SheAware Flutter Project Documentation

Table of Contents

Project Overview

SheAware is a Flutter-based mobile application designed to provide women's health awareness, symptom tracking, educational resources, and support services. The app follows **Clean Architecture** principles with clear separation of concerns across data, domain, and presentation layers.

Tech Stack

- **Framework**: Flutter 3.8.1+
- **State Management**: Riverpod (flutter_riverpod)
- **Networking**: Dio
- **Code Generation**: Freezed, JSON Serializable
- **Dependency Injection**: GetIt
- **UI Components**: ScreenUtil, Shimmer, Lottie, Cached Network Image
- **Architecture**: Clean Architecture (Data-Domain-Presentation)

How to Run the Project

Prerequisites

- Flutter SDK 3.8.1 or higher
- Dart SDK
- Android Studio / Xcode (for iOS)
- A physical device or emulator

Commands

Command	Description
`flutter pub get`	Install all dependencies from pubspec.yaml

Command	Description
`flutter pub run build_runner build --delete-conflicting-outputs`	Generate code for Freezed and JSON serialization
`flutter run`	Run the app in debug mode on connected device
`flutter run --release`	Run the app in release mode
`flutter build apk`	Build Android APK
`flutter build ios`	Build iOS app
`flutter analyze`	Analyze code for issues
`flutter clean`	Clean build artifacts

Step-by-Step Setup

1. **Clone the repository**

   ```
   cd /path/to/she_aware
   ```

2. **Install dependencies**

   ```
   flutter pub get
   ```

3. **Generate code files**

   ```
   flutter pub run build_runner build --delete-conflicting-outputs
   ```

4. **Configure API Base URL**

   - Open lib/di/network_module.dart
   - Update the `baseUrl` constant:

     ```
     const baseUrl = 'http://10.0.2.2:8000/v1/'; // For Android Emulator
     // const baseUrl = 'http://localhost:8000/v1/'; // For iOS Simulator
     // const baseUrl = 'https://your-api-domain.com/v1/'; // For Production
     ```

5. **Run the application**

   ```
   flutter run
   ```

Project Architecture

SheAware follows **Clean Architecture** with three distinct layers:

```
graph TB
    subgraph Presentation Layer
        A[Screens/Widgets] --> B[Notifiers]
        B --> C[UI States]
    end
```

```
    subgraph Domain Layer
        D[Use Cases] --> E[Repositories Interfaces]
        E --> F[Models]
    end

    subgraph Data Layer
        G[Repository Implementations] --> H[Data Sources]
        H --> I[Remote APIs]
        H --> J[Local Storage]
        I --> K[API Client]
        K --> L[Dio/Interceptors]
    end

    B --> D
    G --> E

    style A fill:#e1f5ff
    style D fill:#fff4e1
    style G fill:#ffe1f5
```

Architecture Layers

1. Presentation Layer (`lib/presentation/`)

- **Screens**: UI components and pages
- **Notifiers**: StateNotifier classes managing UI state
- **States**: Freezed classes representing different UI states
- **Widgets**: Reusable UI components
- **Theme**: App-wide styling and theming

2. Domain Layer (`lib/domain/`)

- **Use Cases**: Business logic encapsulation
- **Repository Interfaces**: Abstract contracts for data operations
- **Models**: Domain entities (pure Dart classes)
- **Utils**: Result/Failure handling

3. Data Layer (`lib/data/`)

- **Repository Implementations**: Concrete implementations of domain repositories
- **Data Sources**: Remote (API) and Local (SharedPreferences) data sources
- **Models**: Request/Response DTOs
- **Mappers**: Convert DTOs to domain models
- **API Client**: HTTP client wrapper with error handling

4. **Dependency Injection (`lib/di/`)**
   - Modular DI setup using GetIt
   - Separate modules for cache, network, data sources, repositories, and use cases

---

Folder Structure

```
lib/
  main.dart                           # App entry point
  she_aware.dart                      # Root widget configuration
  injection_container.dart            # DI setup orchestration

  di/                                 # Dependency Injection Modules
      cache_module.dart              # SharedPreferences setup
      network_module.dart            # Dio, API clients, interceptors
      data_source_module.dart        # Local & remote data sources
      repository_module.dart         # Repository implementations
      use_case_module.dart           # Use case registrations
      service_module.dart            # Additional services

  data/                               # Data Layer
      datasource/
          local/
              source/
                  auth_local_data_source_impl.dart
          remote/
              api/                   # API interface definitions
                  auth_api.dart
                  auth_api_impl.dart
                  education_api.dart
                  education_api_impl.dart
                  support_api.dart
                  support_api_impl.dart
                  symptom_api.dart
                  symptom_api_impl.dart
              model/                 # DTOs
                  request/
                      auth/
                      symptom/
                  response/
                      auth/
                      education/
                      support/
                      symptom/
```

```
            source/                # Remote data source implementations
                auth_remote_data_source_impl.dart
                education_remote_data_source_impl.dart
                support_remote_data_source_impl.dart
                symptom_remote_data_source_impl.dart
            util/                  # Network utilities
                api_client.dart    # HTTP wrapper
                auth_interceptor.dart
                logging_interceptor.dart
                json_parser.dart
    mapper/                        # DTO to Domain mappers
        auth/
        education/
        support/
        symptom/
    repository/                    # Repository implementations
        source/                    # Data source interfaces
            local/
                auth_local_data_source.dart
            remote/
                auth_remote_data_source.dart
                education_remote_data_source.dart
                support_remote_data_source.dart
                symptom_remote_data_source.dart
        auth_repository_impl.dart
        education_repository_impl.dart
        settings_repository_impl.dart
        support_repository_impl.dart
        symptom_repository_impl.dart

domain/                            # Domain Layer
    model/                         # Domain entities
        auth/
        education/
        nav_item/
        support/
        symptom/
    repository/                    # Repository interfaces
        auth_repository.dart
        education_repository.dart
        settings_repository.dart
        support_repository.dart
        symptom_repository.dart
    usecase/                       # Business logic
        auth/
            check_auth_status_use_case.dart
            register_device_use_case.dart
            register_use_case.dart
         education/
            get_education_articles_use_case.dart
         onboarding/
            check_onboarding_status_use_case.dart
            set_onboarding_status_use_case.dart
         support/
            get_support_resources_use_case.dart
         symptom/
            add_log_symptom_use_case.dart
            get_symptom_history_use_case.dart
      util/                        # Domain utilities
         result.dart               # Result wrapper (Success/Failure)
         failure.dart              # Error handling
      enum/                        # Enumerations

   presentation/                   # Presentation Layer
      screen/                      # App screens
         splash/
            splash_screen.dart
            notifier/
               splash_notifier.dart
            state/
               splash_ui_state.dart
               splash_ui_state.freezed.dart
         onboarding/
            onboarding_screen.dart
         auth/
            notifier/
               auth_notifier.dart
            state/
               auth_ui_state.dart
               auth_ui_state.freezed.dart
         main/
            main_screen.dart
            notifier/
               tab_index_notifier.dart
         home/
            home_screen.dart
            widget/
         symptom/
            symptom_tracker_screen.dart
            symptom_history_screen.dart
            notifier/
               symptom_notifier.dart
```

```
        state/
            symptom_ui_state.dart
            symptom_ui_state.freezed.dart
    education/
        education_hub_screen.dart
        notifier/
           education_notifier.dart
        state/
            education_ui_state.dart
            education_ui_state.freezed.dart
    support/
        support_resources_screen.dart
        notifier/
           support_notifier.dart
        state/
            support_ui_state.dart
            support_ui_state.freezed.dart
    my_health/
        my_health_screen.dart
common/                         # Shared widgets
    widget/
    notifier/
    state/
dialog/                         # Dialog widgets
theme/                          # App theming
    app_theme.dart
util/                           # Presentation utilities
    routes.dart                 # Route definitions
```

API Documentation

Base URL

`http://10.0.2.2:8000/v1/`

API Modules

1. Authentication API **File**: auth_api_impl.dart

Register Device

- **Endpoint**: `POST /auth/device`
- **Description**: Registers a new device and returns authentication token
- **Request Body**:

```
{
  "device_id": "string",
  "device_type": "string",
  "device_name": "string"
}
```

- **Response**:

```
{
  "token": "string",
  "device_id": "string",
  "created_at": "string"
}
```

- **Implementation**:

```
Future<Auth> registerDevice({required RegisterRequest requestBody})
```

---

2. Symptom Tracking API **File**: symptom_api_impl.dart

Get Symptom History

- **Endpoint**: `GET /symptoms`
- **Description**: Retrieves symptom history for a device
- **Headers**:
  - `X-Device-Id`: Device identifier
- **Response**:

```
{
  "data": [
    {
      "id": "string",
      "symptom_type": "string",
      "severity": "string",
      "notes": "string",
      "logged_at": "string"
    }
  ]
}
```

- **Implementation**:

```
Future<List<SymptomLog>> getSymptomHistory({required String xDeviceId})
```

Add Symptom Log

- **Endpoint**: `POST /symptoms`
- **Description**: Logs a new symptom entry
- **Request Body**:

```
{
  "symptom_type": "string",
  "severity": "string",
  "notes": "string",
  "logged_at": "string"
}
```

- **Response**:

```
{
  "data": {
    "id": "string",
    "symptom_type": "string",
    "severity": "string",
    "notes": "string",
    "logged_at": "string"
  }
}
```

- **Implementation**:

```
Future<SymptomLog> addLogSymptom({required SymptomLogRequest requestBody})
```

---

3. Education API **File**: education_api_impl.dart

Get Education Articles

- **Endpoint**: `GET /education/articles`
- **Description**: Fetches educational articles and resources
- **Response**:

```
{
  "data": [
    {
      "id": "string",
      "title": "string",
      "content": "string",
      "category": "string",
      "image_url": "string",
      "created_at": "string"
    }
  ]
}
```

- **Implementation**:

  ```
  Future<List<EducationHub>> getEducationArticles()
  ```

---

4. Support Resources API **File**: support_api_impl.dart

Get Support Resources

- **Endpoint**: `GET /support/resources`
- **Description**: Retrieves support resources (hotlines, organizations, etc.)
- **Response**:

  ```json
  {
    "data": [
      {
        "id": "string",
        "name": "string",
        "description": "string",
        "contact": "string",
        "type": "string"
      }
    ]
  }
  ```

- **Implementation**:

  ```
  Future<List<Support>> getSupportResources()
  ```

API Client Architecture

File: api_client.dart

The `ApiClient` class wraps Dio and provides:

HTTP Methods

- `get<T, R>()` - GET requests
- `post<T, R>()` - POST requests
- `put<T, R>()` - PUT requests
- `putMultipart<T, R>()` - PUT with multipart form data
- `patch<T, R>()` - PATCH requests

- `delete<T, R>()` - DELETE requests
- `head<T, R>()` - HEAD requests

Features

- **Type-safe converters**: Convert JSON to DTOs
- **Error mapping**: Converts Dio exceptions to domain `Failure` objects
- **Interceptors**:
  - `AuthInterceptor`: Adds authentication tokens to requests
  - `LoggingInterceptor`: Logs HTTP requests/responses

Example Usage

```
final response = await _client.post<JSONObject, SymptomLogResponse>(
  path: 'symptoms',
  data: requestBody.toJson(),
  converter: (json) => SymptomLogResponse.fromJson(json),
);
return response.data.toDomain();
```

State Management

SheAware uses **Riverpod** with **StateNotifier** and **Freezed** for immutable state management.

State Management Pattern

```
graph LR
    A[UI Widget] -->|User Action| B[Notifier]
    B -->|Calls| C[Use Case]
    C -->|Executes| D[Repository]
    D -->|Returns| E[Result Success/Failure]
    E -->|Updates| B
    B -->|Emits| F[UI State]
    F -->|Rebuilds| A

    style A fill:#e1f5ff
    style B fill:#ffe1e1
    style C fill:#fff4e1
    style F fill:#e1ffe1
```

State Management Components

1. UI States (Freezed) UI states are immutable sealed classes representing different screen states:

Example: symptom_ui_state.dart

```
@freezed
class SymptomUiState with _$SymptomUiState {
  const factory SymptomUiState.initial() = InitialState;
  const factory SymptomUiState.loading() = LoadingState;
  const factory SymptomUiState.successLogSymptom({
    required SymptomLog logSymptom,
  }) = SuccessLogSymptomState;
  const factory SymptomUiState.successSymptomHistory({
    required List<SymptomLog> symptomHistory,
  }) = SuccessSymptomHistoryState;
  const factory SymptomUiState.error(String message) = ErrorState;
}
```

States Explained: - `initial`: Default/idle state - `loading`: Data fetching in progress - `successLogSymptom`: Successfully logged a symptom - `successSymptomHistory`: Successfully fetched symptom history - `error`: Error occurred with message

2. Notifiers (StateNotifier) Notifiers manage state transitions and business logic:

Example: symptom_notifier.dart

```
class SymptomNotifier extends StateNotifier<SymptomUiState> {
  SymptomNotifier() : super(const SymptomUiState.loading()) {
    getSymptomHistory(); // Auto-fetch on initialization
  }

  Future<void> getSymptomHistory() async {
    state = const SymptomUiState.loading();

    final xDeviceId = getIt<AuthLocalDataSource>().getDeviceId();

    try {
      final useCase = getIt<GetSymptomHistoryUseCase>();
      final result = await useCase(xDeviceId: xDeviceId);

      state = result.when(
        success: (symptomHistory) {
          return SymptomUiState.successSymptomHistory(
            symptomHistory: symptomHistory,
          );
        },
        failure: (failure) {
          return SymptomUiState.error(failure.message);
        },
      );
    } catch (e) {
      state = SymptomUiState.error(e.toString());
    }
  }

  Future<void> addLogSymptom({required SymptomLogRequest requestBody}) async {
    state = const SymptomUiState.loading();

    try {
      final useCase = getIt<AddLogSymptomUseCase>();
      final result = await useCase(requestBody: requestBody);

      state = result.when(
        success: (logSymptom) {
          return SymptomUiState.successLogSymptom(logSymptom: logSymptom);
        },
        failure: (failure) {
          return SymptomUiState.error(failure.message);
        },
      );
    } catch (e) {
      state = SymptomUiState.error(e.toString());
    }
  }
}
```

Key Features: - Extends `StateNotifier<SymptomUiState>` - Uses GetIt for dependency injection - Handles loading, success, and error states - Calls use cases for business logic

3. Provider Registration Providers are typically registered in screens:

```
final symptomNotifierProvider = StateNotifierProvider<SymptomNotifier, SymptomUiState>(
  (ref) => SymptomNotifier(),
);
```

4. UI Consumption Widgets consume state using `ref.watch()` or `ref.listen()`:

```
class SymptomHistoryScreen extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final state = ref.watch(symptomNotifierProvider);

    return state.when(
      initial: () => SizedBox.shrink(),
      loading: () => CircularProgressIndicator(),
      successSymptomHistory: (history) => ListView.builder(
        itemCount: history.length,
        itemBuilder: (context, index) => SymptomCard(history[index]),
      ),
      successLogSymptom: (log) => SuccessMessage(),
      error: (message) => ErrorWidget(message),
    );
  }
}
```

State Management Best Practices

1. **Immutability**: All states are immutable using Freezed
2. **Single Source of Truth**: Notifiers hold the single state
3. **Separation of Concerns**: UI doesn't contain business logic
4. **Error Handling**: Consistent error state across all features
5. **Loading States**: Explicit loading states for better UX
6. **Auto-disposal**: Riverpod auto-disposes providers when not needed

---

Application Flow

1. App Initialization Flow

```
sequenceDiagram
    participant Main
    participant DI as Dependency Injection
    participant App as SheAware App
    participant Splash as Splash Screen

    Main->>DI: setup()
    DI->>DI: setUpCacheModule()
    DI->>DI: setUpNetworkModule()
    DI->>DI: setUpDataSourceModule()
    DI->>DI: setUpRepositoryModule()
    DI->>DI: setUpUseCaseModule()
    DI->>Main: Ready
    Main->>App: runApp(ProviderScope)
    App->>Splash: Navigate to Splash
```

File: main.dart

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await di.setup(); // Initialize all dependencies
  await ScreenUtil.ensureScreenSize();

  runApp(
    const ProviderScope(child: SheAware()),
  );
}
```

2. Splash Screen Flow

```
sequenceDiagram
    participant Splash as Splash Screen
    participant Notifier as SplashNotifier
    participant Auth as CheckAuthStatusUseCase
    participant Onboard as CheckOnboardingStatusUseCase
    participant Register as RegisterDeviceUseCase

    Splash->>Notifier: Initialize
    Notifier->>Auth: Check if authenticated

    alt Is Authenticated
        Auth-->>Notifier: true
        Notifier->>Splash: Navigate to Main Screen
    else Not Authenticated
        Auth-->>Notifier: false
        Notifier->>Onboard: Check onboarding status

        alt Onboarding Seen
            Onboard-->>Notifier: true
            Notifier->>Register: Register device
            Register-->>Notifier: Success/Failure
            Notifier->>Splash: Navigate to Main Screen
        else Onboarding Not Seen
            Onboard-->>Notifier: false
            Notifier->>Splash: Navigate to Onboarding
        end
    end
```

File: splash_notifier.dart

3. Authentication Flow

```
sequenceDiagram
    participant UI as Auth Screen
    participant Notifier as AuthNotifier
    participant UseCase as RegisterUseCase
    participant Repo as AuthRepository
    participant Remote as AuthRemoteDataSource
    participant API as AuthApi
    participant Local as AuthLocalDataSource

    UI->>Notifier: registerDevice(requestBody)
    Notifier->>Notifier: Set loading state
    Notifier->>UseCase: call(requestBody)
    UseCase->>Repo: registerDevice(requestBody)
    Repo->>Remote: registerDevice(requestBody)
    Remote->>API: registerDevice(requestBody)
    API->>API: POST /auth/device

    alt Success
        API-->>Remote: AuthResponse
        Remote-->>Repo: Auth (domain model)
        Repo-->>UseCase: Result.success(auth)
        UseCase-->>Notifier: Result.success(auth)
        Notifier->>Local: Save tokens
        Notifier->>Notifier: Set success state
        Notifier-->>UI: AuthUiState.success
        UI->>UI: Navigate to Main Screen
    else Failure
        API-->>Remote: Error
        Remote-->>Repo: Failure
        Repo-->>UseCase: Result.failure(failure)
        UseCase-->>Notifier: Result.failure(failure)
        Notifier->>Notifier: Set error state
        Notifier-->>UI: AuthUiState.error
        UI->>UI: Show error message
    end
```

4. Symptom Tracking Flow

```
sequenceDiagram
    participant UI as Symptom Screen
    participant Notifier as SymptomNotifier
    participant UseCase as AddLogSymptomUseCase
    participant Repo as SymptomRepository
    participant Remote as SymptomRemoteDataSource
    participant API as SymptomApi

    UI->>Notifier: addLogSymptom(requestBody)
    Notifier->>Notifier: Set loading state
    Notifier->>UseCase: call(requestBody)
    UseCase->>Repo: addLogSymptom(requestBody)
    Repo->>Remote: addLogSymptom(requestBody)
    Remote->>API: addLogSymptom(requestBody)
    API->>API: POST /symptoms

    alt Success
        API-->>Remote: SymptomLogResponse
        Remote-->>Repo: SymptomLog (domain)
        Repo-->>UseCase: SymptomLog
        UseCase-->>Notifier: Result.success(log)
        Notifier->>Notifier: Set success state
        Notifier-->>UI: SymptomUiState.successLogSymptom
        UI->>UI: Show success message
        UI->>Notifier: getSymptomHistory()
    else Failure
        API-->>Remote: Error
        Remote-->>Repo: Failure
        Repo-->>UseCase: Failure
        UseCase-->>Notifier: Result.failure(failure)
        Notifier->>Notifier: Set error state
        Notifier-->>UI: SymptomUiState.error
        UI->>UI: Show error message
    end
```

5. Navigation Flow

File: routes.dart

```
class Routes {
  static const String splash = 'splash';
  static const String onboarding = 'onboarding';
  static const String main = 'main';
  static const String home = 'home';
  static const String symptomTracker = 'symptomTracker';
  static const String symptomHistory = 'symptomHistory';
  static const String myHealth = 'myHealth';
  static const String educationHub = 'educationHub';
  static const String supportResources = 'supportResources';
}
```

Navigation Graph:

```
Splash Screen
     > Onboarding Screen (if first time)
     > Main Screen (if authenticated or device registered)
             > Home Screen
             > Symptom Tracker Screen
             > Symptom History Screen
             > My Health Screen
```

```
                > Education Hub Screen
                > Support Resources Screen
```

Key Features

1. Dependency Injection with GetIt

Setup Order (injection_container.dart):

```
Future<void> setup() async {
  await setUpCacheModule();         // 1. SharedPreferences
  await getIt.allReady();           // 2. Wait for async registrations
  await setUpNetworkModule();       // 3. Dio, API clients, interceptors
  await setUpDataSourceModule();    // 4. Local & remote data sources
  await setUpRepositoryModule();    // 5. Repository implementations
  await setUpUseCaseModule();       // 6. Use cases
  await setUpServiceModule();       // 7. Additional services
}
```

2. Error Handling

Result Pattern (result.dart):

```
@freezed
class Result<T> with _$Result<T> {
  const factory Result.success(T value) = SuccessResult;
  const factory Result.failure(Failure failure) = FailureResult;
}
```

Usage in Use Cases:

```
Future<Result<SymptomLog>> call({required SymptomLogRequest requestBody}) async {
  return await _symptomRepository
      .addLogSymptom(requestBody: requestBody)
      .then((value) => Result.success(value))
      .onError((Failure failure, stackTrace) => Result.failure(failure));
}
```

3. Interceptors

Auth Interceptor

- Automatically adds authentication tokens to requests
- Retrieves tokens from `AuthLocalDataSource`

Logging Interceptor

- Logs all HTTP requests and responses

- Useful for debugging API calls

4. Code Generation

Freezed: Generates immutable classes with: - `copyWith()` methods - `toString()`, `==`, `hashCode` - Union types for states

JSON Serializable: Generates: - `fromJson()` constructors - `toJson()` methods

5. Local Storage

Uses `SharedPreferences` for: - Authentication tokens - Device ID - Onboarding status - User preferences

---

Summary

SheAware is a well-architected Flutter application following Clean Architecture principles:

Clean Architecture: Clear separation of data, domain, and presentation layers
Dependency Injection: Modular DI with GetIt
State Management: Riverpod + StateNotifier + Freezed
Type Safety: Freezed for immutable states, JSON serialization
Error Handling: Result pattern with Success/Failure
API Integration: Dio with interceptors for auth and logging
Scalability: Modular structure allows easy feature additions

This architecture ensures: - **Testability**: Each layer can be tested independently - **Maintainability**: Clear boundaries and responsibilities - **Scalability**: Easy to add new features without affecting existing code - **Reusability**: Domain models and use cases are framework-agnostic