Flutter Mobile Programming: Professional

(Integrative-Generative AI Edition)

Clean Architecture
Testing
Performance Optimization
Application Security
Cross-platform
Publishing
Maintenance & Updates
Bibliography

by
Student Price Book Center

คำนำ

ในยุคที่เทคโนโลยีมือถือพัฒนาอย่างรวดเร็วและการใช้งานสมาร์ตโฟนกลายเป็นส่วนหนึ่งของวิถีชีวิตประจำวัน ความสามารถในการพัฒนาแอปพลิเคชันที่ตอบสนองต่อความต้องการของผู้ใช้ได้อย่างมีประสิทธิภาพถือเป็นทักษะสำคัญของนักพัฒนา การเรียนรู้และใช้ **Flutter** ซึ่งเป็น Framework พัฒนาแอปแบบ Cross-platform ที่ได้รับการสนับสนุนจาก Google จึงกลายเป็นโอกาสสำคัญสำหรับผู้ที่ต้องการสร้างสรรค์แอปคุณภาพสูงที่ทำงานได้ทั้งบน Android, iOS, Web และ Desktop ภายในฐานโค้ดเดียว

หนังสือ *Flutter Mobile Programming: Professional* เล่มนี้ ถูกออกแบบมาเพื่อพาผู้อ่านก้าวข้ามการเขียนโค้ดพื้นฐาน สู่การพัฒนาเชิงลึกที่เน้น **คุณภาพ ประสิทธิภาพ และความเป็นมืออาชีพ** เนื้อหาครอบคลุมทั้ง **สถาปัตยกรรมระบบ (Clean Architecture)** การทดสอบ (Testing) การปรับแต่งประสิทธิภาพ (Performance Optimization) ความปลอดภัย (Security) การขยายการใช้งานไปสู่ Cross-platform และ Desktop การเผยแพร่แอป (Application Uploading) และการดูแลหลังการปล่อยใช้งาน (App Maintenance & Updates)

จุดเด่นของหนังสือเล่มนี้

1. **ลงลึกโครงสร้างสถาปัตยกรรม (Architecture Deep Dive)**
   อธิบายวิธีออกแบบระบบด้วย *Clean Architecture* ผสานแนวทางการทำงานแบบ Dependency Injection (DI) และการจัดการ Service Locator เพื่อให้แอปมีความยืดหยุ่นและปรับขยายได้ในอนาคต พร้อมตัวอย่างการทำ Modular Architecture สำหรับโปรเจ็กต์ขนาดใหญ่ใน Flutter
2. **แนวทางการทดสอบที่ครบวงจร (Comprehensive Testing)**
   ครอบคลุมการทำ Unit Test, Widget Test และ Integration Test ใน Flutter รวมถึงการ Mock API และการใช้ Mockito เพื่อสร้างสภาพแวดล้อมการทดสอบที่เสมือนจริง พร้อมตัวอย่างโปรแกรมเพื่อประยุกต์ใช้ได้ทันที
3. **การปรับแต่งประสิทธิภาพ (Performance Optimization)**
   เน้นการใช้เครื่องมืออย่าง Flutter DevTools ตรวจสอบ Memory Leak ลดการ Rebuild Widgets ที่ไม่จำเป็น การปรับขนาด Bundle และการใช้ Isolate เพื่อจัดการงานเบื้องหลังอย่างมีประสิทธิภาพ
4. **การรักษาความปลอดภัย (Security Best Practices)**
   สอนวิธีใช้ *flutter_secure_storage* เก็บ Token และข้อมูลสำคัญอย่างปลอดภัย การเข้ารหัส (Encryption) การป้องกันการโจมตีผ่าน API ด้วย JWT และ HTTPS รวมถึงการทำ Code Obfuscation
5. **การพัฒนาแบบ Cross-platform และ Desktop**
   แนะนำการสร้างแอปสำหรับ Web และ Desktop (Windows, macOS) พร้อมการเลือกใช้ Plugin ให้รองรับทุกแพลตฟอร์ม เพื่อเพิ่มโอกาสการเข้าถึงผู้ใช้ในวงกว้าง

6. **การเผยแพร่และดูแลหลังการปล่อยใช้งาน**

   ครอบคลุมขั้นตอนการสร้างไฟล์ APK/AAB การทำ Code Signing การอัปโหลดขึ้น Google Play Store การจัดการเวอร์ชัน การทำ Build Flavors การใช้ Firebase Crashlytics การทำ In-App Update และการวางแผน Release Cycle ควบคู่ไปกับระบบ CI/CD

หนังสือเล่มนี้ไม่เพียงให้ความรู้เชิงเทคนิคเท่านั้น แต่ยังแฝงด้วยแนวคิดและประสบการณ์จริงจากโปรเจ็กต์หลากหลายประเภท เพื่อให้ผู้อ่านสามารถนำไปประยุกต์ใช้ในงานจริงได้ทันที แต่ละบทมาพร้อม **ตัวอย่างบูรณาการ (Integrated Examples)** ที่ออกแบบให้เห็นการทำงานของทุกองค์ประกอบในภาพรวม ซึ่งเป็นสิ่งจำเป็นสำหรับการพัฒนาแอประดับ Production

ไม่ว่าคุณจะเป็นนักพัฒนาที่ต้องการยกระดับจากระดับกลางสู่ระดับมืออาชีพ หรือเป็นหัวหน้าทีมพัฒนาที่ต้องการสร้างมาตรฐานงานให้มีคุณภาพสูง หนังสือเล่มนี้จะช่วยให้คุณเข้าใจทั้งเชิงเทคนิคและเชิงกลยุทธ์ในการพัฒนา Flutter อย่างแท้จริง และสามารถต่อยอดไปสู่การสร้างแอปที่มีคุณภาพ แข็งแรง ปลอดภัย และพร้อมรองรับผู้ใช้จำนวนมากในอนาคต

ผู้เขียนเชื่อว่าการเรียนรู้ที่ดีต้องไม่เพียงบอก *“วิธีทำ”* แต่ต้องบอก *“ทำไมต้องทำแบบนั้น”* ด้วย เพื่อให้ผู้อ่านสามารถปรับประยุกต์ได้อย่างอิสระ หนังสือเล่มนี้จึงมุ่งเน้นการให้เหตุผลเบื้องหลังการเลือกเทคนิคต่าง ๆ เพื่อให้คุณไม่เพียงทำตามตัวอย่าง แต่ยังสามารถออกแบบและแก้ปัญหาได้ด้วยตัวเองอย่างมั่นใจ

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 18
สถาปัตยกรรมแอป
(Clean Architecture)

เนื้อหา

- สถาปัตยกรรมแอป
- รายละเอียดเชิงลึก (deep dive)— สถาปัตยกรรมแอป
- Clean Architecture
- การทำ Dependency Injection (DI)
- การจัดการ Service Locator
- การออกแบบโมดูลสำหรับแอปขนาดใหญ่ (Modular Architecture for Large Flutter Apps)
- ตัวอย่างบูรณาการ

บทนำบทที่ 18: สถาปัตยกรรมแอป

การพัฒนาแอปพลิเคชันที่มีความซับซ้อนและขนาดใหญ่ต้องการ **สถาปัตยกรรมที่มั่นคงและยืดหยุ่น** บทนี้มุ่งเน้นการออกแบบแอป Flutter ด้วยแนวคิด **Clean Architecture** ซึ่งแบ่งแอปออกเป็นชั้น Presentation, Domain และ Data Layer เพื่อแยกความรับผิดชอบของแต่ละส่วน ทำให้โค้ดมีความชัดเจน อ่านง่าย และสามารถทดสอบได้ง่ายขึ้น

บทนี้ยังครอบคลุมการ **ทำ Dependency Injection** ด้วยเครื่องมือยอดนิยมอย่าง **get_it** และ **injectable** ซึ่งช่วยจัดการการสร้างและการเชื่อมโยงระหว่างอ็อบเจ็กต์อย่างมีประสิทธิภาพ ลดความซับซ้อนและเพิ่มความยืดหยุ่นในการจัดการโค้ด

นอกจากนี้ ผู้พัฒนาจะได้เรียนรู้การ **จัดการ Service Locator** เพื่อให้แต่ละชั้นของแอปสามารถเข้าถึงบริการหรือรีซอร์สที่จำเป็นได้โดยไม่ทำให้เกิดการเชื่อมโยงที่ซับซ้อนเกินไป ซึ่งช่วยให้การบำรุงรักษาและการขยายแอปในอนาคตทำได้ง่ายขึ้น

บทนี้ยังสอนการ **ออกแบบโมดูลสำหรับแอปขนาดใหญ่** โดยแนะนำวิธีแบ่งแอปเป็นโมดูลย่อยที่สามารถพัฒนา ทดสอบ และปรับปรุงแยกกันได้อย่างอิสระ ทำให้ทีมพัฒนาสามารถทำงานพร้อมกันได้หลายส่วนโดยไม่เกิดความขัดแย้ง และรองรับการขยายฟีเจอร์ใหม่ ๆ ได้อย่างมีประสิทธิภาพ

เมื่อศึกษาบทนี้จบ ผู้อ่านจะมีความเข้าใจเชิงลึกเกี่ยวกับ **สถาปัตยกรรมแอปใน Flutter** สามารถออกแบบแอปที่โครงสร้างมั่นคง ยืดหยุ่นต่อการเปลี่ยนแปลง และรองรับการขยายตัวในโปรเจกต์ขนาดใหญ่ได้อย่างมั่นใจ

สถาปัตยกรรมแอป

- Clean Architecture (Presentation, Domain, Data Layer)
- การทำ Dependency Injection (get_it, injectable)
- การจัดการ Service Locator
- การออกแบบโมดูลสำหรับแอปขนาดใหญ่

---

1. Clean Architecture ใน Flutter

Clean Architecture คือแนวคิดในการจัดโครงสร้างโค้ดให้แยกความรับผิดชอบ (Separation of Concerns) ออกเป็นเลเยอร์ที่ชัดเจน เพื่อให้โค้ดมีคุณสมบัติ

- **ง่ายต่อการทดสอบ (Testable)**
- **ง่ายต่อการบำรุงรักษา (Maintainable)**
- **ขยายต่อได้ (Scalable)**

ใน Flutter มักจะใช้การแบ่งเลเยอร์หลัก ๆ 3 ส่วน คือ

1.1 Presentation Layer

- ทำหน้าที่แสดง UI และรับ interaction จากผู้ใช้
- ใช้ State Management เช่น **Provider, Riverpod, Bloc, Cubit** เพื่อประสานงานกับ Domain Layer
- ไม่มี business logic อยู่ใน Widget โดยตรง
- ตัวอย่างไฟล์:
- /lib/presentation/
- home_page.dart
- home_viewmodel.dart

โค้ดตัวอย่าง:

```
class HomePage extends StatelessWidget {
  final HomeViewModel viewModel;

  const HomePage({super.key, required this.viewModel});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: StreamBuilder(
        stream: viewModel.dataStream,
```

```
        builder: (context, snapshot) {
          if (!snapshot.hasData) return CircularProgressIndicator();
          return Text(snapshot.data!);
        },
      ),
    );
  }
}
```

1.2 Domain Layer

- เก็บ **Business Rules** (กฎการทำงานของระบบ)
- ไม่พึ่งพา framework ใด ๆ
- ประกอบด้วย
    - **Entities** (โครงสร้างข้อมูลหลักของระบบ)
    - **Use Cases / Interactors** (การกระทำทางธุรกิจ เช่น GetUserProfile, UpdateCartItem)
    - **Repository Interfaces** (สัญญาว่าข้อมูลจะถูกดึง/บันทึกอย่างไร แต่ไม่สนใจวิธีการจริง)

ตัวอย่าง Repository Interface:

```
abstract class UserRepository {
  Future<User> getUserProfile();
}
```

1.3 Data Layer

- ทำหน้าที่ **จัดการข้อมูลจริง** (API, Local DB, Cache)
- Implement Repository จาก Domain Layer
- อาจแบ่งเป็น:
    - **Data Sources**
        - Remote Data Source (API)
        - Local Data Source (SQLite, Hive)
    - **Model** (Data Transfer Object / DTO)

ตัวอย่าง Implementation:

```
class UserRepositoryImpl implements UserRepository {
  final UserApi api;
```

```
  UserRepositoryImpl(this.api);

  @override
  Future<User> getUserProfile() async {
    final response = await api.fetchUser();
    return User(id: response.id, name: response.name);
  }
}
```

---

2. การทำ Dependency Injection (get_it, injectable)

Dependency Injection (DI) คือการส่ง object ที่ class ต้องใช้จากภายนอก แทนที่จะสร้างเองภายใน class เพื่อให้ทดสอบง่ายและปรับเปลี่ยนได้

2.1 ใช้ get_it

- get_it เป็น Service Locator ยอดนิยมใน Flutter
- ลง dependency:

```
dependencies:
  get_it: ^7.6.0
```

- ตั้งค่า:

```
final getIt = GetIt.instance;

void setupLocator() {
  getIt.registerLazySingleton<UserRepository>(() => UserRepositoryImpl(UserApi()));
}
```

- ใช้งาน:

```
final repo = getIt<UserRepository>();
```

2.2 ใช้ injectable

- ทำงานร่วมกับ get_it และ build_runner เพื่อ auto-generate DI code
- ลง dependency:

```
dependencies:
  get_it: ^7.6.0
  injectable: ^2.1.0
dev_dependencies:
```

build_runner: ^2.3.0

injectable_generator: ^2.1.0

- ใช้ Annotation:

```
@injectable
class UserRepositoryImpl implements UserRepository { ... }
```

- รัน generate:

```
flutter pub run build_runner build
```

---

3. การจัดการ Service Locator

Service Locator เป็นจุดศูนย์กลางที่เราสมัคร service ต่าง ๆ เช่น API client, Repository, Use Cases เพื่อให้สามารถดึงมาใช้ได้จากทุกที่

หลักการที่ดี:

- Register service เฉพาะที่จำเป็น
- แยก module ของ service locator ตาม feature
- หลีกเลี่ยงการใช้แบบ global ที่ไม่ควบคุม เพราะอาจทำให้ทดสอบยาก

---

4. การออกแบบโมดูลสำหรับแอปขนาดใหญ่

เมื่อแอปเริ่มใหญ่ขึ้น จำเป็นต้อง **Modularization** เพื่อ:

- ลดเวลา build
- แยกทีมทำงานตาม feature
- ทำให้ dependency management ง่ายขึ้น

โครงสร้างตัวอย่าง:

```
/lib
  /core          # Shared utilities, constants, themes
  /features
    /auth
      /data
      /domain
      /presentation
    /product
      /data
      /domain
      /presentation
  /di            # Dependency Injection setup
```

แนวทางออกแบบ:

1. **Feature-first** แทนการแยกตาม type (Widget, Model)
2. **แยก dependency** ให้แต่ละโมดูล import เฉพาะที่ต้องใช้
3. ใช้ **Clean Architecture** ภายในแต่ละโมดูลด้วย

ภาพรวมการเชื่อมโยง

[Presentation Layer] <--> [Domain Layer] <--> [Data Layer]

(UI, State) (Entities, UseCases) (API, DB)

DI และ Service Locator จะเป็นตัว **เชื่อมระหว่างเลเยอร์** ให้หลวม (loose coupling) และเปลี่ยน implementation ได้ง่าย

รายละเอียดเชิงลึก (deep dive)— สถาปัตยกรรมแอป

รายละเอียดเชิงลึก (deep dive) ของ **บทที่ 18 — สถาปัตยกรรมแอป** สำหรับ Flutter (Clean Architecture + DI + Service Locator + Modularization) — ครอบคลุมแนวคิด วิธีปฏิบัติที่ดี โค้ดตัวอย่าง และ pattern ที่ใช้ใน production เพื่อให้คุณสามารถออกแบบแอปขนาดกลาง⟶ใหญ่ ได้จริงและทดสอบได้

1 — หลักการสำคัญ (quick recap)

- แยกความรับผิดชอบ (Separation of Concerns) — แต่ละเลเยอร์มีหน้าที่เดียว
- เลเยอร์ต้อง “loose-coupled” และ “high-cohesion”
- Domain เป็น core — ต้องไม่พึ่งพา Flutter / 3rd-party libs
- ขยายได้ (Scalable), ทดสอบง่าย (Testable), เปลี่ยน implementation ได้โดยไม่กระทบ business rules

---

2 — โครงสร้างเลเยอร์ (เชิงลึก + ตัวอย่าง)

Presentation Layer (UI / State)

หน้าที่

- แสดงผล UI, ฟัง event ของผู้ใช้
- แปลงข้อมูลจาก Domain/UseCases ให้เป็น View State
- ไม่มี business logic เกินจำเป็น — logic อยู่ใน ViewModel / Controller

รายละเอียดปฏิบัติ

- ใช้ **constructor injection** เพื่อฉีด dependencies (avoid GetIt.get() ทุกที่)
- ใช้ State-management ที่เหมาะ: Riverpod/Bloc/Provider/StateNotifier (เลือกหนึ่งแบบในโปรเจกต์)

- ViewModel = thin orchestration layer (transform UseCase result ⟶ UI model)

ตัวอย่าง ViewModel (ChangeNotifier แบบเรียบ):

```
class ProductViewModel extends ChangeNotifier {
  final GetProductsUseCase _getProducts;

  List<Product> items = [];
  bool loading = false;
  String? error;

  ProductViewModel(this._getProducts);

  Future<void> load() async {
    loading = true; notifyListeners();
    final result = await _getProducts.call(page: 1);
    result.fold(
      (failure) {
        error = failure.message;
      },
      (products) {
        items = products;
      },
    );
    loading = false; notifyListeners();
  }
}
```

Domain Layer (core, pure Dart)

หน้าที่

- Entities / ValueObjects
- UseCases (หนึ่ง UseCase = หนึ่ง action ในธุรกิจ)
- Repository interfaces (contracts) — **no implementation here**
- โค้ดต้องไม่ import flutter หรือ dart:io ที่ผูก platform

ตัวอย่าง Entity + UseCase + Repository interface:

```
class Product {
  final String id;
```

```
  final String name;
  final double price;
  Product({required this.id, required this.name, required this.price});
}

abstract class ProductRepository {
  Future<Either<Failure, List<Product>>> getProducts({int page});
}

class GetProductsUseCase {
  final ProductRepository repository;
  GetProductsUseCase(this.repository);

  Future<Either<Failure, List<Product>>> call({int page = 1}) {
    return repository.getProducts(page: page);
  }
}
```

(แนะนำใช้ Either จาก package เช่น dartz/fpdart — ช่วยแยก success/failure แบบชัดเจน)

Data Layer (implementation)

หน้าที่

- RemoteDataSource (API client)
- LocalDataSource (DB / cache)
- DTOs, mappers (DTO <-> Entity)
- Implement ของ Repository interface

สำคัญ: RepositoryImpl เป็นจุดรวม logic เช่น

- ตัดสินใจแคช vs remote
- แปลง DTO ⟶ Entity
- จัดการ error mapping ⟶ Failure

ตัวอย่างโครงสร้าง repository implementation:

```
class ProductRepositoryImpl implements ProductRepository {
  final ProductRemoteDataSource remote;
  final ProductLocalDataSource local;
  final NetworkInfo networkInfo;
```

```
  ProductRepositoryImpl({
    required this.remote,
    required this.local,
    required this.networkInfo,
  });

  @override
  Future<Either<Failure, List<Product>>> getProducts({int page = 1}) async {
    if (await networkInfo.isConnected) {
      try {
        final dtos = await remote.fetchProducts(page);
        final entities = dtos.map((d) => d.toEntity()).toList();
        await local.cacheProducts(entities);
        return Right(entities);
      } on ServerException {
        return Left(ServerFailure('Server error'));
      }
    } else {
      try {
        final cached = await local.getCachedProducts();
        if (cached.isNotEmpty) return Right(cached);
        return Left(CacheFailure('No cached data'));
      } on CacheException {
        return Left(CacheFailure('Cache error'));
      }
    }
  }
}
```

3 — Error Handling & Result types

- สร้าง Failure hierarchy (ServerFailure, CacheFailure, ValidationFailure)
- Return Either<Failure, T> แทน throw exceptions ขึ้นมาจัดการที่ upper layer
- แปลง Exceptions ⟶ Failure ใน Data Layer

ตัวอย่าง Failure:

```
class Failure {
  final String message;
  Failure(this.message);
}

class ServerFailure extends Failure { ServerFailure(String message): super(message); }
class CacheFailure extends Failure { CacheFailure(String message): super(message); }
```

---

4 — Dependency Injection (get_it + injectable) — ขั้นตอนจริงและตัวอย่าง

ทำไมใช้ get_it + injectable?

- get_it = simple Service Locator/DI container
- injectable = annotation-based generator ที่สร้างการลงทะเบียนให้อัตโนมัติ (ลด boilerplate)

pubspec (สำคัญ)

```
dependencies:
  get_it: ^7.0.0
  injectable: ^2.0.0
  dartz: ^0.10.0
  dio: ^5.0.0
  connectivity_plus: ^4.0.0
  json_annotation: ^4.0.0

dev_dependencies:
  build_runner: ^2.4.0
  injectable_generator: ^2.0.0
  json_serializable: ^6.0.0
```

ขั้นตอนการตั้งค่า (ตัวอย่าง)

1. สร้างไฟล์ injection.dart:

```
import 'package:get_it/get_it.dart';
import 'package:injectable/injectable.dart';
import 'injection.config.dart';

final getIt = GetIt.instance;

@InjectableInit()
```

```
void configureDependencies(String env) => $initGetIt(getIt, environment: env);
```

2. Annotate classes:

```
@module
abstract class ExternalModule {
  @lazySingleton
  Dio dio() => Dio(BaseOptions(baseUrl: 'https://api.example.com'));

  @preResolve
  Future<SharedPreferences> prefs() => SharedPreferences.getInstance();
}

@LazySingleton(as: ProductRepository)
class ProductRepositoryImpl implements ProductRepository { ... }

@injectable
class GetProductsUseCase { ... }

@injectable
class ProductViewModel {
  final GetProductsUseCase getProducts;
  ProductViewModel(this.getProducts);
}
```

3. สั่ง generate:

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

4. เรียกใช้ตอนเริ่มแอป:

```
void main() {
  configureDependencies(Environment.prod);
  runApp(const MyApp());
}
```

คำอธิบายสำคัญเกี่ยวกับ scope

- registerLazySingleton / @lazySingleton — สร้างครั้งเดียวเมื่อเรียกใช้ครั้งแรก
- registerSingleton — สร้างทันที (eager)
- registerFactory — คืน instance ใหม่ทุกครั้ง (ViewModel แบบไม่คง state)
- preResolve — สำหรับ async init (SharedPreferences, DB connection)

5 — Service Locator: good & bad practices

Good

- ใช้ get_it เพื่อเก็บ service/infra level dependencies (Dio, DB, Repos)
- Inject into constructors (ให้ ViewModels รับ dependencies ผ่าน constructor)
- แยก registration เป็น modules (auth_module, product_module) — ช่วย lazy-load

Bad (anti-patterns)

- เรียก GetIt.I.get<T>() จากทุกที่ใน widget tree — ทำให้ test ยาก
- ปรับ state ผ่าน global singletons โดยตรง
- เก็บ mutable global state ใน service locator

Testing tip:

- ใน unit test: reset get_it ก่อน/หลัง:

```
setUp(() async {
  await getIt.reset();
  configureDependencies(Environment.test);
});
tearDown(() async {
  await getIt.reset();
});
```

6 — Modularization (ออกแบบแอปขนาดใหญ่)

สองแนวทางหลัก

- **Mono-repo (single repo, multi packages)** — แยกเป็น package ภายใน repo (/packages/core, /packages/features/product)
  - ข้อดี: การแชร์โค้ดง่าย, single CI, sync changes
  - ข้อเสีย: ต้องจัดการ dependency ระหว่าง package ให้ชัด
- **Multi-repo** — แต่ละ module repository แยกทีมเป็นเจ้าของ (useful บางองค์กร)

ตัวอย่างโครงสร้าง (feature-first + Clean Arch ภายในแต่ละ feature):

```
/lib
  /core
    /network
    /utils
    /di
  /features
```

```
/product
  /data
    product_remote_ds.dart
    product_local_ds.dart
    product_repository_impl.dart
    models/
  /domain
    entities/
    repositories/
    usecases/
  /presentation
    viewmodels/
    pages/
main.dart
```

แนวปฏิบัติ

- ให้ dependency flow เป็น one-way: presentation -> domain -> data
- core package เก็บ shared types (Failure, NetworkInfo, constants, theme)
- แต่ละ feature ควร expose only public API (ไม่ leak internal types)

Lazy feature loading

- ถ้าต้องการ lazy load module (ลด startup cost) — อย่าลงทะเบียน service ของ module ทั้งหมดตอน start — ให้ register ตอนเข้าหน้านั้นครั้งแรก

---

7 — ตัวอย่างการต่อสายทั้งหมด (minimal flow)

파일 structure (feature product):

features/product/domain/entities/product.dart

features/product/domain/repositories/product_repository.dart

features/product/domain/usecases/get_products.dart

features/product/data/datasources/product_remote_ds.dart

features/product/data/datasources/product_local_ds.dart

features/product/data/models/product_dto.dart

features/product/data/repositories/product_repository_impl.dart

features/product/presentation/product_viewmodel.dart

(ตัวอย่างโค้ดสำคัญอยู่ในหัวข้อก่อนหน้า — แนวทางคือ DTO <-> Entity <-> UseCase <-> ViewModel <-> UI)

8 — Testing strategy (layer-by-layer)

- **Domain**: unit test UseCases (mock Repository) — logic must be pure and deterministic
- **Data**: unit test RepositoryImpl (mock remote/local datasource and networkInfo)
- **Presentation**: unit test ViewModel (mock UseCases) and widget tests for UI (golden tests ถ้าต้องการ)
- Tools: mocktail (null-safe), flutter_test, integration_test (end-to-end)

ตัวอย่าง test (mocktail):

```
class MockProductRepository extends Mock implements ProductRepository {}

void main() {
  late MockProductRepository mockRepo;
  late GetProductsUseCase usecase;

  setUp(() {
    mockRepo = MockProductRepository();
    usecase = GetProductsUseCase(mockRepo);
  });

  test('should return list of products when repository returns data', () async {
    final p = Product(id: '1', name: 'x', price: 1.0);
    when(() => mockRepo.getProducts(page: 1))
      .thenAnswer((_) async => Right([p]));

    final result = await usecase.call(page: 1);
    expect(result.isRight(), true);
    expect(result.getOrElse(() => []), [p]);
  });
}
```

9 — Performance, maintainability, และข้อควรระวัง

- อย่า parse JSON จำนวนมากบน main isolate — ใช้ compute หรือ Isolate สำหรับ heavy parsing

- Cache: ใช้ cache + stale-while-revalidate pattern (show cached immediately, refresh in background)
- Database migrations: วาง strategy สำหรับ upgrade schema (drift/sql migrations)
- Avoid huge widget trees in single file — แยก widget ย่อยและใช้ const where possible
- การลงทะเบียน dependency: registerFactory สำหรับ object ที่มี short lifecycle (Widget/ViewModel)

---

10 — CI / code generation notes

- ใน CI: รัน flutter pub run build_runner build --delete-conflicting-outputs ก่อน test/build
- อย่า commit generated files (แต่บางทีม commit injection.config.dart ก็ได้เพื่อป้องกัน build errors) — เลือกนโยบายทีมเดียวกัน
- เพิ่ม lint (effective_dart, pedantic) และ pre-commit hook เพื่อป้องกัน import ข้ามเลเยอร์

---

11 — Checklist / Best Practices (สรุปสั้น ๆ)

- Domain must be pure (no framework)
- Inject dependencies via constructors — เท่านั้น! (for testability)
- Use Either/Result to surface failures — avoid throwing through layers
- Split dependency registration by module and use preResolve for async init
- Use feature-first folder layout; keep core small and stable
- Clear and consistent naming: XRepository = interface, XRepositoryImpl = implementation
- Tests for each layer + integration tests for flows

Clean Architecture

Clean Architecture สำหรับ Flutter โดยเน้นสามเลเยอร์หลัก **Presentation ⟶ Domain ⟶ Data Layer** แบบ technical depth เพื่อให้เข้าใจทั้งแนวคิด, การแยกความรับผิดชอบ, และตัวอย่างโค้ดจริง

---

1. ภาพรวมแนวคิด Clean Architecture

Clean Architecture ของ Robert C. Martin (Uncle Bob) เน้นการจัดเลเยอร์ให้ **Domain Logic** เป็นศูนย์กลางของระบบ และให้เลเยอร์อื่นเป็นเพียงตัวเชื่อม (Adapters) โดยใช้หลักการ:

- **Dependency Rule**: การอ้างอิงต้องวิ่งจาก outer layer ⟶ inner layer เท่านั้น (Presentation ⟶ Domain ⟶ Data)
- **Framework-independent**: Core business logic ต้องไม่พึ่งพา Flutter หรือ library ใด ๆ

- **Testability**: Domain สามารถทดสอบได้โดย mock dependency

---

2. สามเลเยอร์หลักใน Flutter

2.1 Presentation Layer

หน้าที่

- แสดง UI และรับอินพุตจากผู้ใช้
- ประสานการทำงานกับ Domain Layer ผ่าน ViewModel/Controller/Bloc
- ไม่มี business rule อยู่ใน UI โดยตรง

ส่วนประกอบ

- **Widgets** (StatelessWidget/StatefulWidget)
- **State Management** (Bloc, Riverpod, Provider, Cubit ฯลฯ)
- **View Models** หรือ Bloc/Cubit เพื่อจัดการ state

ตัวอย่าง (ใช้ ChangeNotifier เป็น ViewModel)

```
class LoginViewModel extends ChangeNotifier {
  final LoginUseCase _loginUseCase;
  bool isLoading = false;
  String? error;

  LoginViewModel(this._loginUseCase);

  Future<void> login(String email, String password) async {
    isLoading = true;
    notifyListeners();

    final result = await _loginUseCase(LoginParams(email, password));
    result.fold(
      (failure) => error = failure.message,
      (user) => print('Logged in: ${user.name}'),
    );

    isLoading = false;
    notifyListeners();
  }
}
```

ข้อสังเกต

- ไม่มีโค้ดเรียก API หรือจัดการ DB ใน Presentation
- สื่อสารกับ Domain ผ่าน Use Case เท่านั้น

2.2 Domain Layer

หน้าที่

- เป็นหัวใจของระบบ (Business Logic)
- เก็บ Entities, Value Objects, Business Rules
- ไม่พึ่งพา framework ใด ๆ
- มี **Repository Interface** ที่กำหนดวิธีสื่อสารกับ Data Layer

ส่วนประกอบ

- **Entities**: โครงสร้างข้อมูลหลักของระบบ
- **Use Cases**: ขั้นตอนการทำงานเชิงธุรกิจ
- **Repository Interfaces**: สัญญา (contract) ว่าจะเข้าถึงข้อมูลอย่างไร

ตัวอย่าง (Entity + Use Case + Repository Interface)

```
// Entity
class User {
  final String id;
  final String name;
  User({required this.id, required this.name});
}

// Repository Interface
abstract class AuthRepository {
  Future<Either<Failure, User>> login(String email, String password);
}

// Use Case
class LoginUseCase {
  final AuthRepository repository;
  LoginUseCase(this.repository);

  Future<Either<Failure, User>> call(LoginParams params) {
    return repository.login(params.email, params.password);
```

```
  }
}

class LoginParams {
  final String email;
  final String password;
  LoginParams(this.email, this.password);
}
```

ข้อสังเกต

- ไม่มีโค้ด Flutter, HTTP, DB
- สามารถ mock AuthRepository ได้ง่ายตอนทดสอบ

2.3 Data Layer

หน้าที่

- ดึง/บันทึกข้อมูลจากแหล่งจริง เช่น API, Local DB, Cache
- แปลงข้อมูลจาก DTO ⟶ Entity และกลับกัน
- Implement Repository Interfaces จาก Domain Layer

ส่วนประกอบ

- **Models/DTOs**: โครงสร้างข้อมูลตรงตาม API หรือ DB
- **Data Sources**: RemoteDataSource (API), LocalDataSource (DB/Cache)
- **Repository Implementations**

ตัวอย่าง (Implement Repository)

```
class AuthRepositoryImpl implements AuthRepository {
  final AuthRemoteDataSource remoteDataSource;

  AuthRepositoryImpl(this.remoteDataSource);

  @override
  Future<Either<Failure, User>> login(String email, String password) async {
    try {
      final dto = await remoteDataSource.login(email, password);
      return Right(User(id: dto.id, name: dto.name));
    } catch (e) {
      return Left(ServerFailure('Login failed'));
```

```
    }
  }
}

// Remote Data Source
class AuthRemoteDataSource {
  final Dio httpClient;
  AuthRemoteDataSource(this.httpClient);

  Future<UserDTO> login(String email, String password) async {
    final response = await httpClient.post('/login', data: {
      'email': email,
      'password': password,
    });
    return UserDTO.fromJson(response.data);
  }
}

// DTO
class UserDTO {
  final String id;
  final String name;
  UserDTO({required this.id, required this.name});

  factory UserDTO.fromJson(Map<String, dynamic> json) =>
      UserDTO(id: json['id'], name: json['name']);
}
```

---

3. การเชื่อมโยงทั้งสามเลเยอร์

Flow การทำงานตัวอย่าง (Login):

[UI Widget]

↓ เรียก ViewModel.login()

[Presentation Layer - ViewModel]

↓ เรียก LoginUseCase.call()

[Domain Layer - UseCase]

↓ เรียก AuthRepository.login()

[Data Layer - RepositoryImpl]

↓ เรียก AuthRemoteDataSource.login()

[API/DB] ⟶ ส่ง DTO ⟶ Map ⟶ Entity ⟶ กลับขึ้นไป

---

4. ข้อดีของการแยกเลเยอร์

- **Testable**: สามารถ unit test Domain โดย mock Data Layer
- **Maintainable**: เปลี่ยน API/DB ได้โดยไม่กระทบ Domain หรือ UI
- **Reusable**: Domain สามารถนำไปใช้ซ้ำใน platform อื่นได้ (Web, Mobile, CLI)

---

6 โปรแกรม

- **3 โปรแกรมพื้นฐาน** — เพื่อปูพื้น Clean Architecture ให้เข้าใจง่าย
- **3 โปรแกรมประยุกต์** — เพื่อเห็นการนำไปใช้ในงานจริง

โครงสร้างแต่ละโปรแกรมจะมี

1. **โครงสร้างโฟลเดอร์** (presentation, domain, data)
2. **โค้ดเต็มทุกไฟล์**
3. **คำอธิบายโค้ด**
4. **ผลการรัน** (output/หน้าจอ)

โปรแกรมพื้นฐาน

พื้นฐานที่ 1 — Counter App (Clean Architecture)

โครงสร้างโฟลเดอร์

```
lib/
├── presentation/
│   ├── counter_page.dart
│   └── counter_viewmodel.dart
├── domain/
│   ├── entities/counter.dart
│   └── usecases/increment_counter.dart
├── data/
│   ├── repositories/counter_repository_impl.dart
│   └── sources/counter_local_data_source.dart
```

└── main.dart

โค้ด

main.dart

```
import 'package:flutter/material.dart';
import 'presentation/counter_page.dart';
import 'domain/usecases/increment_counter.dart';
import 'data/sources/counter_local_data_source.dart';
import 'data/repositories/counter_repository_impl.dart';
import 'presentation/counter_viewmodel.dart';

void main() {
  final localDataSource = CounterLocalDataSource();
  final repository = CounterRepositoryImpl(localDataSource);
  final incrementCounter = IncrementCounter(repository);
  final viewModel = CounterViewModel(incrementCounter);

  runApp(MyApp(viewModel: viewModel));
}

class MyApp extends StatelessWidget {
  final CounterViewModel viewModel;
  const MyApp({required this.viewModel});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: CounterPage(viewModel: viewModel),
    );
  }
}
```

presentation/counter_page.dart

```
import 'package:flutter/material.dart';
import 'counter_viewmodel.dart';
```

```
class CounterPage extends StatefulWidget {
  final CounterViewModel viewModel;
  const CounterPage({required this.viewModel});

  @override
  State<CounterPage> createState() => _CounterPageState();
}

class _CounterPageState extends State<CounterPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Counter - Clean Architecture")),
      body: Center(
        child: Text(
          "Count: ${widget.viewModel.counter.value}",
          style: TextStyle(fontSize: 30),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          setState(() {
            widget.viewModel.increment();
          });
        },
        child: Icon(Icons.add),
      ),
    );
  }
}
```

presentation/counter_viewmodel.dart

```
import '../domain/usecases/increment_counter.dart';

class CounterViewModel {
```

```
  final IncrementCounter _incrementCounter;
  var counter = CounterValue(0);

  CounterViewModel(this._incrementCounter);

  void increment() {
    counter = _incrementCounter(counter);
  }
}

class CounterValue {
  final int value;
  CounterValue(this.value);
}
```

domain/entities/counter.dart

```
class Counter {
  final int value;
  Counter(this.value);
}
```

domain/usecases/increment_counter.dart

```
import '../entities/counter.dart';

abstract class CounterRepository {
  Counter increment(Counter current);
}

class IncrementCounter {
  final CounterRepository repository;
  IncrementCounter(this.repository);

  CounterValue call(CounterValue current) {
    final newCounter = repository.increment(Counter(current.value));
    return CounterValue(newCounter.value);
  }
```

```
}

class CounterValue {
  final int value;
  CounterValue(this.value);
}
```

data/repositories/counter_repository_impl.dart

```
import '../../domain/entities/counter.dart';
import '../../domain/usecases/increment_counter.dart';
import '../sources/counter_local_data_source.dart';

class CounterRepositoryImpl implements CounterRepository {
  final CounterLocalDataSource localDataSource;
  CounterRepositoryImpl(this.localDataSource);

  @override
  Counter increment(Counter current) {
    final newValue = localDataSource.increment(current.value);
    return Counter(newValue);
  }
}
```

data/sources/counter_local_data_source.dart

```
class CounterLocalDataSource {
  int increment(int current) {
    return current + 1;
  }
}
```

คำอธิบาย

- **Presentation** — แสดงผล UI และเรียกใช้ ViewModel
- **Domain** — กำหนด Entity Counter และ Use Case IncrementCounter
- **Data** — จำลอง LocalDataSource เพิ่มค่า counter

ผลการรัน

- แอปเริ่มต้น Count: 0
- กดปุ่ม + ⟶ Count: 1, Count: 2, ...

- **3 โปรแกรมพื้นฐาน** ⟶ เพื่อปูความเข้าใจ Clean Architecture (Presentation, Domain, Data)
- **3 โปรแกรมประยุกต์** ⟶ ใช้กับสถานการณ์จริงขึ้น เช่น การดึง API, การจัดการ State, และการเชื่อมกับ Local Storage

ผมจะทำเป็นโครงสร้าง ⟶ โค้ดเต็มทุกไฟล์ ⟶ คำอธิบาย ⟶ ผลการรัน

โปรแกรมพื้นฐานที่ 1 — Counter App

โครงสร้าง

```
lib/
├── presentation/
│   ├── counter_page.dart
│   └── counter_viewmodel.dart
├── domain/
│   ├── entities/counter.dart
│   └── usecases/increment_counter.dart
├── data/
│   ├── repositories/counter_repository_impl.dart
│   └── sources/counter_local_data_source.dart
└── main.dart
```

main.dart

```
import 'package:flutter/material.dart';
import 'presentation/counter_page.dart';
import 'domain/usecases/increment_counter.dart';
import 'data/sources/counter_local_data_source.dart';
import 'data/repositories/counter_repository_impl.dart';
import 'presentation/counter_viewmodel.dart';

void main() {
  final localDataSource = CounterLocalDataSource();
  final repository = CounterRepositoryImpl(localDataSource);
  final incrementCounter = IncrementCounter(repository);
  final viewModel = CounterViewModel(incrementCounter);
```

```
  runApp(MyApp(viewModel: viewModel));
}

class MyApp extends StatelessWidget {
  final CounterViewModel viewModel;
  const MyApp({required this.viewModel});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: CounterPage(viewModel: viewModel),
    );
  }
}
```

presentation/counter_page.dart

```
import 'package:flutter/material.dart';
import 'counter_viewmodel.dart';

class CounterPage extends StatefulWidget {
  final CounterViewModel viewModel;
  const CounterPage({required this.viewModel});

  @override
  State<CounterPage> createState() => _CounterPageState();
}

class _CounterPageState extends State<CounterPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Counter - Clean Architecture")),
      body: Center(
        child: Text(
```

```
            "Count: ${widget.viewModel.counter.value}",
            style: TextStyle(fontSize: 30),
          ),
        ),
        floatingActionButton: FloatingActionButton(
          onPressed: () {
            setState(() {
              widget.viewModel.increment();
            });
          },
          child: Icon(Icons.add),
        ),
      );
  }
}
```

presentation/counter_viewmodel.dart

```
import '../domain/usecases/increment_counter.dart';

class CounterViewModel {
  final IncrementCounter _incrementCounter;
  var counter = CounterValue(0);

  CounterViewModel(this._incrementCounter);

  void increment() {
    counter = _incrementCounter(counter);
  }
}

class CounterValue {
  final int value;
  CounterValue(this.value);
}
```

domain/entities/counter.dart

```
class Counter {
  final int value;
  Counter(this.value);
}
```

domain/usecases/increment_counter.dart

```
import '../entities/counter.dart';

abstract class CounterRepository {
  Counter increment(Counter current);
}

class IncrementCounter {
  final CounterRepository repository;
  IncrementCounter(this.repository);

  CounterValue call(CounterValue current) {
    final newCounter = repository.increment(Counter(current.value));
    return CounterValue(newCounter.value);
  }
}

class CounterValue {
  final int value;
  CounterValue(this.value);
}
```

data/repositories/counter_repository_impl.dart

```
import '../../domain/entities/counter.dart';
import '../../domain/usecases/increment_counter.dart';
import '../sources/counter_local_data_source.dart';

class CounterRepositoryImpl implements CounterRepository {
  final CounterLocalDataSource localDataSource;
  CounterRepositoryImpl(this.localDataSource);
```

```
  @override
  Counter increment(Counter current) {
    final newValue = localDataSource.increment(current.value);
    return Counter(newValue);
  }
}
```

data/sources/counter_local_data_source.dart

```
class CounterLocalDataSource {
  int increment(int current) {
    return current + 1;
  }
}
```

ผลการรัน

- เปิดแอป ⟶ Count: 0
- กด + ⟶ Count: 1, Count: 2, ...

โปรแกรมพื้นฐานที่ 2 — Simple Greeting

แนวคิด: Clean Architecture สำหรับการสร้างข้อความต้อนรับ

โครงสร้าง

```
lib/
  ├── presentation/
  │     ├── greeting_page.dart
  │     └── greeting_viewmodel.dart
  ├── domain/
  │     ├── entities/greeting.dart
  │     └── usecases/get_greeting.dart
  ├── data/
  │     ├── repositories/greeting_repository_impl.dart
  │     └── sources/greeting_local_data_source.dart
  └── main.dart
```

main.dart

```
import 'package:flutter/material.dart';
import 'presentation/greeting_page.dart';
```

```
import 'domain/usecases/get_greeting.dart';
import 'data/sources/greeting_local_data_source.dart';
import 'data/repositories/greeting_repository_impl.dart';
import 'presentation/greeting_viewmodel.dart';

void main() {
  final localDataSource = GreetingLocalDataSource();
  final repository = GreetingRepositoryImpl(localDataSource);
  final getGreeting = GetGreeting(repository);
  final viewModel = GreetingViewModel(getGreeting);

  runApp(MyApp(viewModel: viewModel));
}

class MyApp extends StatelessWidget {
  final GreetingViewModel viewModel;
  const MyApp({required this.viewModel});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: GreetingPage(viewModel: viewModel),
    );
  }
}
```

presentation/greeting_page.dart

```
import 'package:flutter/material.dart';
import 'greeting_viewmodel.dart';

class GreetingPage extends StatelessWidget {
  final GreetingViewModel viewModel;
  const GreetingPage({required this.viewModel});

  @override
```

```
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Greeting App")),
      body: Center(
        child: Text(
          viewModel.getGreetingMessage(),
          style: TextStyle(fontSize: 25),
        ),
      ),
    );
  }
}
```

presentation/greeting_viewmodel.dart

```
import '../domain/usecases/get_greeting.dart';

class GreetingViewModel {
  final GetGreeting _getGreeting;

  GreetingViewModel(this._getGreeting);

  String getGreetingMessage() {
    return _getGreeting().message;
  }
}
```

domain/entities/greeting.dart

```
class Greeting {
  final String message;
  Greeting(this.message);
}
```

domain/usecases/get_greeting.dart

```
import '../entities/greeting.dart';

abstract class GreetingRepository {
  Greeting getGreeting();
```

```
}

class GetGreeting {
  final GreetingRepository repository;
  GetGreeting(this.repository);

  Greeting call() {
    return repository.getGreeting();
  }
}
```

data/repositories/greeting_repository_impl.dart

```
import '../../domain/entities/greeting.dart';
import '../../domain/usecases/get_greeting.dart';
import '../sources/greeting_local_data_source.dart';

class GreetingRepositoryImpl implements GreetingRepository {
  final GreetingLocalDataSource localDataSource;
  GreetingRepositoryImpl(this.localDataSource);

  @override
  Greeting getGreeting() {
    return Greeting(localDataSource.getMessage());
  }
}
```

data/sources/greeting_local_data_source.dart

```
class GreetingLocalDataSource {
  String getMessage() {
    return "Hello, Flutter Developer!";
  }
}
```

ผลการรัน

- แอปเปิดขึ้น ⟶ แสดงข้อความ "Hello, Flutter Developer!"

โปรแกรมพื้นฐานที่ 3 — Random Number

แนวคิด: แยกชั้นเพื่อสุ่มตัวเลข

โครงสร้าง

```
lib/
 ├── presentation/
 │    ├── random_page.dart
 │    └── random_viewmodel.dart
 ├── domain/
 │    ├── entities/random_number.dart
 │    └── usecases/generate_random_number.dart
 ├── data/
 │    ├── repositories/random_repository_impl.dart
 │    └── sources/random_local_data_source.dart
 └── main.dart
```

main.dart

```
import 'package:flutter/material.dart';
import 'presentation/random_page.dart';
import 'domain/usecases/generate_random_number.dart';
import 'data/sources/random_local_data_source.dart';
import 'data/repositories/random_repository_impl.dart';
import 'presentation/random_viewmodel.dart';

void main() {
  final localDataSource = RandomLocalDataSource();
  final repository = RandomRepositoryImpl(localDataSource);
  final generateRandomNumber = GenerateRandomNumber(repository);
  final viewModel = RandomViewModel(generateRandomNumber);

  runApp(MyApp(viewModel: viewModel));
}

class MyApp extends StatelessWidget {
  final RandomViewModel viewModel;
  const MyApp({required this.viewModel});
```

```
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: RandomPage(viewModel: viewModel),
    );
  }
}
```

presentation/random_page.dart

```
import 'package:flutter/material.dart';
import 'random_viewmodel.dart';

class RandomPage extends StatefulWidget {
  final RandomViewModel viewModel;
  const RandomPage({required this.viewModel});

  @override
  State<RandomPage> createState() => _RandomPageState();
}

class _RandomPageState extends State<RandomPage> {
  String display = "";

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Random Number")),
      body: Center(
        child: Text(
          display.isEmpty ? "Press Button" : display,
          style: TextStyle(fontSize: 25),
        ),
      ),
      floatingActionButton: FloatingActionButton(
```

```
        onPressed: () {
          setState(() {
            display = widget.viewModel.getRandom().value.toString();
          });
        },
        child: Icon(Icons.shuffle),
      ),
    );
  }
}
```

presentation/random_viewmodel.dart

```
import '../domain/usecases/generate_random_number.dart';

class RandomViewModel {
  final GenerateRandomNumber _generateRandomNumber;
  RandomViewModel(this._generateRandomNumber);

  RandomNumberValue getRandom() {
    return _generateRandomNumber();
  }
}

class RandomNumberValue {
  final int value;
  RandomNumberValue(this.value);
}
```

domain/entities/random_number.dart

```
class RandomNumber {
  final int value;
  RandomNumber(this.value);
}
```

domain/usecases/generate_random_number.dart

```
import '../entities/random_number.dart';
```

```
abstract class RandomRepository {
  RandomNumber getRandom();
}

class GenerateRandomNumber {
  final RandomRepository repository;
  GenerateRandomNumber(this.repository);

  RandomNumberValue call() {
    final result = repository.getRandom();
    return RandomNumberValue(result.value);
  }
}

class RandomNumberValue {
  final int value;
  RandomNumberValue(this.value);
}
```

data/repositories/random_repository_impl.dart

```
import '../../domain/entities/random_number.dart';
import '../../domain/usecases/generate_random_number.dart';
import '../sources/random_local_data_source.dart';

class RandomRepositoryImpl implements RandomRepository {
  final RandomLocalDataSource localDataSource;
  RandomRepositoryImpl(this.localDataSource);

  @override
  RandomNumber getRandom() {
    return RandomNumber(localDataSource.generate());
  }
}
```

data/sources/random_local_data_source.dart

```
import 'dart:math';
```

```
class RandomLocalDataSource {
  int generate() {
    return Random().nextInt(100);
  }
}
```

ผลการรัน

- กดปุ่ม ⟶ แสดงเลขสุ่ม เช่น 42, 7, 88

---

3 โปรแกรมประยุกต์ (Advanced/Applied) ในโครงสร้าง **Clean Architecture**

ซึ่งแต่ละโปรแกรมจะซับซ้อนขึ้น มีการเชื่อมต่อ API, จัดการ State, และเพิ่มการแยกเลเยอร์ให้เหมือน production จริงมากขึ้น

โปรแกรมประยุกต์ที่ 1 — Weather App (ดึงข้อมูลจาก OpenWeather API)

โครงสร้าง

```
lib/
 ├── presentation/
 │    ├── pages/weather_page.dart
 │    ├── viewmodels/weather_viewmodel.dart
 ├── domain/
 │    ├── entities/weather.dart
 │    ├── repositories/weather_repository.dart
 │    ├── usecases/get_weather.dart
 ├── data/
 │    ├── models/weather_model.dart
 │    ├── repositories/weather_repository_impl.dart
 │    ├── sources/weather_remote_data_source.dart
 └── main.dart
```

---

main.dart

```
import 'package:flutter/material.dart';
import 'presentation/pages/weather_page.dart';
import 'presentation/viewmodels/weather_viewmodel.dart';
```

```
import 'domain/usecases/get_weather.dart';
import 'data/repositories/weather_repository_impl.dart';
import 'data/sources/weather_remote_data_source.dart';

void main() {
  final remoteDataSource = WeatherRemoteDataSource();
  final repository = WeatherRepositoryImpl(remoteDataSource);
  final getWeather = GetWeather(repository);
  final viewModel = WeatherViewModel(getWeather);

  runApp(MyApp(viewModel: viewModel));
}

class MyApp extends StatelessWidget {
  final WeatherViewModel viewModel;
  const MyApp({required this.viewModel});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: WeatherPage(viewModel: viewModel),
    );
  }
}
```

---

presentation/pages/weather_page.dart

```
import 'package:flutter/material.dart';
import '../viewmodels/weather_viewmodel.dart';

class WeatherPage extends StatefulWidget {
  final WeatherViewModel viewModel;
  const WeatherPage({required this.viewModel});

  @override
```

```
  State<WeatherPage> createState() => _WeatherPageState();
}

class _WeatherPageState extends State<WeatherPage> {
  final cityController = TextEditingController();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Weather App")),
      body: Padding(
        padding: const EdgeInsets.all(16),
        child: Column(
          children: [
            TextField(
              controller: cityController,
              decoration: InputDecoration(labelText: "Enter city name"),
            ),
            SizedBox(height: 10),
            ElevatedButton(
              onPressed: () async {
                await widget.viewModel.fetchWeather(cityController.text);
                setState(() {});
              },
              child: Text("Get Weather"),
            ),
            SizedBox(height: 20),
            if (widget.viewModel.weather != null)
              Text(
                "${widget.viewModel.weather!.city}: ${widget.viewModel.weather!.temperature}°C",
                style: TextStyle(fontSize: 24),
              ),
          ],
        ),
```

```
      ),
    );
  }
}
```

---

presentation/viewmodels/weather_viewmodel.dart

```
import '../../domain/entities/weather.dart';
import '../../domain/usecases/get_weather.dart';

class WeatherViewModel {
  final GetWeather _getWeather;
  Weather? weather;

  WeatherViewModel(this._getWeather);

  Future<void> fetchWeather(String city) async {
    weather = await _getWeather(city);
  }
}
```

---

domain/entities/weather.dart

```
class Weather {
  final String city;
  final double temperature;

  Weather({required this.city, required this.temperature});
}
```

---

domain/repositories/weather_repository.dart

```
import '../entities/weather.dart';

abstract class WeatherRepository {
  Future<Weather> getWeather(String city);
}
```

domain/usecases/get_weather.dart

```
import '../entities/weather.dart';
import '../repositories/weather_repository.dart';

class GetWeather {
  final WeatherRepository repository;
  GetWeather(this.repository);

  Future<Weather> call(String city) {
    return repository.getWeather(city);
  }
}
```

data/models/weather_model.dart

```
import '../../domain/entities/weather.dart';

class WeatherModel extends Weather {
  WeatherModel({required String city, required double temperature})
      : super(city: city, temperature: temperature);

  factory WeatherModel.fromJson(Map<String, dynamic> json) {
    return WeatherModel(
      city: json["name"],
      temperature: (json["main"]["temp"] - 273.15).toDouble(), // Kelvin ⟶ °C
    );
  }
}
```

data/sources/weather_remote_data_source.dart

```
import 'dart:convert';
import 'package:http/http.dart' as http;
import '../models/weather_model.dart';
```

```
class WeatherRemoteDataSource {
  final String apiKey = "YOUR_API_KEY";

  Future<WeatherModel> fetchWeather(String city) async {
    final response = await http.get(Uri.parse(
        "https://api.openweathermap.org/data/2.5/weather?q=$city&appid=$apiKey"));
    if (response.statusCode == 200) {
      return WeatherModel.fromJson(json.decode(response.body));
    } else {
      throw Exception("Failed to load weather");
    }
  }
}
```

---

data/repositories/weather_repository_impl.dart

```
import '../../domain/entities/weather.dart';
import '../../domain/repositories/weather_repository.dart';
import '../sources/weather_remote_data_source.dart';

class WeatherRepositoryImpl implements WeatherRepository {
  final WeatherRemoteDataSource remoteDataSource;
  WeatherRepositoryImpl(this.remoteDataSource);

  @override
  Future<Weather> getWeather(String city) {
    return remoteDataSource.fetchWeather(city);
  }
}
```

ผลการรัน

- ใส่ชื่อเมือง ⟶ กด Get Weather ⟶ แสดงอุณหภูมิ (°C) ของเมืองนั้น

การทำ Dependency Injection (DI)

การทำ Dependency Injection (DI) ใน Flutter โดยใช้สองแพ็กเกจยอดนิยม คือ **get_it** และ **injectable** พร้อมตัวอย่างจริง ๆ เพื่อให้เข้าใจลึก ๆ

การทำ Dependency Injection (DI) คืออะไร?

Dependency Injection คือเทคนิคการจัดการการสร้างและส่งผ่าน (inject) dependencies ที่แต่ละคลาสหรือออบเจ็กต์ต้องใช้ เพื่อแก้ปัญหาการผูกมัด (tight coupling) และทำให้โค้ดมีความยืดหยุ่น, ทดสอบง่าย และจัดการง่ายขึ้น

---

1. get_it — Service Locator แบบง่าย

หลักการทำงาน

- get_it คือ **Service Locator** ที่เก็บ instance ของ dependencies ไว้ในที่เดียว (singleton registry)
- เวลาต้องการใช้ dependency ที่ไหน ก็แค่เรียก getIt.get<T>() หรือ getIt<T>() เพื่อดึง instance นั้นออกมา
- เหมาะกับโค้ดขนาดเล็ก-กลาง ที่อยากได้ DI แบบไม่ซับซ้อน

วิธีใช้งาน get_it เบื้องต้น

1. เพิ่ม dependency ใน pubspec.yaml

```
dependencies:
  get_it: ^7.2.0
```

2. สร้างไฟล์ service_locator.dart

```
import 'package:get_it/get_it.dart';
import 'package:your_app/data/services/api_service.dart';
import 'package:your_app/domain/usecases/fetch_data_usecase.dart';

final getIt = GetIt.instance;

void setup() {
  // ลงทะเบียน Service แบบ singleton
  getIt.registerLazySingleton<ApiService>(() => ApiService());

  // ลงทะเบียน UseCase โดยดึง ApiService จาก getIt
  getIt.registerFactory<FetchDataUseCase>(() => FetchDataUseCase(getIt<ApiService>()));
}
```

3. เรียก setup() ตอน app เริ่มต้น (main.dart)

```
void main() {
  setup();
  runApp(MyApp());
}
```

4. ดึงใช้งานในโค้ด (เช่น ViewModel)

```
class MyViewModel {
  final FetchDataUseCase fetchDataUseCase = getIt<FetchDataUseCase>();

  void load() {
    fetchDataUseCase.execute();
  }
}
```

จุดเด่น get_it

- ใช้งานง่าย ไม่ต้องใช้ annotation
- ไม่ต้อง generate โค้ด
- เหมาะกับโปรเจกต์ขนาดเล็กถึงกลาง

---

2. injectable — DI + Code Generation + Integration กับ get_it

หลักการทำงาน

- injectable คือ **เครื่องมือช่วยสร้างโค้ดสำหรับลงทะเบียน dependencies ให้อัตโนมัติ**
- ทำงานร่วมกับ get_it
- ใช้ **annotation** (เช่น @injectable, @singleton) ประกาศ class ที่จะลงทะเบียน
- รันคำสั่ง generate โค้ด จะสร้างไฟล์ registrar ให้เรียบร้อย
- ลดความผิดพลาด และจัดการ dependencies ได้ดีในโปรเจกต์ขนาดใหญ่

วิธีใช้งาน injectable เบื้องต้น

1. เพิ่ม dependencies

```
dependencies:
  get_it: ^7.2.0
  injectable: ^2.1.0

dev_dependencies:
```

```
  injectable_generator: ^2.1.0
  build_runner: ^2.4.0
```

2. สร้างไฟล์ service_locator.dart (เรียก code generator)

```
import 'package:get_it/get_it.dart';
import 'package:injectable/injectable.dart';
import 'service_locator.config.dart'; // ไฟล์ generate อัตโนมัติ

final getIt = GetIt.instance;

@InjectableInit()
void configureDependencies() => getIt.init();
```

3. ประกาศ class ที่จะ inject ด้วย annotation

ตัวอย่าง service:

```
import 'package:injectable/injectable.dart';

@lazySingleton
class ApiService {
  void fetch() {
    print('Fetching data...');
  }
}
```

Use case ที่พึ่งพา ApiService:

```
import 'package:injectable/injectable.dart';
import 'api_service.dart';

@injectable
class FetchDataUseCase {
  final ApiService apiService;
  FetchDataUseCase(this.apiService);

  void execute() {
    apiService.fetch();
  }
}
```

4. รันคำสั่ง generate โค้ด

ใน terminal รัน:

flutter pub run build_runner build

ไฟล์ service_locator.config.dart จะถูกสร้างขึ้นอัตโนมัติพร้อมกับโค้ด register

5. เรียกใช้งานใน main.dart

```
void main() {
  configureDependencies();
  runApp(MyApp());
}
```

6. ดึง instance ใช้งาน

```
final useCase = getIt<FetchDataUseCase>();
useCase.execute();
```

ข้อดี injectable

- ลดโค้ด boilerplate ลงเยอะ
- ง่ายต่อการจัดการ dependencies แบบซับซ้อน
- รัน time code generation แทนเขียน manual
- รองรับ scopes, singleton, factories, async injection

สรุปเปรียบเทียบ get_it vs injectable

Feature	get_it	injectable + get_it
การลงทะเบียน dependencies	Manual (เขียนโค้ดเอง)	Auto-generate ด้วย annotation
ความซับซ้อนโปรเจกต์	เหมาะกับขนาดเล็ก-กลาง	เหมาะกับขนาดกลาง-ใหญ่
ความสะดวกในการใช้งาน	ง่ายมาก	ง่ายกว่าเมื่อโปรเจกต์ใหญ่
ต้องใช้ Code Generation?	ไม่ต้อง	ต้อง (build_runner)

ตัวอย่างใช้งานจริงแบบย่อ

api_service.dart

```
import 'package:injectable/injectable.dart';

@lazySingleton
class ApiService {
  String fetchData() => "Data from API";
```

```
}
```

fetch_data_usecase.dart

```
import 'package:injectable/injectable.dart';
import 'api_service.dart';

@injectable
class FetchDataUseCase {
  final ApiService apiService;
  FetchDataUseCase(this.apiService);

  String execute() => apiService.fetchData();
}
```

service_locator.dart

```
import 'package:get_it/get_it.dart';
import 'package:injectable/injectable.dart';
import 'service_locator.config.dart';

final getIt = GetIt.instance;

@InjectableInit()
void configureDependencies() => getIt.init();
```

main.dart

```
void main() {
  configureDependencies();

  final useCase = getIt<FetchDataUseCase>();
  print(useCase.execute()); // Output: Data from API
}
```

- **3 โปรแกรมพื้นฐาน**: สอนใช้ Dependency Injection ด้วย get_it และ injectable แบบง่าย ๆ
- **3 โปรแกรมประยุกต์**: ใช้ DI กับแอปจริง เช่น API, ViewModel, Repository

---

ส่วนที่ 1: โปรแกรมพื้นฐาน (get_it + injectable)

พื้นฐาน 1: ใช้งาน get_it แบบง่าย — Service Locator

โครงสร้าง

```
lib/
  ├── main.dart
  ├── service_locator.dart
  ├── api_service.dart
  └── data_repository.dart
```

โค้ด

service_locator.dart

```
import 'package:get_it/get_it.dart';
import 'api_service.dart';
import 'data_repository.dart';

final getIt = GetIt.instance;

void setup() {
  getIt.registerLazySingleton<ApiService>(() => ApiService());
  getIt.registerFactory<DataRepository>(() => DataRepository(getIt<ApiService>()));
}
```

api_service.dart

```
class ApiService {
  String fetchData() => "Data from ApiService";
}
```

data_repository.dart

```
import 'api_service.dart';

class DataRepository {
  final ApiService apiService;
  DataRepository(this.apiService);

  String getData() {
    return apiService.fetchData();
  }
}
```

main.dart

```
import 'package:flutter/material.dart';
import 'service_locator.dart';
import 'data_repository.dart';

void main() {
  setup();
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  final DataRepository repository = getIt<DataRepository>();

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("get_it Example")),
        body: Center(child: Text(repository.getData())),
      ),
    );
  }
}
```

คำอธิบาย

- ลงทะเบียน ApiService เป็น singleton
- DataRepository ดึง ApiService ผ่าน constructor
- main เรียก getIt ดึง DataRepository มาใช้แสดงข้อความ

ผลการรัน

หน้าจอแสดงข้อความ:

Data from ApiService

พื้นฐาน 2: ใช้ injectable + get_it พร้อม code generation

โครงสร้าง

lib/

```
├── main.dart
├── service_locator.dart
├── api_service.dart
├── data_repository.dart
└── service_locator.config.dart (generate เอง)
```

โค้ด

pubspec.yaml

```
dependencies:
  flutter:
    sdk: flutter
  get_it: ^7.2.0
  injectable: ^2.1.0

dev_dependencies:
  build_runner: ^2.4.0
  injectable_generator: ^2.1.0
```

service_locator.dart

```
import 'package:get_it/get_it.dart';
import 'package:injectable/injectable.dart';
import 'service_locator.config.dart';

final getIt = GetIt.instance;

@InjectableInit()
void configureDependencies() => getIt.init();
```

api_service.dart

```
import 'package:injectable/injectable.dart';

@lazySingleton
class ApiService {
  String fetchData() => "Data from ApiService (injectable)";
}
```

data_repository.dart

```
import 'package:injectable/injectable.dart';
```

```
import 'api_service.dart';

@injectable
class DataRepository {
  final ApiService apiService;
  DataRepository(this.apiService);

  String getData() {
    return apiService.fetchData();
  }
}
```

main.dart

```
import 'package:flutter/material.dart';
import 'service_locator.dart';
import 'data_repository.dart';

void main() {
  configureDependencies();
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  final DataRepository repository = getIt<DataRepository>();

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("injectable Example")),
        body: Center(child: Text(repository.getData())),
      ),
    );
  }
}
```

รันคำสั่ง generate

flutter pub run build_runner build

ผลการรัน

หน้าจอแสดงข้อความ:

Data from ApiService (injectable)

พื้นฐาน 3: ใช้ get_it กับ ViewModel

โครงสร้าง

lib/

```
├── main.dart
├── service_locator.dart
├── api_service.dart
├── data_repository.dart
└── viewmodel.dart
```

โค้ด

service_locator.dart

```
import 'package:get_it/get_it.dart';
import 'api_service.dart';
import 'data_repository.dart';
import 'viewmodel.dart';

final getIt = GetIt.instance;

void setup() {
  getIt.registerLazySingleton<ApiService>(() => ApiService());
  getIt.registerFactory<DataRepository>(() => DataRepository(getIt<ApiService>()));
  getIt.registerFactory<MyViewModel>(() => MyViewModel(getIt<DataRepository>()));
}
```

viewmodel.dart

```
import 'data_repository.dart';

class MyViewModel {
  final DataRepository repository;
  MyViewModel(this.repository);
```

```
  String getData() => repository.getData();
}
```

main.dart

```
import 'package:flutter/material.dart';
import 'service_locator.dart';
import 'viewmodel.dart';

void main() {
  setup();
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  final MyViewModel viewModel = getIt<MyViewModel>();

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("get_it with ViewModel")),
        body: Center(child: Text(viewModel.getData())),
      ),
    );
  }
}
```

ผลการรัน

หน้าจอแสดงข้อความ:

Data from ApiService

ส่วนที่ 2: โปรแกรมแนวประยุกต์ (ใช้ get_it + injectable ในแอปจริง)

ประยุกต์ 1: Fetch Data API + DI ด้วย get_it

โครงสร้าง

```
lib/
 ├── main.dart
 ├── service_locator.dart
 ├── api_service.dart
 ├── data_repository.dart
 ├── domain/usecases/fetch_user_usecase.dart
 ├── presentation/viewmodel/user_viewmodel.dart
 └── presentation/pages/user_page.dart
```

โค้ดตัวอย่างย่อ

api_service.dart

```
import 'dart:convert';
import 'package:http/http.dart' as http;

class ApiService {
  Future<String> fetchUser() async {
    final response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/users/1'));
    if (response.statusCode == 200) {
      final data = json.decode(response.body);
      return data['name'];
    } else {
      throw Exception('Failed to load user');
    }
  }
}
```

data_repository.dart

```
import '../api_service.dart';

class DataRepository {
  final ApiService apiService;
  DataRepository(this.apiService);

  Future<String> getUserName() {
    return apiService.fetchUser();
  }
```

```
}
```

domain/usecases/fetch_user_usecase.dart

```
import '../../data_repository.dart';

class FetchUserUseCase {
  final DataRepository repository;
  FetchUserUseCase(this.repository);

  Future<String> execute() {
    return repository.getUserName();
  }
}
```

presentation/viewmodel/user_viewmodel.dart

```
import '../../domain/usecases/fetch_user_usecase.dart';

class UserViewModel {
  final FetchUserUseCase fetchUserUseCase;
  UserViewModel(this.fetchUserUseCase);

  Future<String> loadUserName() {
    return fetchUserUseCase.execute();
  }
}
```

presentation/pages/user_page.dart

```
import 'package:flutter/material.dart';
import '../viewmodel/user_viewmodel.dart';

class UserPage extends StatefulWidget {
  final UserViewModel viewModel;
  const UserPage({required this.viewModel});

  @override
  _UserPageState createState() => _UserPageState();
}
```

```
class _UserPageState extends State<UserPage> {
  String? userName;
  bool loading = false;

  @override
  void initState() {
    super.initState();
    load();
  }

  void load() async {
    setState(() => loading = true);
    userName = await widget.viewModel.loadUserName();
    setState(() => loading = false);
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('User')),
      body: Center(
        child: loading ? CircularProgressIndicator() : Text(userName ?? 'No data'),
      ),
    );
  }
}
```

service_locator.dart

```
import 'package:get_it/get_it.dart';
import 'api_service.dart';
import 'data_repository.dart';
import 'domain/usecases/fetch_user_usecase.dart';
import 'presentation/viewmodel/user_viewmodel.dart';
```