0. Overview

DirectoryTree

```
project_root/
├── lib/
|   ├── common/
|   |   ├── utils/
|   |   ├── constants.dart
|   |   └── extensions.dart
|   |
|   ├── model/
|   |   ├── user_model.dart
|   |   ├── auth_response.dart
|   |   └── domain/
|   |
|   ├── view/
|   |   ├── screen/
|   |   |   ├── home_screen.dart
|   |   |   ├── login_screen.dart
|   |   |   └── settings_screen.dart
|   |   |
|   |   └── widget/
|   |       ├── custom_button.dart
|   |       ├── dialog/
|   |       └── card_item.dart
|   |
|   ├── viewmodel/
|   |   ├── auth_viewmodel.dart
|   |   ├── settings_viewmodel.dart
|   |   └── provider/
|   |
|   ├── config.dart
|   ├── router.dart
|   ├── router.g.dart     # GENERATED - DO NOT EDIT
|   └── main.dart
|
├── test/
|   ├── widget_test.dart
|   └── viewmodel_test.dart
|
├── .flutter-plugins-dependencies
├── .gitignore
├── .metadata
├── analysis_options.yaml
├── pubspec.yaml
└── pubspec.lock

```

`lib/`

Flutter 애플리케이션의 주요 소스 코드가 위치하는 디렉터리이다.
Dart 컴파일러는 기본적으로 이 경로를 entry point로 인식한다.

`lib/common/`

애플리케이션 전반에서 공통적으로 사용되는 모듈을 배치한다.
예:

- 유틸리티 함수
- 공용 상수
- 예외 처리 헬퍼
- 확장(extension) 모듈

`lib/model/`

데이터 모델 및 엔티티 클래스가 포함된다.
일반적으로 다음이 포함된다:

- API 응답 모델
- 상태(state) 데이터 구조
- DTO / Domain 모델

`lib/view/`

UI 계층(View layer)이 위치한다.

`lib/view/screen/`

각 화면(Screen)을 구성하는 StatefulWidget 또는 StatelessWidget 파일들을 포함한다.
페이지 단위 UI가 이 위치에서 정의된다.

`lib/view/widget/`

재사용 가능한 UI 컴포넌트를 포함한다.
예:

- 버튼
- 카드
- Dialog
- Custom widget

`lib/viewmodel/`

상태 관리 로직(ViewModel)이 위치한다.
Riverpod, Provider, Bloc 등 상태 관리 기술과 관련된 로직이 여기에 포함된다.

애플리케이션의 UI와 비즈니스 로직을 분리하기 위한 MVVM 구조에 적합하다.

`lib/config.dart`

전역 설정(Configuration) 정보를 제공하는 모듈이다.
예:

- debug 여부
- 빌드 환경 정보
- 앱 전반에서 사용되는 설정값

`lib/main.dart`

애플리케이션의 진입점(entry point)이다.
다음 요소들이 포함된다:

- Flutter 엔진 초기화
- Localization 설정
- ProviderScope 설정
- 화면 회전, UI 모드 설정
- `runApp()` 호출

`lib/router.dart`

라우팅 구성을 정의한다.
GoRouter를 사용하여 화면 이동 규칙을 선언적 방식으로 작성한다.

`lib/router.g.dart`

Riverpod code generator가 생성한 파일이다.
Provider metadata 및 자동 등록 코드가 포함되며, 수동 편집은 금지된다.
(`// GENERATED CODE - DO NOT MODIFY BY HAND`)

`test/`

단위 테스트 및 위젯 테스트 파일이 포함된다.
`flutter test` 명령을 통해 실행된다.

`.flutter-plugins-dependencies`

플러그인 의존성 관리에 사용되는 내부 파일이다.
수동 편집은 금지되며, Flutter가 자동 생성한다.

`.gitignore`

Git 버전 관리에서 제외할 파일들을 정의한다.

`.metadata`

Flutter 프로젝트 내부 관리용 파일이다.
프로젝트 생성 당시 Flutter 버전 등의 정보가 기록된다.

`analysis_options.yaml`

Dart analyzer 규칙을 정의하는 Lint 설정 파일이다.
코드 품질 및 규칙을 프로젝트 단위로 제어할 수 있다.

`pubspec.yaml`

Flutter 프로젝트의 핵심 설정 파일이다.
의존성(dependencies), asset 경로, 환경 설정 등을 포함한다.

`pubspec.lock`

현재 설치된 패키지 버전 정보를 고정(lock)하여
모든 개발 환경에서 동일 버전의 패키지를 사용할 수 있게 한다.

Flutter 코드 분석 절차(권장 순서)

1. 프로젝트 구조 식별

분석 초기 단계에서는 프로젝트의 폴더 구조와 주요 구성 요소를 파악한다.

- `lib/` 디렉터리의 구성
- `pubspec.yaml`의 의존성 목록
- 플랫폼 관련 코드(`android/`, `ios/`, `web/`, `windows/`) 포함 여부
- 빌드/런처 파일(`main.dart`)의 위치

> 목적: 의존성, 모듈 배치, 플랫폼 기능 포함 여부를 기반으로 전체적인 아키텍처를 파악한다.

---

2. 진입 지점(Entry Point) 확인

Flutter 프로젝트의 실행 흐름은 `main.dart`에서 시작한다.

- `main()` 함수
- `runApp()` 호출
- 최상위 위젯(App root widget)
- DI(Container), Provider, state-management 초기화 코드

> 목적: 앱의 초기화 경로를 명확히 하고 글로벌 상태나 초기 설정을 이해한다.

---

3. 상태 관리(State Management) 구조 파악

Flutter 프로젝트의 복잡도는 상태 관리 방식에 따라 달라진다.

다음 요소들을 우선적으로 분석한다.

- Provider, Riverpod, Bloc, MobX, GetX 등 사용 여부
- 상태 주입 방식(Provider 구조, Bloc 구조, Riverpod Provider Graph 등)
- 전역 상태와 지역 상태의 구분
- 각 state가 어떤 UI 또는 Service와 연결되는지

> 목적: 앱의 작동 원리는 **상태 변화 → UI 업데이트** 흐름으로 구성되므로 상태 구조를 먼저 이해해야 논리 흐름 파악이 가능하다.

---

4. 라우팅 및 화면 전환 구조 파악

다음 요소들을 검토한다.

- 라우팅 방식(Navigator 1.0, Router 2.0, go_router 등)
- route table 또는 route configuration
- 로그인 → 홈 → 상세 화면 등 주요 화면 흐름

> 목적: UI 화면 구조를 계층화하면 기능 단위 분석이 수월해진다.

---

5. 핵심 서비스 모듈 분석

Flutter는 UI 계층과 비 UI 계층(Service/Repository)이 분리된 경우가 많다.
다음 항목을 모듈 단위로 분석한다.

- Repository (서버/DB/로컬 스토리지)
- API 클라이언트(Dio, http 등)
- BLE, Serial, Sensor 등 플랫폼 채널 연동 코드(MethodChannel)
- 데이터 모델(JSON ⇄ Dart model)
- 비즈니스 로직(Provider/Bloc/Riverpod 내부 로직)

> 목적: 앱의 실질적인 동작(flow)은 서비스/로직 레이어에서 결정된다.

6. UI 위젯 계층 분석

UI 분석 시 다음 순서를 따른다.

- 가장 상위 Screen(Page) 위젯
- 해당 화면이 사용하는 ViewModel, Bloc, Provider
- 화면 구성 요소(View)
- 이벤트 처리(onPressed, onChanged 등)
- UI가 참조하는 상태 변수(State)의 변화 경로

> 목적: 기능이 UI에 어떻게 표현되고 동작 이벤트가 상태 관리로 전달되는지 확인한다.

---

7. 플랫폼 종속 코드 확인(필요 시)

Flutter 앱이 플랫폼 기능을 활용하는 경우 다음을 직접 확인한다.

- `android/` Kotlin 코드
- `ios/` Swift/Objective-C 코드
- MethodChannel, EventChannel 구현체
- Permissions 관련 처리

> 목적: Flutter UI 영역에서 호출되는 Native 기능의 동작 방식을 이해한다.

---

8. 유틸리티 및 보조 코드 검토

가장 마지막에 읽어야 하는 구성 요소들이다.

- validator, formatter
- 공용 util 함수
- theme/style
- 공용 위젯(components)
- extensions

> 목적: 전체 흐름을 이해한 뒤 세부 로직 보조 요소를 확인한다.

전체 요약(권장 분석 순서)

1. **프로젝트 구조 파악** (`pubspec.yaml`, lib 구조)
2. **main.dart → runApp → RootWidget**
3. **상태 관리 구조 분석** (Provider/Bloc/Riverpod 등)
4. **라우팅 구조 분석** (go_router 등)
5. **서비스/Repository/Model 분석**
6. **화면(UI) 구조 분석**

7. **플랫폼 네이티브 코드 분석(필요 시)**
8. **유틸리티 및 공용 컴포넌트 분석**