Flutter Clean Architecture

Per-Feature Structure & Best Practices

This guide provides an in-depth explanation of implementing Clean Architecture in Flutter using a feature-first approach. You will learn about architectural layers, how to structure your project, separation of concerns, dependency inversion, and how to scale your app efficiently.

Flutter Clean Architecture: Per-Feature Structure

Table of Contents

1. Introduction

Flutter Clean Architecture helps you build scalable, testable, and maintainable apps. By dividing code into layers and organizing it per feature, developers can isolate concerns, improve productivity, and enable easier testing and scaling.

2. Clean Architecture Principles

1. Separation of concerns: UI, domain logic, and data access are kept separate.
2. Dependency inversion: High-level modules do not depend on low-level modules; both depend on abstractions.
3. Testability: Each layer is independently testable.

4. Framework independence: Business logic is not tied to Flutter or external libraries.

3. Folder Structure Overview

```
lib/
 core/
 features/
   auth/
      data/
      domain/
      presentation/
 injection_container.dart
 main.dart
```

4. Layer-by-Layer Breakdown

Presentation Layer

Handles UI and state management. Includes pages, widgets, and BLoC/Cubit logic. No business logic.

Domain Layer

Pure business logic. Includes entities, use cases, and abstract repositories. Framework-agnostic.

Data Layer

Responsible for data fetching and persistence. Includes models, data sources, and repository implementations.

Core Layer

Holds reusable utilities such as error handling, constants, theming, and network helpers.

Injection Layer

Flutter Clean Architecture: Per-Feature Structure

Manages dependency injection using tools like GetIt or Riverpod.

5. Dependency Injection

Use GetIt, Riverpod, or other DI tools to manage your dependencies. Create a central file like `injection_container.dart` to register repositories, data sources, use cases, and BLoCs or Cubits. Use lazySingleton or singleton patterns for efficiency.

6. Real Feature Example: Authentication

Example structure for the 'auth' feature:

- domain/entities/user.dart
- domain/usecases/login_user.dart
- domain/repositories/auth_repository.dart
- data/models/user_model.dart
- data/datasources/auth_remote_data_source.dart
- data/repositories/auth_repository_impl.dart
- presentation/bloc/auth_bloc.dart
- presentation/pages/login_page.dart
- presentation/widgets/login_form.dart

7. Testing in Clean Architecture

Test each layer independently:

- Unit test UseCases and Entities in domain layer.
- Mock repositories for testing business logic.
- Widget tests in presentation layer.
- Integration tests for end-to-end functionality.

8. Best Practices & Tips

Flutter Clean Architecture: Per-Feature Structure

- Keep layers independent.
- Use DTOs for transforming between layers.
- Minimize logic in UI.
- Use naming conventions consistently.
- Document your code and architecture.
- Keep feature folders isolated.

9. Summary

By using Clean Architecture with per-feature organization, you ensure that your Flutter app is easy to test, scale, and maintain. Always enforce layer boundaries and use dependency injection to manage your object graph efficiently.