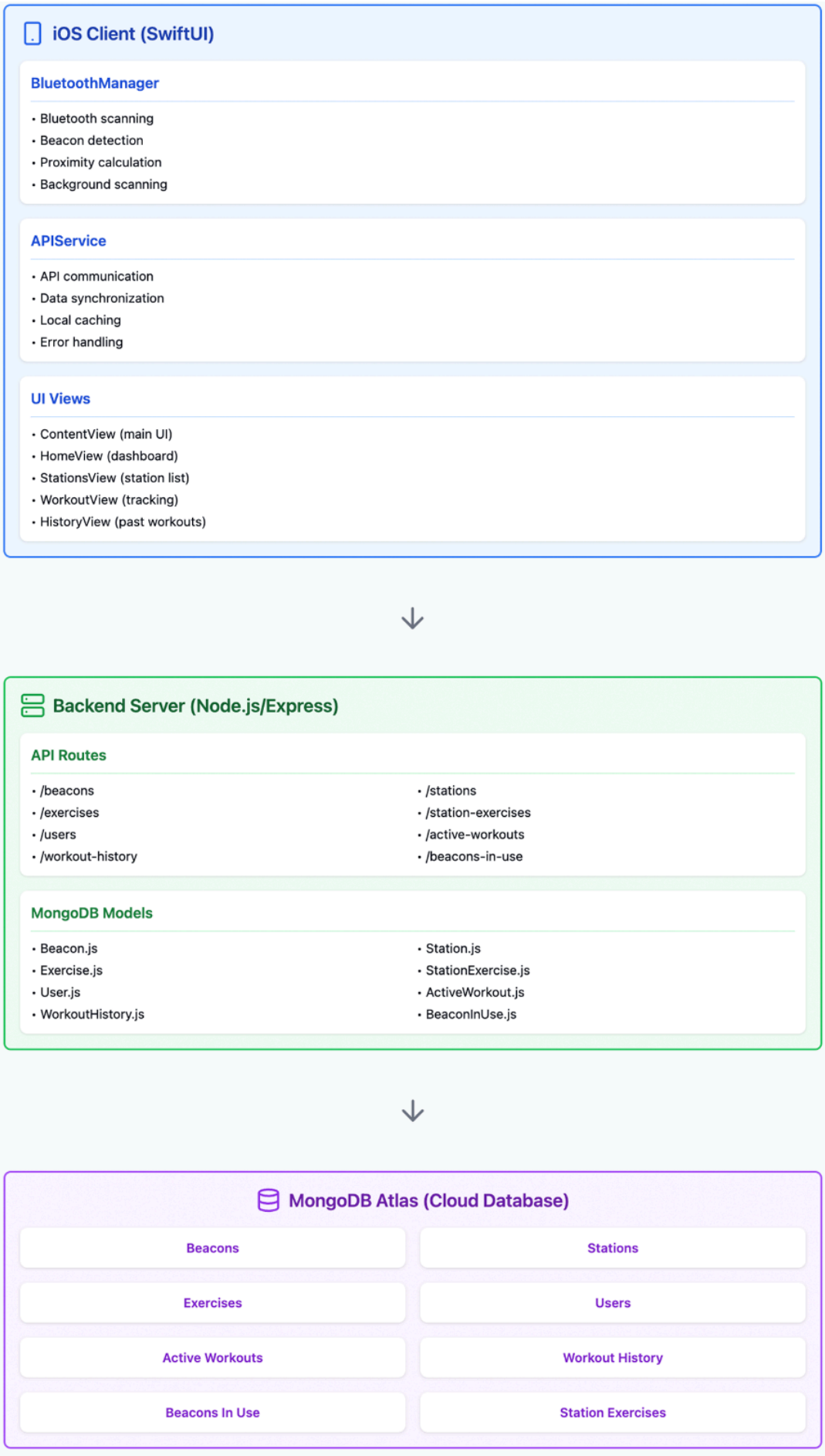
This manual provides technical details for developers working on the BeaconIO codebase, including setup instructions, architecture overview, file descriptions, and development guidelines.

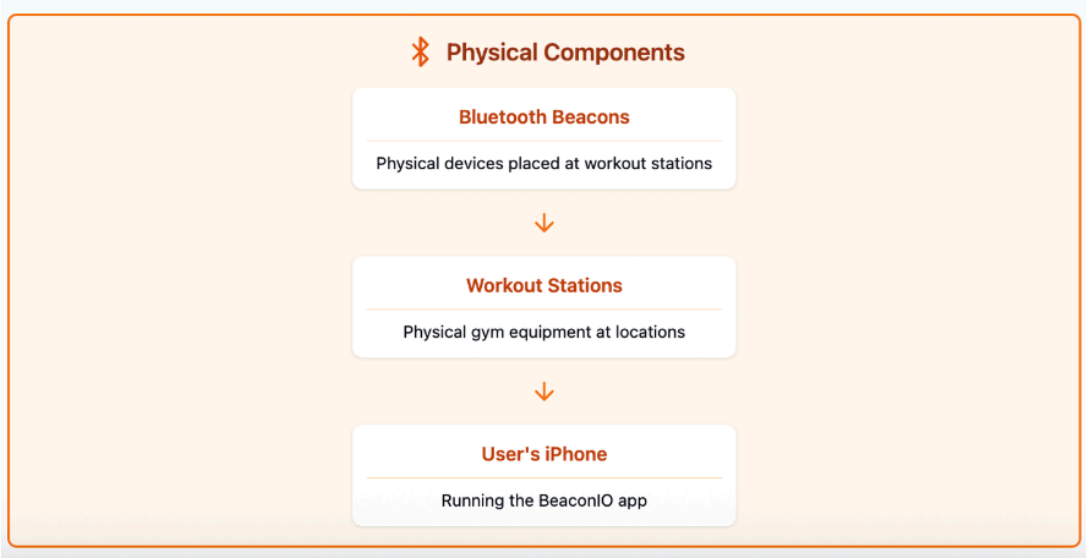
System Architecture

BeaconIO follows a client-server architecture with these main components:

1. **iOS Client Application**: Swift/SwiftUI app that interacts with Bluetooth beacons and provides the user interface
2. **RESTful API Server**: Node.js/Express backend that handles data operations
3. **MongoDB Database**: NoSQL database that stores all application data

Architecture Diagram


iOS Client (SwiftUI)
BluetoothManager
• Bluetooth scanning
• Beacon detection
• Proximity calculation
• Background scanning
APIService
• API communication
• Data synchronization
• Local caching
• Error handling
UI Views
• ContentView (main UI)
• HomeView (dashboard)
• StationsView (station list)
• WorkoutView (tracking)
• HistoryView (past workouts)
Backend Server (Node.js/Express)
API Routes
• /beacons
• /exercises
• /users
• /workout-history
• /stations
• /station-exercises
• /active-workouts
• /beacons-in-use
MongoDB Models
• Beacon.js
• Exercise.js
• User.js
• WorkoutHistory.js
• Station.js
• StationExercise.js
• ActiveWorkout.js
• BeaconInUse.js
MongoDB Atlas (Cloud Database)
Beacons
Stations
Exercises
Users
Active Workouts
Workout History
Beacons In Use
Station Exercises



Development Environment Setup

iOS App Development Prerequisites

1. **Xcode 14.0+**: Required for iOS development
2. **Swift 5.5+**: The programming language used
3. **iOS 15.0+ SDK**: Target SDK version
4. **CocoaPods or Swift Package Manager**: For managing dependencies
5. **Physical iOS Device**: Required for testing Bluetooth functionality (simulator does not support full BLE capabilities)

Backend Development Prerequisites

1. **Node.js 14.0+**: JavaScript runtime
2. **npm or yarn**: Package manager
3. **MongoDB Atlas account**: Cloud database service (no local MongoDB installation required)
4. **Express.js 4.0+**: Web framework

Setting Up the Development Environment

iOS App Setup

1. Download and extract the iOS project files to your development machine:
    - Extract the BeaconIO iOS project files to a location on your computer
    - Navigate to the extracted folder
2. Open the project in Xcode:
    - Double-click on the **BeaconIO.xcodeproj** file

    - Alternatively, open Xcode and select "Open..." from the File menu, then navigate to the project location
3. Configure the development team in Xcode:
    - Select the project in the Project Navigator
    - Select the BeaconIO target
    - Go to the Signing & Capabilities tab
    - Select your development team
4. Update the API base URL in **APIService.swift** to point to your development server:
   swift

```
Unset
let baseURL = "http://localhost:3948/api"
```

5. Connect a physical iOS device and build/run the project

Backend Setup

1. Download and extract the backend project files:
    - Extract the BeaconIO backend files to a location on your server
    - Navigate to the extracted folder using the command line
2. Install dependencies:
   bash

```
Unset
      npm install
```

3. Create a `.env` file with the following content:

```
Unset

PORT=3948

MONGO_URI=mongodb+srv://<username>:<password>@<cluster>.mongodb.n
et/beaconio?retryWrites=true&w=majority
```

Replace **<username>**, **<password>**, and **<cluster>** with your MongoDB Atlas credentials

3. Start the server:

In your console

```
Unset
npm start
```

Project Structure

iOS App Structure

The iOS app follows a modular architecture with these main components:

Core Files

File	Description
BeaconIOApp.swift	Entry point for the application, sets up the environment objects
ContentView.swift	Main view controller with tab-based navigation
APIService.swift	Handles all API communication with the backend
BluetoothManager.swift	Manages BLE scanning and beacon detection
Models.swift	Data models for the API responses
StationData.swift	Local models for station information

View Files

File	Description
HomeView.swift	Main dashboard showing nearby stations and suggestions
StationsView.swift	List of all available workout stations
StationDetailView.swift	Detailed view of a specific station
WorkoutTrackingView.swift	Interface for tracking workout progress
HistoryView.swift	View showing past workout history

File	Description
HomeView.swift	Main dashboard showing nearby stations and suggestions
StationsView.swift	List of all available workout stations
APIView.swift	Admin view for exploring raw API data

Supporting Files

File	Description
BeaconData.swift	Structs for representing beacon data
RadioButtonGroup.swift	Custom UI component for selection controls
TestViewController.swift	Test harness for UI components
Info.plist	App configuration including permissions

Backend Structure

The backend follows a standard Express.js structure with routes and models:

Main Files

File	Description
server.js	Main entry point that sets up Express and connects to MongoDB

Routes

File	Description
beacons.js	API endpoints for managing beacons
beaconInUse.js	Endpoints for tracking which beacons are in use
stations.js	Endpoints for managing stations
exercises.js	Endpoints for exercises
stationExercises.js	Endpoints for station-exercise relationships

users.js	User management endpoints
activeWorkouts.js	Endpoints for tracking in-progress workouts
workoutHistory.js	Endpoints for completed workout history

Models

File	Description
Beacon.js	MongoDB schema for beacons
BeaconInUse.js	Schema for tracking beacon usage status
Station.js	Schema for workout stations
Exercise.js	Schema for exercises
StationExercise.js	Schema for mapping stations to exercises
User.js	Schema for user data
ActiveWorkout.js	Schema for in-progress workouts
WorkoutHistory.js	Schema for completed workouts

Key Algorithms and Program Flow

Bluetooth Scanning and Proximity Detection

BeaconIO uses CoreBluetooth to scan for and detect BLE beacons. The key steps in this process are:

1. **Initialize Scanning**: **BluetoothManager** creates a **CBCentralManager** to manage Bluetooth operations
2. **Scan for Peripherals**: Performs a scan for all nearby BLE devices
3. **Filter Peripherals**: Applies name-based filtering to identify BeaconIO beacons
4. **Proximity Calculation**: Uses RSSI (Received Signal Strength Indicator) values to determine distance
5. **Threshold Application**: Compares RSSI values against beacon-specific thresholds from the API

swift

Unset

```
// Core proximity detection logic in BluetoothManager.swift
func centralManager(_ central: CBCentralManager, didDiscover
peripheral: CBPeripheral,
                    advertisementData: [String : Any], rssi RSSI:
NSNumber) {
    // Apply filtering
    if shouldIncludeBeacon(peripheral) {
        let peripheralName = peripheral.name ?? "Unknown Beacon"
        let currentRSSI = RSSI.intValue

        // Update or create beacon
        if let index = discoveredBeacons.firstIndex(where: {
$0.name == peripheralName }) {
            // Update existing beacon with new RSSI
            var updatedBeacon = discoveredBeacons[index]
            updatedBeacon.rssi = currentRSSI
            discoveredBeacons[index] = updatedBeacon
        } else {
            // Create new beacon with threshold from API or
default
            let threshold = beaconThresholds[peripheralName] ??
-60
            let newBeacon = BeaconData(
                id: UUID(),
                name: peripheralName,
                rssi: currentRSSI,
                proximityThreshold: threshold
            )
            discoveredBeacons.append(newBeacon)
        }
    }
}
```

Workout Lifecycle

The workout tracking flow follows these stages:

1. **Workout Initiation**: User starts a workout, creating an **ActiveWorkout** record
2. **Exercise Tracking**: User performs exercises, tracking sets, reps, and weights
3. **Workout Pausing**: User can pause the workout, which updates the status and records pause time
4. **Workout Resumption**: User can resume a paused workout, calculating total paused duration
5. **Workout Completion**: User completes the workout, which:
    - Converts the **ActiveWorkout** to a **WorkoutHistory** record
    - Calculates total duration (excluding paused time)
    - Removes the **ActiveWorkout** record

JavaScript

```
// Example: Completing a workout (from activeWorkouts.js)
router.put('/:id/complete', async (req, res) => {
  try {
    const { endTime } = req.body;

    if (!endTime) {
      return res.status(400).json({ message: 'End time is
required' });
    }

    const activeWorkout = await
ActiveWorkout.findById(req.params.id);
    if (!activeWorkout) return res.status(404).json({ message:
'Active workout not found' });

    // Calculate total workout duration (excluding paused time)
    const totalDuration = new Date(endTime) - new
Date(activeWorkout.startTime) - activeWorkout.pausedDuration;

    // Create a workout history entry
    const workoutHistory = new WorkoutHistory({
      userId: activeWorkout.userId,
      date: new Date(endTime),
```

```
      startTime: activeWorkout.startTime,
      endTime: endTime,
      completedStations: [{
        stationId: activeWorkout.stationId,
        startTime: activeWorkout.startTime,
        endTime: endTime,
        completedExercises: activeWorkout.completedExercises
      }],
      totalDuration: totalDuration
    });

    await workoutHistory.save();

    // Mark the beacon as not in use
    await BeaconInUse.findOneAndUpdate(
      { beaconId: activeWorkout.beaconId },
      { inUse: false },
      { new: true }
    );

    // Remove the active workout
    await activeWorkout.deleteOne();

    res.json(workoutHistory);
  } catch (err) {
    res.status(400).json({ message: err.message });
  }
});
```

Data Synchronization Flow

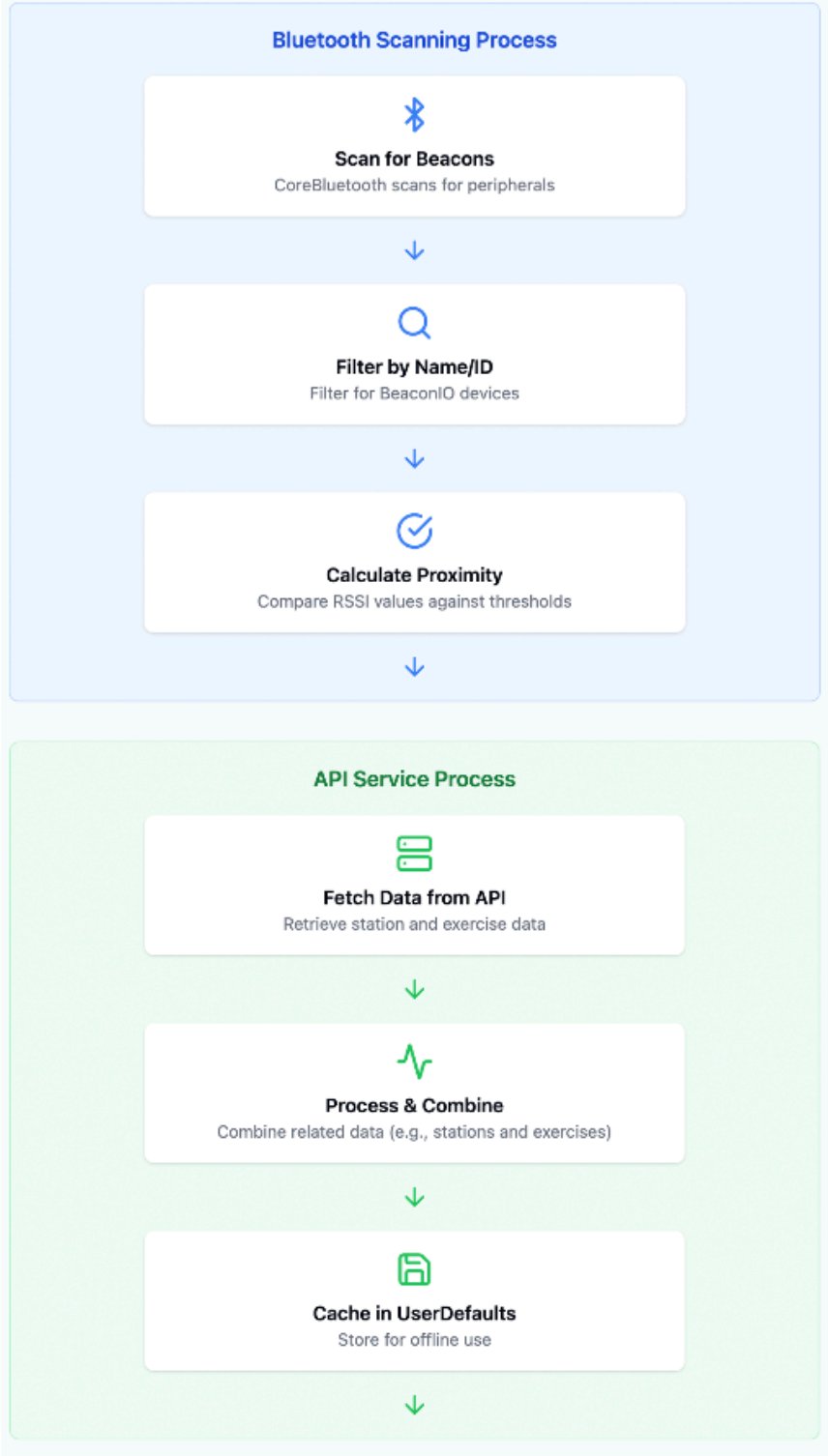
The app follows this sequence for data synchronization:

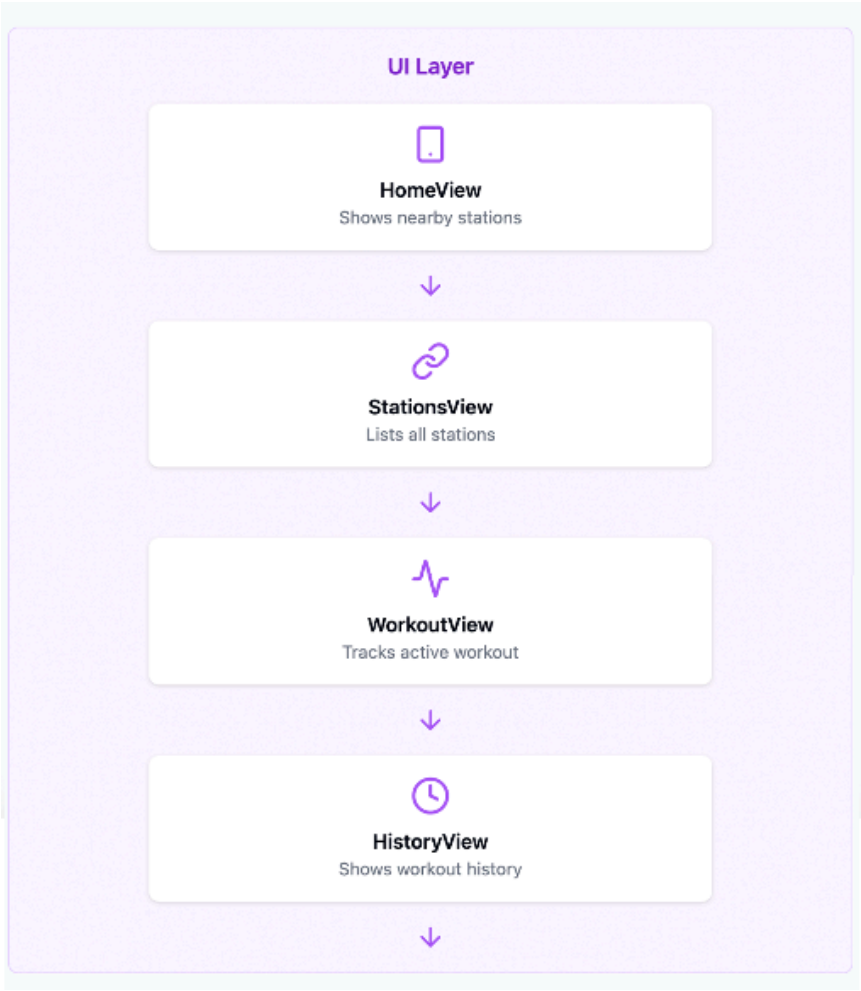
1. **Initial Data Load**: On app launch, **APIService** loads cached data from **UserDefaults**
2. **Data Refresh**: The app fetches fresh data from the API
3. **Data Processing**: **processDataForLocalStorage()** combines related data into user-friendly formats

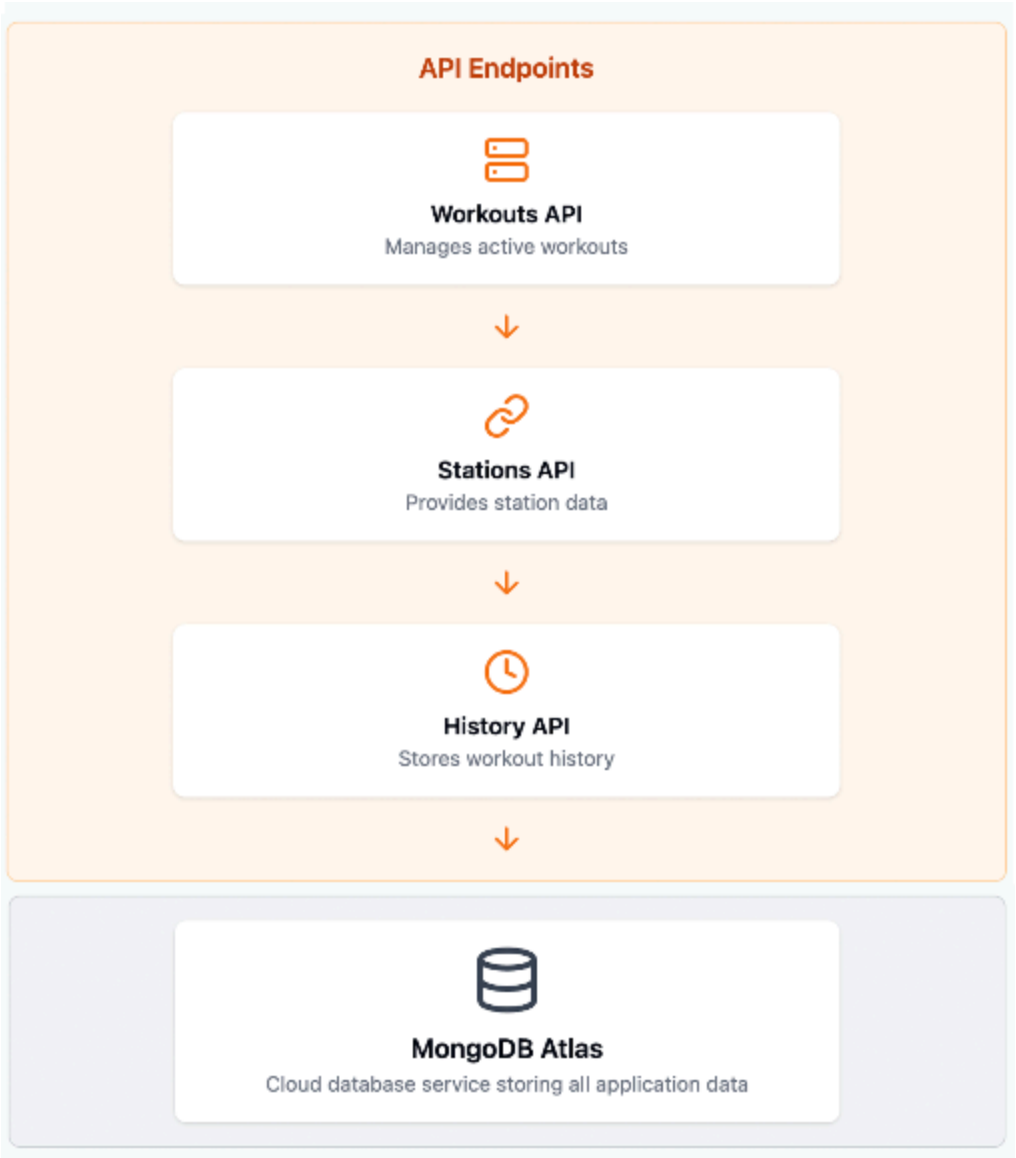
4. **Local Caching**: Processed data is cached in **UserDefaults** for offline access
5. **Incremental Updates**: During a workout, exercise data is incrementally updated

Data Flow Diagram

The following diagram illustrates how data flows through the system:




UI Layer
HomeView
Shows nearby stations
StationsView
Lists all stations
WorkoutView
Tracks active workout
HistoryView
Shows workout history



Key Components in Detail

BluetoothManager

The **BluetoothManager** is responsible for BLE operations:

- **Initialization**: Sets up the **CBCentralManager** and registers as a delegate
- **Scanning Control**: Methods to start/stop scanning and toggle filtering
- **Beacon Processing**: Filters and processes discovered peripherals
- **Proximity Calculation**: Determines if a user is within range of a beacon
- **API Integration**: Updates beacon thresholds from API data

Scanning is performed in a background mode to ensure continuous beacon detection even when the app is not in the foreground.

APIService

The **APIService** handles all communication with the backend API:

- **Data Fetching**: Methods to retrieve data from various API endpoints
- **Data Processing**: Combines related data (e.g., stations and exercises)
- **Caching**: Stores data locally for offline access
- **Workout Management**: Methods for starting, pausing, resuming, and completing workouts
- **Exercise Tracking**: Updates exercise data during a workout

The service implements a dispatch group pattern to handle parallel API requests and processes data only when all requests are complete.

ActiveWorkouts API

The **activeWorkouts.js** route handles the core workout tracking functionality:

- **Starting Workouts**: Creates new workout records and marks beacons as in use
- **Pausing/Resuming**: Updates workout status and calculates paused durations
- **Exercise Updates**: Records completed sets and exercises
- **Workout Completion**: Converts active workouts to history records

The API also handles potential conflicts, such as multiple users attempting to use the same station simultaneously.

Development Guidelines

iOS Development

1. **Core Bluetooth Usage**:
    - Always check **centralManager.state** before starting a scan
    - Use **CBCentralManagerScanOptionAllowDuplicatesKey: true** for continuous RSSI updates
    - Be mindful of battery consumption during scanning
2. **SwiftUI Best Practices**:
    - Use environment objects for shared state
    - Leverage **@Published** properties for reactivity
    - Follow the MVVM (Model-View-ViewModel) pattern
3. **Error Handling**:
    - Implement consistent error handling patterns

    - Always provide fallback data when API requests fail
    - Use optional chaining to handle nil values safely

Backend Development

1. **API Design**:
    - Follow RESTful principles
    - Use consistent response formats
    - Implement proper error handling with appropriate status codes
2. **MongoDB Usage**:
    - Use Atlas Search for complex text queries if needed
    - Create Atlas indexes for frequently queried fields
    - Use appropriate data types for optimal performance
    - Monitor database performance in the Atlas dashboard
3. **Security Considerations**:
    - Sanitize all user inputs
    - Implement proper error handling to prevent information leakage
    - Plan for future authentication implementation

Testing

iOS App Testing

1. **Unit Tests**:
    - Test core algorithms (proximity calculation, data processing)
    - Mock API responses for predictable testing
    - Test error handling and recovery
2. **UI Tests**:
    - Test navigation flows
    - Verify input validation
    - Test offline functionality
3. **Bluetooth Testing**:
    - Must be tested on physical devices
    - Verify beacon detection at various distances
    - Test scanning in background mode

Backend Testing

1. **API Tests**:
    - Test each endpoint for successful operations
    - Verify error handling with invalid inputs
    - Test concurrent operations
2. **Integration Tests**:

- Test the complete workout lifecycle
- Verify data integrity across operations
- Test race conditions and concurrency issues

Deployment

iOS App Deployment

1. **TestFlight**:
   - Build the app with appropriate signing
   - Upload to App Store Connect
   - Distribute to testers via TestFlight
2. **App Store**:
   - Prepare app metadata and screenshots
   - Complete App Review Information
   - Submit for App Store Review

Backend Deployment

1. **Development Server**:
   - Use PM2 or similar for process management
   - Set up environment variables for configuration
   - Ensure your MongoDB Atlas connection string is correctly configured
2. **Production Server**:
   - Implement HTTPS using certificates
   - Set up a reverse proxy (e.g., Nginx)
   - Configure proper logging and monitoring
   - Update the MongoDB Atlas IP Access List to include your production server

Extending the Application

Adding New Features

1. **New API Endpoints**:
   - Create route file in the backend
   - Define MongoDB schema if needed
   - Implement CRUD operations
   - Add to server.js
2. **New UI Components**:
   - Create Swift file for the view
   - Add to navigation if needed
   - Implement API integration
   - Handle loading states and errors

3. **New Bluetooth Features**:
    - Update BluetoothManager.swift
    - Test extensively with physical beacons
    - Consider battery implications

Troubleshooting

Common Issues

1. **Bluetooth Detection Problems**:
    - Verify Bluetooth permissions
    - Check beacon battery levels
    - Ensure proper beacon naming convention
    - Verify RSSI thresholds are appropriate
2. **API Connection Issues**:
    - Check network connectivity
    - Verify baseURL configuration
    - Verify Info.plist for network permissions
    - Check that your IP address is allowed in MongoDB Atlas IP Access List
    - Verify MongoDB Atlas connection string is correct
3. **Data Synchronization Problems**:
    - Verify MongoDB Atlas connection
    - Check for schema validation errors
    - Review API error responses
    - Inspect data format consistency
    - Check MongoDB Atlas logs in the Atlas dashboard

BeaconIO File and Component Connections

Core Application Structure

BeaconIOApp.swift

- **Role**: Main entry point for the application
- **Contains**:
    - Initializes **BluetoothManager** and **APIService** as environment objects
    - Sets up the main **ContentView**
- **Connections**:

    - Provides **BluetoothManager** to all child views via **.environmentObject(bluetoothManager)**
    - Provides **APIService** to all child views via **.environmentObject(apiService)**
    - Connects **BluetoothManager** to **APIService** via **bluetoothManager.setAPIService(apiService)**

ContentView.swift

- **Role**: Main container view with tab-based navigation
- **Contains**:
    - TabView with navigation to HomeView, StationsView, and HistoryView
    - Alert handling for Bluetooth errors
- **Connections**:
    - References **HomeView** in the first tab
    - References **StationsView** in the second tab
    - References **HistoryView** in the third tab
    - Uses **BluetoothManager** for error alerts
    - Uses **APIService** for data fetching

Core Services

BluetoothManager.swift

- **Role**: Manages Bluetooth beacon scanning and proximity detection
- **Contains**:
    - CoreBluetooth management
    - Beacon discovery and filtering
    - Proximity calculation logic
- **Connections**:
    - Used by **ContentView** for error handling
    - Used by **HomeView** for beacon detection and proximity information
    - References **APIService** for getting station data
    - Contains method **getStationDataForBeaconName** which calls **apiService.getStationDataForBeaconName**

APIService.swift

- **Role**: Handles all API communication with the backend
- **Contains**:
    - Data fetching methods for all backend endpoints
    - Caching logic using UserDefaults
    - Data processing and combining
    - Workout management methods

- **Connections**:
    - Referenced by **HomeView** for workout data and suggestions
    - Referenced by **StationsView** for station listings
    - Referenced by **HistoryView** for workout history
    - Referenced by **WorkoutTrackingView** for workout operations
    - Referenced by **BluetoothManager** for station data
    - Contains extensions:
        - **APIService+History.swift** for history-specific methods
        - **APIService+WorkoutTracking.swift** for workout tracking methods

Main Views

HomeView.swift

- **Role**: Main dashboard showing nearby stations and workout suggestions
- **Contains**:
    - Current stations section (stations the user is near)
    - Recently paused workouts section
    - Suggested next workout section
- **Connections**:
    - Uses **BluetoothManager** to detect nearby beacons
    - Uses **APIService** to fetch workout data and suggestions
    - References **WorkoutTrackingView** via NavigationLink for starting/resuming workouts
    - References **StationDetailView** in sheets for station information
    - Contains subviews:
        - **CurrentStationCard** for displaying detected stations
        - **RecentWorkoutCard** for showing paused workouts
        - **SuggestedWorkoutCard** for workout recommendations
        - **RefreshControl** for pull-to-refresh functionality

StationsView.swift

- **Role**: Lists all available workout stations
- **Contains**:
    - Filter controls for station types
    - List of stations with details
- **Connections**:
    - Uses **APIService** to fetch station data
    - References **StationDetailView** via NavigationLink when a station is selected
    - Contains subview **StationCard** for displaying station information

HistoryView.swift

- **Role**: Shows workout history and statistics
- **Contains**:
    - List of past workouts with summary information
    - Empty state handling
- **Connections**:
    - Uses **APIService** to fetch workout history
    - References **WorkoutDetailView** via sheet for showing detailed workout information
    - Contains subviews:
        - **WorkoutHistoryRow** for displaying workout summary
        - **WorkoutDetailView** for detailed workout information

WorkoutTrackingView.swift

- **Role**: Interface for tracking an active workout
- **Contains**:
    - Timer for workout duration
    - Exercise list with sets, reps, and weights
    - Workout control buttons (start, pause, resume, finish)
- **Connections**:
    - Uses **APIService** for workout operations (start, pause, resume, complete)
    - Contains subviews:
        - **WorkoutHeaderView** for timer and workout controls
        - **ExerciseTrackingCard** for exercise tracking
        - **SetRowView** for individual set tracking

Detail Views

StationDetailView.swift

- **Role**: Shows detailed information about a workout station
- **Contains**:
    - Station information (name, equipment, location)
    - List of available exercises
    - Start workout button
- **Connections**:
    - Can receive data from either:
        - **LocalStationData** (from API) via initializer overload
        - **StationData** (from BluetoothManager) via standard initializer
    - References **WorkoutTrackingView** via NavigationLink for starting a workout
    - Used by both **HomeView** and **StationsView**

APIView.swift

- **Role**: Admin/debug view for exploring raw API data
- **Contains**:
    - Tab selector for different data types
    - Refresh button for fetching latest data
- **Connections**:
    - Uses **APIService** directly for data fetching
    - Contains subviews:
        - **StationsListView** for raw station data
        - **BeaconsListView** for raw beacon data
        - **ExercisesListView** for raw exercise data

Supporting Files and Models

BeaconData.swift

- **Role**: Data structure for representing beacon information
- **Contains**:
    - Properties for beacon identification, signal strength, and proximity
    - Computed properties for display formatting
- **Connections**:
    - Used by **BluetoothManager** to store discovered beacons
    - Used by **HomeView** to display beacon information

StationData.swift

- **Role**: Data structure for station information
- **Contains**:
    - Station details and associated exercises
    - Static method for sample data
- **Connections**:
    - Used by **BluetoothManager** for providing station data
    - Used by **StationDetailView** for displaying station details

Models.swift

- **Role**: Contains all data models for API communication
- **Contains**:
    - Structs matching backend data models
    - Codable implementations for JSON parsing
- **Connections**:
    - Used by **APIService** for data serialization/deserialization
    - Referenced by most views for type definitions

Detailed Connection Map

From ContentView:

- → HomeView (via TabView)
- → StationsView (via TabView)
- → HistoryView (via TabView)
- ← BluetoothManager (via environmentObject)
- ← APIService (via environmentObject)

From HomeView:

- → StationDetailView (via sheet)
- → WorkoutTrackingView (via NavigationLink)
- ← BluetoothManager (via environmentObject)
- ← APIService (via environmentObject)

From StationsView:

- → StationDetailView (via NavigationLink)
- ← APIService (via environmentObject)

From HistoryView:

- → WorkoutDetailView (via sheet)
- ← APIService (via environmentObject)

From StationDetailView:

- → WorkoutTrackingView (via NavigationLink)
- ← (Receives data from either HomeView or StationsView)

From WorkoutTrackingView:

- ← APIService (via environmentObject)
- ← (Receives data from either HomeView or StationDetailView)

From BluetoothManager:

- → APIService (via reference passed in setAPIService)
- ← BeaconData (uses for storing beacon information)

From APIService:

- ← Models (uses for data serialization/deserialization)

Data Flow Connections

1. **Beacon Detection Flow**:

```
Unset
BeaconDevice → CoreBluetooth → BluetoothManager → HomeView →
StationDetailView
```

2. **Station Browsing Flow**:

```
Unset
APIService → StationsView → StationDetailView →
WorkoutTrackingView
```

3. **Workout Tracking Flow**:

```
Unset
WorkoutTrackingView → APIService → Backend → APIService →
WorkoutTrackingView
```

4. **History Review Flow**:

```
Unset
APIService → HistoryView → WorkoutDetailView
```

Component Initialization Order

1. **BeaconIOApp** initializes **BluetoothManager** and **APIService**

2. **ContentView** is created with these environment objects
3. The tab views (**HomeView**, **StationsView**, **HistoryView**) receive the environment objects
4. **APIService** fetches initial data on app launch
5. **BluetoothManager** begins scanning when HomeView appears
6. Detail views are created as needed when navigating through the app

ViewModifier and Extension Connections

- **RefreshControl.swift** in HomeView detects pull-to-refresh gestures and triggers data refresh

Resources

- Blue Charm Beacon Documentation
  - https://bluecharmbeacons.com/bc011-ibeacon-multibeacon-quick-start-guide/
- Swift Documentation
  - https://swift.org/documentation/
- Core Bluetooth Programming Guide
  - https://developer.apple.com/documentation/corebluetooth
- SwiftUI Documentation
  - https://developer.apple.com/documentation/swiftui
- Express.js Documentation
  - https://expressjs.com/
- MongoDB Atlas Documentation
  - https://docs.atlas.mongodb.com/
- MongoDB Node.js Driver
  - https://docs.mongodb.com/drivers/node/
- Swift UI Cheat Sheet
  - https://github.com/SimpleBoilerplates/SwiftUI-Cheat-Sheet
- Core Bluetooth Video
  - https://www.youtube.com/watch?v=n-f0BwxKSD0
- Startup Video
  - https://www.youtube.com/watch?v=nqTcAzPS3oc
- Apple's "Transferring Data Between BLE Devices" Sample Code
  - https://developer.apple.com/documentation/corebluetooth/transferring-data-between-bluetooth-low-energy-devices
- Punch Through's Core Bluetooth Guide
  - https://punchthrough.com/core-bluetooth-guide/
- Novel Bits iOS BLE Development Tutorial
  - https://novelbits.io/intro-ble-mobile-development-ios/
- GitHub BLE Swift Samples
  - https://github.com/shu223/iOS-BLE-Tutorials
  - https://github.com/exPHAT/SwiftBluetooth

    - https://github.com/netguru/BlueSwift
- Ray Wenderlich BLE Tutorial
    - https://www.kodeco.com/231-core-bluetooth-tutorial-for-ios-heart-rate-monitor
- QuickBird Studios Guide on Reading BLE Characteristics
    - https://quickbirdstudios.com/blog/read-ble-characteristics-swift/
- WWDC Videos on Core Bluetooth
    - https://developer.apple.com/videos/play/wwdc2019/901/