Project Architecture & Developer Onboarding Guide

Clean Architecture · Modular · Feature-Based

---

1. Architecture Overview

The project follows a **Clean Architecture + Modular (Feature-based)** approach.
This ensures:

- High scalability
- Easy onboarding for new developers
- Minimal chance of breaking unrelated modules
- Independent feature development
- Easier testing

---

2. High-Level Module Structure

```
/lib
  /core
    /constants
    /utils
    /network
    /services
    /error_handling
  /features
    /auth
    /profile
    /jobs
```

```
    /payments
    /notifications
  /shared
    /widgets
    /theme
    /components
    /routes
```

---

3. Clean Architecture Per Feature

Each feature will follow this structure:

```
/features/feature_name
  /data
    /models
    /datasources
    /repositories_impl
  /domain
    /entities
    /repositories
    /usecases
  /presentation
    /screens
    /providers or blocs
    /widgets
```

Example: `/features/auth`

```
auth/
  data/
    models/
    datasources/
      auth_remote_ds.dart
    repositories_impl/
      auth_repo_impl.dart

  domain/
    entities/
      user_entity.dart
    repositories/
      auth_repo.dart
```

```
    usecases/
      login.dart
      register.dart

  presentation/
    screens/
      login_screen.dart
      register_screen.dart
    providers/
      auth_provider.dart
```

---

4. Responsibilities Per Layer

Domain Layer

- Pure business logic
- Framework-independent
- Contains entities, abstract repositories, and use cases

Data Layer

- Talks to APIs, databases, Firebase, etc
- Converts raw data → domain models
- Implements repositories

Presentation Layer

- UI
- State management (Provider/BLoC)
- Screen-level widgets
- Calls use cases from domain

---

5. Shared Layer (Core)

`/core` contains things used across the whole project:

- Exceptions
- Network handling
- Base classes
- Theme
- Constants
- Response formats
- Global helpers

---

6. Git Workflow & Commit Conventions

Branches

```
main      // stable production code
develop   // active development
feature/<feature_name>
hotfix/<bug_name>
```

Commit Message Style (Conventional Commits)

```
feat(auth): added login usecase
feat(profile): implemented update profile API

fix(auth): resolved token refresh crash

refactor(core): cleaned error handling structure

chore(ci): added linting config
```

---

7. Feature Development Flow

When adding a new feature (example: Job Listings)

Step 1: Create a Feature Branch

```
git checkout -b feature/jobs
```

Step 2: Add Domain Layer

```
feat(jobs-domain): added job entity and usecases
```

Step 3: Add Data Layer

```
feat(jobs-data): implemented remote datasource & repo impl
```

Step 4: Add Presentation + UI

```
feat(jobs-ui): added job list screen & provider
```

Step 5: Integrate Navigation

```
feat(routes): added route for job listing screen
```

Step 6: Write Tests (if applicable)

```
test(jobs): added unit tests for repo & usecases
```

Step 7: Raise PR → `develop`

PR should include:

- What's added
- Testing steps
- Risks
