Flutter Clean Architecture Project Structure

Domain-driven design with BLoC state management. Three-layer architecture for complex, maintainable apps.

#flutter #dart #mobile #clean-architecture #bloc #domain-driven

PNG PDF Copy Prompt

Project Directory

```
myapp/
  pubspec.yaml  Dependencies an…
  pubspec.lock
  analysis_options.yaml  Strict linting
  .env  Environment var…
  .env.example
  .gitignore
  README.md
  lib/
    main.dart  Entry point and…
    injection_container.dart  Dependency inje…
    core/  Shared infrastr…
      error/
        failures.dart  Failure classes
        exceptions.dart
      usecases/
        usecase.dart  Base UseCase cl…
      network/
        network_info.dart
      utils/
        input_converter.dart
    features/  Feature modules
      auth/
        domain/  Business logic …
          entities/
            user.dart  Pure domain ent…
          repositories/
            auth_repository.dart  Abstract contra…
          usecases/
            login.dart
            register.dart
            logout.dart
        data/  Data layer
          models/
            user_model.dart  DTO with fromJs…
          datasources/
            auth_remote_datasource.dart
            auth_local_datasource.dart
          repositories/
            auth_repository_impl.dart  Implements doma…
        presentation/  UI layer
          bloc/
            auth_bloc.dart
            auth_event.dart
            auth_state.dart
          pages/
            login_page.dart
            register_page.dart
          widgets/
            auth_form.dart
      home/
        domain/
          entities/
          repositories/
          usecases/
        data/
          models/
          datasources/
          repositories/
        presentation/
          bloc/
          pages/
          widgets/
  test/
    features/
      auth/
        domain/
          usecases/
        data/
          repositories/
        presentation/
          bloc/
    core/
      utils_test.dart
    fixtures/  JSON test data
      fixture_reader.dart
      user.json
  android/
    app/
  ios/
    Runner/
  assets/
    images/
    fonts/
```

Why This Structure?

Clean Architecture separates your app into three layers: Domain (business rules), Data (APIs, databases), and Presentation (UI). Dependencies point inward—domain has no external dependencies. This makes the codebase testable, maintainable, and framework-agnostic.

Key Directories

`domain/entities/` - Pure business objects with no dependencies

`domain/usecases/` - Single-purpose business operations

`domain/repositories/` - Abstract contracts (interfaces)

`data/repositories/` - Concrete implementations of domain contracts

`presentation/bloc/` - BLoC for state management

UseCase Pattern

```dart
// domain/usecases/login.dart
class Login implements UseCase {
  final AuthRepository repository;
  Login(this.repository);

  @override
  Future> call(LoginParams params) {
    return repository.login(params.email, params.password)
  }
}
```

Getting Started

1. `flutter create myapp`
2. `flutter pub add flutter_bloc get_it dartz equatable`
3. Create layer folders: `domain/`, `data/`, `presentation/`
4. Set up `injection_container.dart` with GetIt
5. Create your first feature with all three layers

When To Use This

- Large-scale apps with complex business logic
- Teams of 3+ developers
- Apps requiring extensive testing
- Long-term projects (1+ year maintenance)
- When domain logic must be framework-independent

Trade-offs

`Verbose` - Lots of folders and boilerplate

`Steep learning curve` - Team must understand layer boundaries

`Overkill for simple apps` - Don't use for todo apps

Best Practices

- Domain layer has ZERO external package imports
- Use `Either` for error handling (dartz package)
- One UseCase = one business operation
- BLoC handles UI state, not business logic
- Test domain layer in isolation—mock repository interfaces

Testing Strategy

`Unit tests` - Test UseCases with mocked repositories

`Repository tests` - Test data layer with mocked datasources

`BLoC tests` - Use `bloc_test` for state transitions

`Fixtures` - JSON files for consistent test data