UV Monitor App

Technical Architecture & Implementation

A comprehensive guide for developers

Project Overview

UV Monitor App - A Flutter-based mobile application for real-time UV exposure monitoring with Bluetooth Low Energy (BLE) sensor integration.

Target Platforms:

- Android
- iOS
- (Windows, macOS, Linux with limited BLE support)

Course: Cross-Platform Development ASE456
Tech Stack: Flutter, Dart, BLE, ESP32

Key Features Implemented

✅ **BLE Integration** - Connect to ESP32 UV sensors via Bluetooth

✅ **Real-time Data Streaming** - Live UV index readings

✅ **State Management** - Provider pattern for reactive UI

✅ **Local Persistence** - SharedPreferences for data storage

✅ **Skin Type Quiz** - 11-question assessment based on Fitzpatrick scale

✅ **Recommendation Engine** - Personalized UV protection advice

✅ **Comprehensive Testing** - Unit, widget, and integration tests

Tech Stack

Framework & Language:

- Flutter SDK 3.9+
- Dart language

Key Dependencies:

```
flutter_blue_plus: ^1.32.0    # BLE connectivity
shared_preferences: ^2.2.0     # Local storage
permission_handler: ^11.0.0    # Runtime permissions
provider: ^6.1.0               # State management
```

Dev Dependencies:

- mockito: ^5.4.4 (mocking for tests)
- build_runner: ^2.4.13 (code generation)

Project Architecture

```
lib/
├── main.dart                    # App entry point
├── config/
│   └── ble_config.dart         # BLE configuration constants
├── data/
│   └── mock_data.dart          # Mock data & quiz questions
├── models/
│   └── data_models.dart        # Data models
├── providers/
│   └── skin_type_provider.dart # Skin type state management
├── screens/
│   ├── uv_monitor_screen.dart  # Main dashboard
│   ├── quiz_screen.dart        # Skin type quiz
│   └── results_screen.dart     # Quiz results
└── services/
    ├── ble_service.dart        # BLE communication
    └── storage_service.dart    # Local persistence
```

State Management Architecture

Provider Pattern - Reactive state management

```
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (_) => BleService()),
        ChangeNotifierProvider(create: (_) => SkinTypeProvider()),
      ],
      child: MaterialApp(/* ... */),
    );
  }
}
```

Data Models

Core Models

```
class UVReading {
  final String id;
  final double uvIndex;
  final DateTime timestamp;

  // JSON serialization
  Map<String, dynamic> toJson();
  factory UVReading.fromJson(Map<String, dynamic> json);
}

class SkinType {
  final String id;
  final String name;
  final String description;
  final Color color;
  final int burnTime;  // Minutes to burn at UV=1
```

Data Models (Continued)

```
class UVRecommendation {
  final double uvIndex;
  final int safeExposure;          // Minutes safe in sun
  final List<String> protection;   // Protection advice
}

enum BleConnectionState {
  disconnected, connecting, connected, disconnecting
}

class Device {
  final String id;
  final String name;
  final int batteryLevel;
  final bool isConnected;
  final BleConnectionState connectionState;
}
```

BLE Service Architecture

Key Responsibilities:

- Device scanning and discovery
- Connection management
- Data streaming from UV sensor
- Auto-reconnection to last device
- Permission handling

Technology: `flutter_blue_plus` package
Communication: GATT protocol with custom UUIDs

BLE Service Implementation

Device Scanning

```dart
Future<void> scanForDevices() async {
  // Request permissions
  final hasPermissions = await requestPermissions();
  if (!hasPermissions) throw Exception('Permissions denied');

  // Start scan with timeout
  await FlutterBluePlus.startScan(
    timeout: Duration(seconds: BleConfig.scanDuration),
  );

  // Listen for results
  FlutterBluePlus.scanResults.listen((results) {
    for (ScanResult result in results) {
      if (result.device.platformName.contains(
        BleConfig.deviceNameFilter
      )) {
        _discoveredDevices.add(result.device);
      }
```

BLE Service Implementation

Device Connection

```
Future<void> connectToDevice(BluetoothDevice device) async {
  // Connect with timeout
  await device.connect(
    timeout: Duration(seconds: BleConfig.connectionTimeout),
  );

  // Discover services
  final services = await device.discoverServices();

  // Find UV characteristic
  for (var service in services) {
    if (service.uuid == BleConfig.serviceUUID) {
      for (var char in service.characteristics) {
        if (char.uuid == BleConfig.characteristicUUID) {
          // Enable notifications
          await char.setNotifyValue(true);
          // Listen to updates
          _characteristicSubscription = char.lastValueStream
            .listen(_handleUVData);
        }
      }
```

BLE Configuration

Custom UUIDs defined in `ble_config.dart`:

```dart
class BleConfig {
  // Service and characteristic UUIDs
  static const String serviceUUID =
    '4fafc201-1fb5-459e-8fcc-c5c9c331914b';
  static const String characteristicUUID =
    'beb5483e-36e1-4688-b7f5-ea07361b26a8';

  // Device filtering
  static const String deviceNameFilter = 'ESP32';

  // Timeouts
  static const int scanForDevicesDurationSeconds = 10;
  static const int connectionTimeoutSeconds = 15;
}
```

Data Streaming & Processing

```dart
void _handleUVData(List<int> value) {
  try {
    // Decode bytes to string
    String dataString = utf8.decode(value);

    // Parse JSON
    Map<String, dynamic> data = jsonDecode(dataString);

    // Extract UV index
    double uvIndex = (data['uv_index'] as num).toDouble();

    // Notify listeners with new value
    _currentUVIndex = uvIndex;
    notifyListeners();

    // Callback for UI updates
    if (_onDataReceived != null) {
      _onDataReceived!(uvIndex);
    }
  } catch (e) {
    print('Error parsing UV data: $e');
  }
}
```

Storage Service

Purpose: Persist data locally using SharedPreferences

```dart
class StorageService {
  static const String _readingsKey = 'uv_readings';
  static const String _alertThresholdKey = 'uv_alert_threshold';

  Future<void> saveReadings(List<UVReading> readings) async {
    final prefs = await SharedPreferences.getInstance();
    final jsonList = readings.map((r) => r.toJson()).toList();
    final jsonString = jsonEncode(jsonList);
    await prefs.setString(_readingsKey, jsonString);
  }

  Future<List<UVReading>> loadReadings() async {
    final prefs = await SharedPreferences.getInstance();
    final jsonString = prefs.getString(_readingsKey);
    if (jsonString == null) return [];
    // Decode and return
    final jsonList = jsonDecode(jsonString) as List;
    return jsonList.map((j) => UVReading.fromJson(j)).toList();
```

UV Monitor Screen

Main Dashboard - `uv_monitor_screen.dart`

Key Features:

- Real-time UV display with color-coded levels
- Connection status indicator
- Device scanning and connection UI
- Recent readings list (last 50)
- Skin type quiz navigation
- Settings (alert threshold)

State Management:

- Listens to `BleService` for UV data

UV Monitor Implementation

UV Level Calculation

```dart
UVLevel _getUVLevel(double uvIndex) {
  if (uvIndex <= 2) return UVLevel('Low', Colors.green);
  if (uvIndex <= 5) return UVLevel('Moderate', Colors.yellow);
  if (uvIndex <= 7) return UVLevel('High', Colors.orange);
  if (uvIndex <= 10) return UVLevel('Very High', Colors.red);
  return UVLevel('Extreme', Colors.purple);
}
```

Based on WHO UV Index standards

UV Recommendation Engine

Algorithm:

1. Get current UV index from sensor
2. Get user's skin type from provider
3. Look up base recommendation for UV level
4. Adjust safe exposure time based on skin burn time
5. Return personalized advice

```
UVRecommendation _getRecommendations(SkinType skinType) {
  final baseRec = uvRecommendations.firstWhere(
    (r) => currentUV >= r.uvIndex,
    orElse: () => uvRecommendations.last,
  );
  final adjustedTime = (baseRec.safeExposure *
    (skinType.burnTime / 20)).round();
  return UVRecommendation(
```

Skin Type Quiz

Based on Fitzpatrick Skin Type Scale

11 Questions covering:

- Physical characteristics (eye color, hair, skin tone)
- Sun reaction (burning vs tanning)
- Freckling tendency
- Recent sun exposure
- Face sensitivity

Scoring System:

- Each answer has a value (0-5)
- Total score determines skin type category

Quiz Implementation

Screen: `quiz_screen.dart`

```
class _QuizScreenState extends State<QuizScreen> {
  final PageController _pageController = PageController();
  Map<int, int> responses = {};

  void _handleOptionSelect(int questionId, int value) {
    setState(() {
      responses[questionId] = value;
    });

    if (questionId == 11) {
      // Navigate to results
      Navigator.pushNamed(context, '/results',
        arguments: responses);
    } else {
      // Next question
      _pageController.nextPage(
        duration: Duration(milliseconds: 300),
        curve: Curves.easeInOut,
      );
    }
```

Quiz Results Screen

Calculates skin type from responses:

```
SkinType _calculateSkinType(Map<int, int> responses) {
  int total = 0;
  responses.forEach((key, value) {
    if (key >= 1 && key <= 10) {
      total += value;
    }
  });

  // Map score to skin type
  if (total <= 7) return skinTypes[0];  // Very Fair
  if (total <= 16) return skinTypes[1]; // Fair
  if (total <= 25) return skinTypes[2]; // Medium
  if (total <= 30) return skinTypes[3]; // Olive
  if (total <= 35) return skinTypes[4]; // Brown
  return skinTypes[5];                   // Dark
}
```

Navigation Flow

```
Main App Entry
    ↓
UV Monitor Screen (Home)
    ├─→ Scan for Devices → Connect → Stream Data
    ├─→ Take Quiz
    │      ↓
    │   Quiz Screen (11 questions)
    │      ↓
    │   Results Screen
    │      ↓
    │   Back to Monitor (with new skin type)
    └─→ Settings
```

Route Management:

- Named routes in `main.dart`
- Arguments passed via `ModalRoute`

Testing Strategy

Unit Tests

- `ble_service_test.dart` - BLE connection logic
- `storage_service_test.dart` - Data persistence
- `data_models_test.dart` - Model serialization
- `skin_type_provider_test.dart` - State management

Widget Tests

- `uv_monitor_screen_test.dart` - UI components
- `quiz_screen_test.dart` - Quiz interaction
- `results_screen_test.dart` - Results display

Running Tests

Unit & Widget Tests:

```
flutter test
```

Integration Tests:

```
# On device/emulator
flutter test integration_test/

# Specific test
flutter test integration_test/quiz_flow_integration_test.dart
```

Test Coverage:

- Run: `flutter test --coverage`
- View: `genhtml coverage/lcov.info -o coverage/html`

Building the App

Android

```
flutter build apk --release
# Output: build/app/outputs/flutter-apk/app-release.apk
```

iOS

```
flutter build ios --release
# Requires Mac and Xcode
```

Desktop (Windows example)

```
flutter build windows --release
# Note: BLE support may be limited on desktop
```

Development Setup

Prerequisites:

1. Flutter SDK 3.9 or higher
2. Dart SDK (bundled with Flutter)
3. Android Studio / Xcode for mobile development
4. VS Code with Flutter extension (recommended)

Setup Steps:

```
# Clone the repository
git clone <repo-url>
cd uv_app

# Get dependencies
flutter pub get
```

Key Design Decisions

1. Provider over Bloc/Riverpod

- Simpler learning curve for students
- Sufficient for app complexity
- Good integration with Flutter widgets

2. SharedPreferences over SQLite

- Lightweight data (JSON arrays)
- Simple key-value storage sufficient
- No complex queries needed

3. flutter_blue_plus over flutter_blue

- More actively maintained

Key Design Decisions (Continued)

4. Mock Data Approach

- Skin types defined in `mock_data.dart`
- Quiz questions centralized
- UV recommendations table-driven
- Easy to update without code changes

5. Separation of Concerns

- Services handle external communication
- Providers manage app state
- Screens are presentation-only
- Models are pure data structures

Performance Considerations

BLE Connection:

- Automatic reconnection on disconnect
- Connection timeout handling
- Scan duration limited to 10 seconds

Data Storage:

- Limit to 50 most recent readings
- JSON encoding/decoding for persistence
- Async operations to avoid UI blocking

UI Responsiveness:

- Provider pattern for efficient rebuilds

Known Limitations & Future Work

Current Limitations:

- No cloud sync (local-only storage)
- Single device connection at a time
- No historical data visualization (charts)
- Basic alert system (no push notifications)

Potential Enhancements:

- Firebase integration for cloud sync
- Data visualization with charts (fl_chart)
- Export data to CSV
- Multiple device profiles
- Push notifications for UV alerts

Code Quality & Standards

Linting: Using `flutter_lints ^5.0.0`

- Enforces Dart style guide
- Catches common errors
- Ensures code consistency

Code Organization:

- Feature-based directory structure
- Clear naming conventions
- Comprehensive documentation
- Type safety enforced

Version Control:

Documentation

Available in DOCS/:

- `requirements_features.md` - Feature specifications
- `milestones.md` - Project milestones
- `sprint_1_report.md` - Sprint retrospective
- `integration_test/README.md` - Integration test guide
- Screenshots for user reference

Code Documentation:

- Inline comments for complex logic
- Doc comments for public APIs
- README for setup instructions

Debugging Tips

BLE Connection Issues:

1. Check permissions in device settings
2. Verify Bluetooth is enabled
3. Check device name filter matches ESP32
4. Monitor logs: `flutter logs`
5. Test with BLE scanner app

State Management Issues:

1. Verify Provider setup in widget tree
2. Check `notifyListeners()` calls
3. Use `Consumer` or `context.watch()` appropriately
4. Inspect state with Flutter DevTools

Resources

Flutter:

- Flutter Documentation
- Dart Language Tour

BLE:

- flutter_blue_plus Package
- BLE GATT Specification

State Management:

- Provider Package
- Flutter State Management Guide

Testing:

Key Technical Achievements

✅ **Cross-platform BLE integration** with robust connection handling
✅ **Clean architecture** with separation of concerns
✅ **Comprehensive testing** (unit, widget, integration)
✅ **Reactive UI** with Provider state management
✅ **Data persistence** with local storage
✅ **Personalization algorithm** based on dermatology research
✅ **User-friendly interface** with Material Design 3

Lines of Code: ~3000+ (including tests)

Test Coverage: Extensive coverage across all layers

Questions?

Additional Resources

- Check the codebase on GitHub
- Review inline code documentation
- Refer to integration test examples
- Consult Flutter documentation

Happy Coding! 🚀