40 · Clean Architecture

Flutter Practice Notes

Contents

40 · Clean Architecture

Introduction

This module covers Clean Architecture in Flutter: the **layered design** (presentation / domain / data) governed by the **dependency rule** (dependencies point **inward**, toward the domain), the **domain layer** (entities, use cases, repository *interfaces*), the **data layer** (repository *implementations*, data sources, DTOs, mapping), the **presentation layer** (state management + view state, mapping domain → UI), and how they **wire together** via DI — with a capstone building a full vertical slice. It's the synthesis of SOLID (Module 04), design patterns (Module 05), DI (Module 14), state (Module 11), and error handling (Module 38).

Why this module exists

As apps grow, "put everything in the widget" collapses under change — business rules tangle with UI and I/O, nothing is testable, and swapping a data source ripples everywhere. Clean Architecture imposes **boundaries** so the **business logic (domain) is independent** of Flutter, the network, and the database. The payoff is testability (domain tests need no device/network), flexibility (swap data sources/UI/state libs), and longevity — the difference between an app you can evolve for years and one that ossifies. It's the backbone that later architecture modules (MVVM, feature-first, DDD) build on.

Module map

#	File	Topic	Level
1	01_clean_architecture_overview.md	Layers, the dependency rule, why boundaries, tradeoffs	🔴
2	02_domain_layer.md	Entities, use cases, repository interfaces (the pure core)	🔴
3	03_data_layer.md	Repository impls, data sources, DTOs, mapping, `Result`	🔴
4	04_presentation_layer.md	State/view models, calling use cases, domain→UI mapping	🟡
5	05_clean_architecture_integration.md	Capstone: a full vertical slice wired via DI	🔴

> **Cross-references:** SOLID (DIP underpins the dependency rule): Module 04. Patterns (Repository/Adapter): Module 05. DI (wiring): Module 14. State management: Module 11. Error handling (`Result` /failures): Module 38. Testing: Module 49. Compared with MVC/MVP/MVVM: 41/42/43; feature-first/DDD: 44/46.

Prerequisites

04 SOLID (esp. DIP), 05 Design Patterns (Repository), 14 Dependency Injection, 11 State Management, 38 Error Handling.

What you'll be able to do after this module

- Explain the layers and the dependency rule, and why the domain must be independent.
- Model a pure domain (entities, use cases, repository interfaces) with no Flutter/IO.
- Implement the data layer (repositories, data sources, DTOs, mapping) returning `Result`.
- Connect UI via state/view models that call use cases and map domain → UI state.
- Wire a full vertical slice via DI and test each layer in isolation.

Capstone

Vertical slice: A feature (e.g., "user profile") built cleanly end-to-end — domain (entity + `GetProfile` use case + `ProfileRepository` interface), data (repository impl over remote/local data sources + DTO mapping + `Result`), presentation (bloc/notifier calling the use case, mapping to view state) — wired with DI, each layer unit-tested with fakes, and a documented tradeoff analysis (when this structure is worth it).

Summary

Clean Architecture = layered boundaries with the dependency rule pointing inward, keeping the **domain independent** of Flutter/network/DB. Domain (entities/use cases/interfaces) is pure; data implements interfaces and maps DTOs; presentation calls use cases and maps to UI; DI wires it. The result is testable, flexible, long-lived software — applied proportionally to the app's complexity.

Clean Architecture Overview (Layers & the Dependency Rule)

> Clean Architecture organizes code into concentric layers — **domain** (innermost: entities + business rules), **application/use cases**, and outer **data** and **presentation** — governed by one rule: **the Dependency Rule — source-code dependencies point only inward.** The domain knows nothing of Flutter, HTTP, or SQLite; outer layers depend on inner **abstractions** (interfaces), never the reverse. This keeps business logic **independent, testable, and swappable** — you can change the UI framework, network client, or database without touching the core. Apply it **proportionally**: full ceremony for complex, long-lived apps; lighter for simple ones.

Introduction

This file establishes the mental model: the layers, the dependency rule (and its DIP underpinning), what "independent domain" buys you, and the honest tradeoffs (boilerplate vs longevity). Everything else in the module is an implementation of these principles.

Why this concept exists

Without boundaries, business rules get entangled with UI widgets and I/O, so a change to the API or a swap of state library ripples through the whole app, and nothing can be tested without a device/network. Clean Architecture (Uncle Bob's synthesis of hexagonal/onion/ports-and-adapters) makes the **policy** (business rules) independent of the **details** (frameworks, DB, UI), so details can change without disturbing policy.

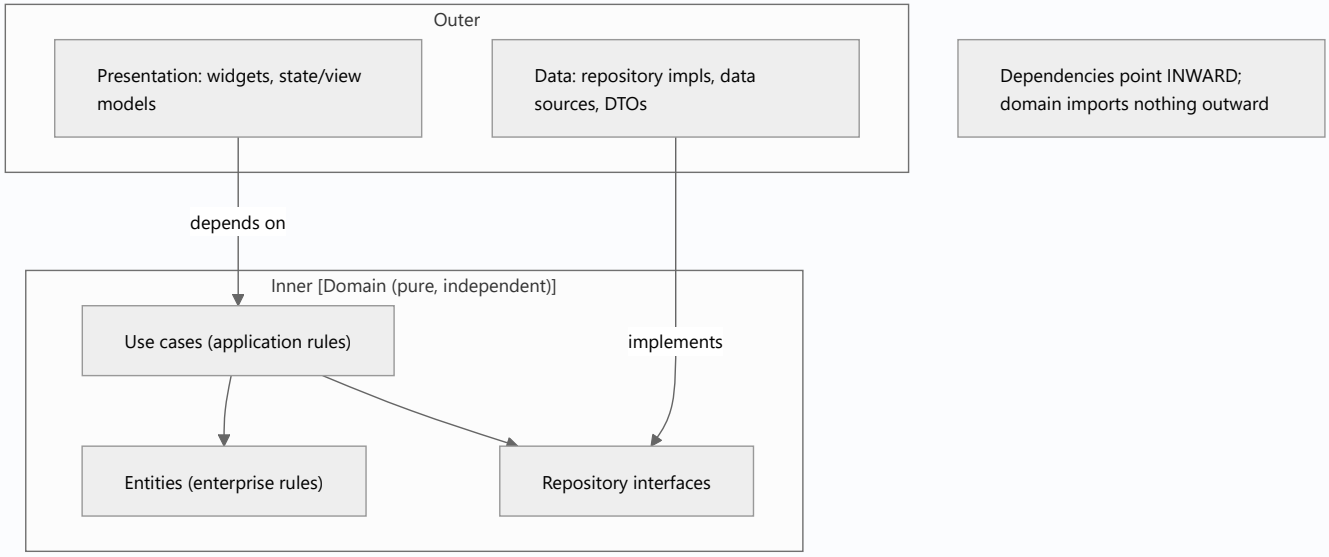
Real-world analogy

Think of a **power tool with swappable attachments**: the **motor (domain)** is the valuable core; the **attachments (UI, DB, network)** plug into a **standard socket (interfaces)**. The motor doesn't know or care which attachment is fitted — you can swap a drill bit for a sander (change the database, or Flutter for another UI) without rebuilding the motor. The dependency rule is that **attachments fit the motor's socket**, never the motor reshaped to fit an attachment.

Problem Statement

For a feature, decide what belongs in each layer, ensure the domain has **zero** framework/IO imports, make outer layers depend on domain **interfaces** (not concretes), and justify when this structure is worth its cost. You'll map responsibilities and enforce the dependency direction.

Internal Working



- **Layers (inner → outer)**:
  - **Entities** (enterprise business rules): core domain objects + invariants, the most stable, framework-free.
  - **Use cases / interactors** (application business rules): orchestrate entities to fulfill one app-specific action (`GetProfile` , `PlaceOrder`); depend on **repository interfaces**.
  - **Interface adapters** (data + presentation): **repositories** (impl the domain interfaces, map DTOs↔entities) and **presenters/state** (map domain↔UI).
  - **Frameworks & drivers** (outermost): Flutter, `dio` , `sqflite` , Firebase — the volatile details.
- **The Dependency Rule (the whole point)**: **source dependencies point inward only.** Domain has **no imports** of Flutter/HTTP/DB. Outer layers depend on **inner abstractions** (repository *interfaces* defined in the domain), realized by **Dependency Inversion** (Module 04) — the domain declares *what* it needs; the data layer provides *how*.
- **Independence buys**: **testability** (domain/use cases unit-tested with fakes, no device/network), **flexibility** (swap DB/network/UI/state lib without touching domain), **clarity** (business rules in one framework-free place), **longevity** (details churn; policy stays).
- **Crossing boundaries with data**: pass **simple data structures** (entities/DTOs), not framework objects, across boundaries; **map** at the edges (DTO↔entity, entity↔view state) so no layer leaks another's types.
- **Where things go**: business rule → domain; JSON/SQL/HTTP → data; widgets/state → presentation. If the domain imports `package:flutter` or `dio` , the rule is broken.
- **Proportionality (be honest)**: full Clean Architecture adds layers/boilerplate/mapping — worth it for **complex, long-lived, team** apps; **overkill** for a small prototype. Scale the ceremony to the app (05_clean_architecture_integration.md).

Memory Representation

Not a runtime structure — a **source-dependency graph**. The invariant is directional: inner layers have no edges to outer ones. Data crossing boundaries is plain objects (entities/DTOs), not framework handles.

Compiler Behavior

The rule is enforceable at **compile time**: the domain package/folder simply doesn't import Flutter/data packages (tools/linters can assert import boundaries). If it compiles without those imports, the core is independent.

Runtime Behavior

At runtime, DI wires concrete data/presentation implementations to domain interfaces; calls flow outward→inward (UI → use case → repository interface → impl) while **dependencies** (compile-time) point inward.

Flutter Engine Behavior

Flutter lives only in the **presentation/outermost** layer; the domain runs on plain Dart (testable in a pure Dart VM, no widget binding).

Dart VM Behavior

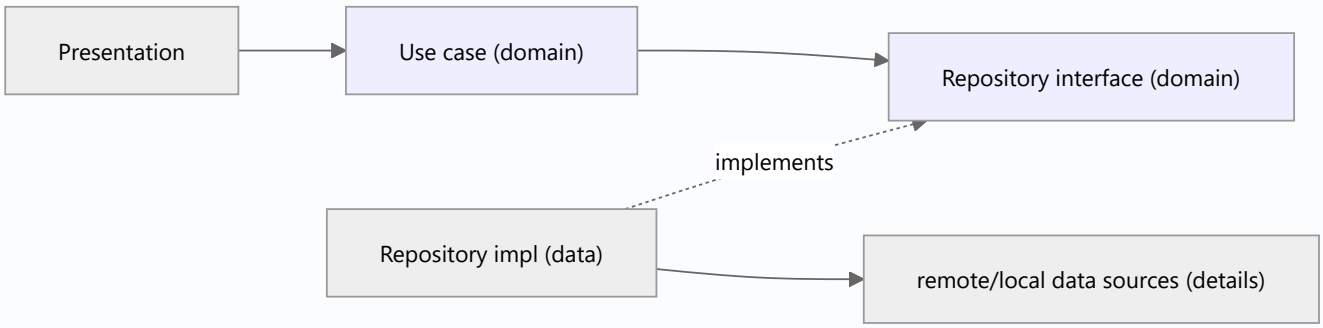
Domain/use-case tests run as fast pure-Dart unit tests (no Flutter binding), because the domain has no framework dependency — a direct benefit of the rule.

Examples

```
// DOMAIN (pure Dart — NO Flutter/dio/sqflite imports)
class Profile { final String id, name; const Profile(this.id, this.name); } // entity

abstract class ProfileRepository {                // interface lives in the domain
  Future<Result<Profile>> getProfile(String id);
}

class GetProfile {                                // use case: one app action
  final ProfileRepository repo;                   // depends on the ABSTRACTION
  GetProfile(this.repo);
  Future<Result<Profile>> call(String id) => repo.getProfile(id);
}
```

```
// DATA (outer) — implements the domain interface; may import dio/sqflite
class ProfileRepositoryImpl implements ProfileRepository {  // dependency points inward
  final ProfileRemote remote;                               // detail
  ProfileRepositoryImpl(this.remote);
  @override
  Future<Result<Profile>> getProfile(String id) async {
    final dto = await remote.fetch(id);                     // DTO (data-layer type)
    return Success(dto.toEntity());                         // map DTO -> entity at the edge
  }
}
// PRESENTATION (outer) depends on GetProfile (domain), never on ProfileRepositoryImpl directly.
```

Diagrams



Common Mistakes

Mistake	Why it breaks Clean Arch	Fix
Domain imports Flutter/ `dio` / `sqflite`	Couples policy to details	Keep domain pure; abstractions only
Use case depends on a concrete repo	Violates dependency rule/DIP	Depend on the interface
Passing DTOs/framework objects to UI	Leaks data-layer types inward/upward	Map DTO↔entity↔view state at edges
Business rules in widgets/repos	Untestable, tangled	Rules in domain (entities/use cases)
Interface defined in the data layer	Dependency points outward	Define interfaces in the domain
Full ceremony on a tiny app	Needless boilerplate	Scale to complexity
Anemic use cases that just pass through	Ceremony w/o value	Add real orchestration or simplify

Best Practices

- Keep the **domain pure** (no Flutter/IO imports) — entities, use cases, and repository **interfaces** only; enforce the **dependency rule** (inward-only) via **DIP**.
- **Define interfaces in the domain**, implement them in **data**; cross boundaries with **plain data** (entities/DTOs) and **map at the edges** so no layer leaks types.
- Put **business rules in the domain**, **I/O in data**, **UI/state in presentation**; keep use cases meaningful (real orchestration), not empty pass-throughs.
- **Scale ceremony to complexity** — full structure for complex/long-lived/team apps, lighter for simple ones; verify independence by unit-testing the domain with no device/network.

Performance

No runtime cost — it's a compile-time/source-organization concern. Indirection (interfaces/mapping) is negligible. The "cost" is developer time (boilerplate/mapping); the payoff is testability + change-resilience, which is why proportionality matters.

Advantages / Disadvantages

- **+** Independent, testable domain; swap details (DB/network/UI/state) freely; clear separation; long-lived; team-scalable.
- **−** Boilerplate + mapping + more files; overkill for simple apps; learning curve; risk of anemic/ceremonial layers if misapplied.

Interview Questions

1. 🟢 **What is the Dependency Rule?** — Source-code dependencies point only inward; inner layers (domain) know nothing of outer layers (Flutter/DB/network).

2. 🟢 **What are the layers?** — Entities → use cases (domain) → interface adapters (data + presentation) → frameworks/drivers (outermost).
3. 🟡 **How is the dependency rule enforced technically?** — Via Dependency Inversion: the domain defines repository interfaces; the data layer implements them, so the domain imports nothing outward.
4. 🟡 **Why must the domain be framework-free?** — So business rules are testable without a device/network and independent of UI/DB/network churn.
5. 🟡 **How does data cross boundaries?** — As plain data (entities/DTOs), mapped at the edges (DTO↔entity↔view state) — no framework objects leak across layers.
6. 🔴 **When is Clean Architecture overkill?** — For small/short-lived apps where the boilerplate/mapping cost outweighs the testability/flexibility benefit — scale ceremony to complexity.
7. 🔴 **How do runtime call flow and compile-time dependencies differ?** — Calls flow outward→inward (UI→use case→repo impl), while source dependencies point inward (impl depends on the domain interface, wired by DI).

Senior Engineer Tips

- The one rule that matters: the domain folder imports neither Flutter nor any data package — if it does, you don't have Clean Architecture, you have layers-in-name-only.
- Define repository interfaces in the domain and let DI supply implementations; this single inversion is what makes the whole thing testable and swappable.
- Right-size it: reach for the full structure on complex, long-lived, multi-dev apps, and a lighter version elsewhere — ceremony without payoff is just cost.

Architect Perspective

Clean Architecture is DIP applied at the module scale: policy (domain) is stable and independent; details (frameworks/DB/UI) are plugins behind interfaces. This is the backbone the rest of the architecture band elaborates — MVVM/feature-first/DDD are variations on the same boundary discipline — and it's what makes large Flutter apps testable, evolvable, and team-scalable. Applied proportionally, it's the difference between software that lasts and software that ossifies (Module 04, 02_domain_layer.md, Module 46).

Summary

- Layers: entities → use cases (domain) → data + presentation (adapters) → frameworks (outer); the **Dependency Rule** points inward only.
- Keep the domain pure (no Flutter/IO), define interfaces in the domain (DIP), map plain data at boundaries — buying testability, flexibility, longevity.

- No runtime cost; boilerplate cost → apply proportionally to app complexity.

Revision Notes

- Layers (in→out): Entities, Use cases (domain), Interface adapters (data/presentation), Frameworks/drivers. Dependency Rule = inward only.
- Domain = pure Dart (no Flutter/dio/sqflite); interfaces defined in domain, implemented in data (DIP); cross boundaries with plain data + map at edges.
- Benefits: testable/independent domain, swappable details, longevity; cost: boilerplate/mapping → scale to complexity; enforced at compile time (imports).

Practice Questions

1. State the dependency rule and how DIP enforces it.
2. Why can the domain be tested without a device or network?
3. When would you *not* use full Clean Architecture?

Coding Questions

1. Sketch a pure domain (entity + use case + repository interface) with no framework imports.
2. Implement the interface in a data class that may import `dio`.
3. Identify and fix a dependency-rule violation (domain importing Flutter).

Mini Project

Layer map + dependency check (Flutter): For a feature, produce a layer map (what goes in domain/data/presentation), implement a pure domain (entity + use case + repository interface) and a data impl of the interface, and verify the domain compiles with **no** Flutter/data imports (unit-tested in pure Dart). Write a short tradeoff note on when this structure is worth it. Acceptance: correct per-layer responsibilities; dependency rule holds (domain pure, interfaces in domain, impl inward); data crosses boundaries as plain types; domain unit-tested without device/network; proportionality justified.

The Domain Layer (Entities, Use Cases, Interfaces)

> The domain is the **pure heart** of the app — **entities** (business objects + invariants), **use cases/interactors** (one class = one application action, orchestrating entities), and **repository interfaces** (what the app needs from the outside, declared abstractly) — written in **plain Dart with zero framework/IO imports**. It depends on **nothing**; everything depends on it. This is what makes business logic testable in milliseconds, reusable across UIs, and immune to changes in Flutter, the network, or the database.

Introduction

This file details the innermost layer: what entities/use cases/interfaces are, how to design them, and why keeping them pure is the whole game. It's the concrete realization of "policy independent of details" from the overview (01_clean_architecture_overview.md).

Why this concept exists

Business rules are the app's most valuable and most stable asset — and the thing most damaged by being tangled with UI/IO. Isolating them in a framework-free layer means they can be tested exhaustively without a device, reused across platforms/UIs, and left untouched when details churn. Use cases give each action a single, named, testable home instead of scattering logic across widgets and repositories.

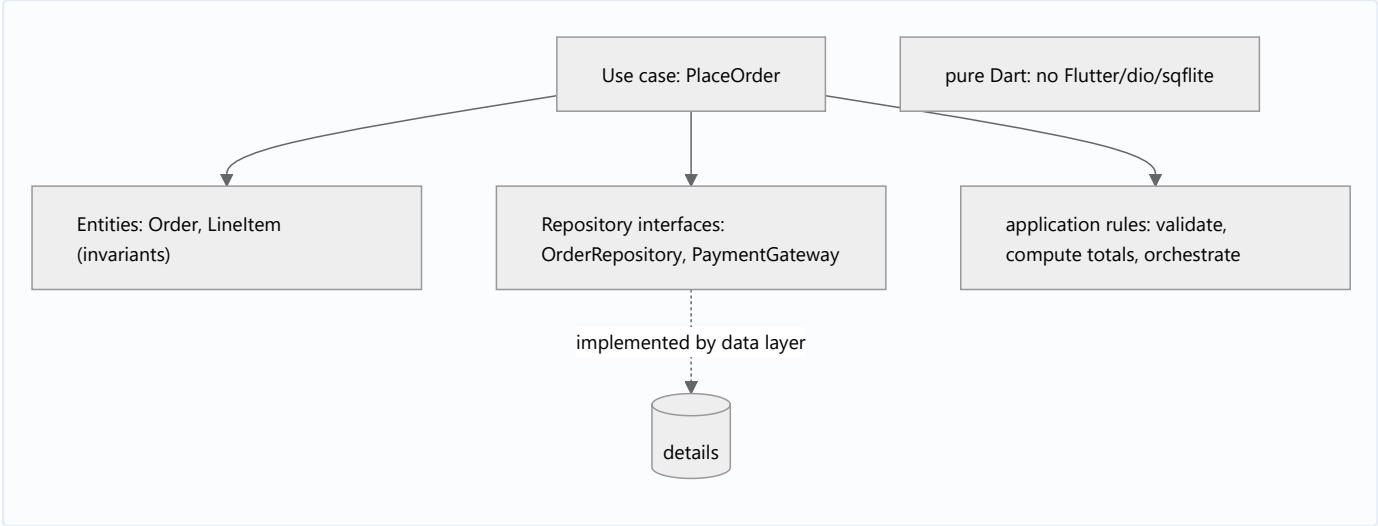
Real-world analogy

The domain is a **recipe book**: **entities** are the ingredients with their properties (an egg, its freshness rules), **use cases** are the recipes (one recipe = "make an omelette," a specific sequence), and **repository interfaces** are the **shopping list** ("I need eggs and butter") — the recipe says *what* it needs, not *which shop* provides it. The recipe book works in any kitchen (UI) with any supplier (data source).

Problem Statement

Model a feature's core: define entities with invariants, a use case per action that orchestrates them via repository interfaces, and abstract interfaces declaring exactly what the app needs from outside — all in pure Dart, unit-testable with fakes. You'll design the pure domain for, say, placing an order.

Internal Working



- **Entities** (enterprise business rules): the core objects + **invariants** (an `Order` total must equal the sum of line items; an email must be valid). Prefer **immutable** value objects; put **business validation/behavior** on them (not just data bags). They're the most stable, framework-free code.
- **Use cases / interactors** (application rules): **one class per application action** (`GetProfile` , `PlaceOrder` , `SearchProducts`), typically with a single `call(...)` (callable class). They **orchestrate** entities + repository interfaces to fulfill the action, applying app-specific rules (validation, sequencing, combining sources) and returning a `Result` / `Either` (Module 38). They depend only on **abstractions**.
- **Repository interfaces**: abstract contracts declaring **what the domain needs** (`Future<Result<Order>> placeOrder(Order)`), **defined in the domain**, implemented in the data layer (03_data_layer.md). This is the **inversion** that keeps dependencies inward (Module 04).
- **Purity (non-negotiable)**: **no** `package:flutter` , `dio` , `sqflite` , `path_provider` , etc. If the domain needs a value (current time, id) it takes it as a parameter or via an injected abstraction (a `Clock` / `IdGenerator` interface) — never a framework call.
- **Failures as values**: use cases return typed failures (`Result` /sealed `Failure`) rather than throwing for expected outcomes (Module 38); bugs still throw.
- **No UI/persistence concerns**: no widgets, no JSON, no SQL — those live outward. The domain speaks only entities + interfaces.
- **Testability**: use cases are unit-tested by injecting **fake repositories** and asserting orchestration/rules — fast, deterministic, no device (Module 49).

Memory Representation

Entities are plain (ideally immutable) Dart objects. Use cases hold references to injected interfaces. No framework handles, no IO state — just business data + behavior.

Compiler Behavior

The domain compiles with **no framework imports** — a hard, checkable boundary. If it references Flutter/dio, purity is broken. Sealed classes/ `Result` give exhaustive failure handling.

Runtime Behavior

Use cases execute orchestration synchronously/asynchronously against injected interfaces; at runtime DI supplies real implementations, but the domain code is oblivious to which.

Flutter Engine Behavior

None — the domain runs on the plain Dart VM (unit tests need no `WidgetsFlutterBinding`).

Dart VM Behavior

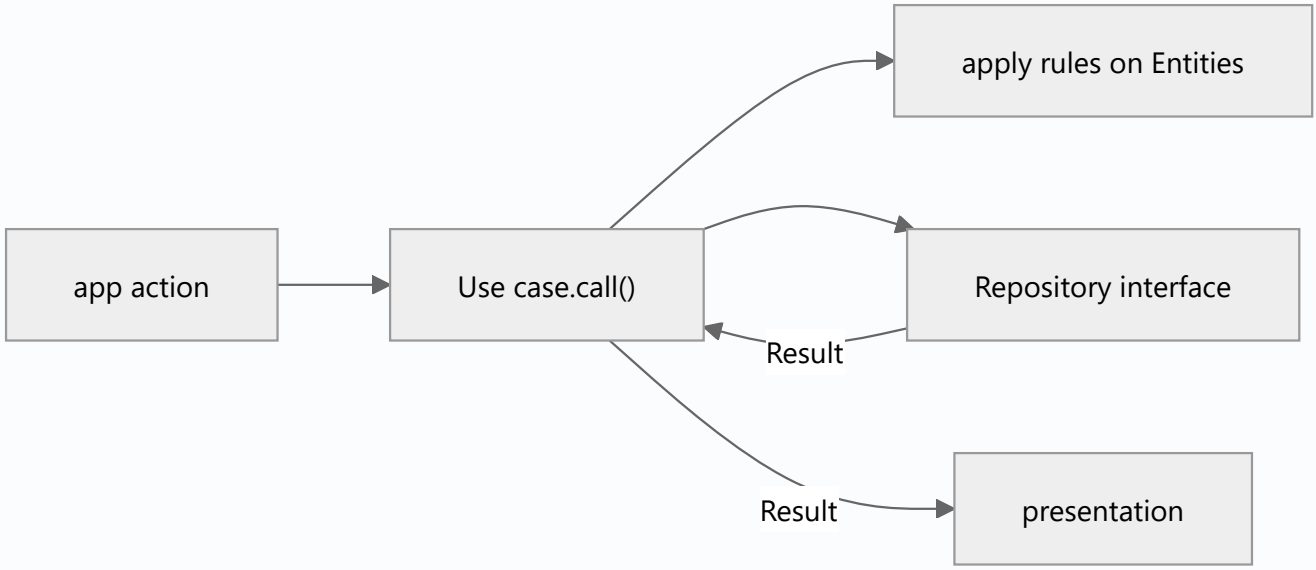
Pure-Dart, so domain tests are the fastest tier (no Flutter binding, no IO). Immutable entities are cheap and safe to share.

Examples

```
// ENTITIES — business objects with invariants (pure Dart, immutable)
class LineItem {
  final String productId; final int qty; final int unitPriceCents;
  const LineItem(this.productId, this.qty, this.unitPriceCents);
  int get subtotalCents => qty * unitPriceCents;
}

class Order {
  final String id; final List<LineItem> items;
  Order(this.id, this.items) {
    if (items.isEmpty) throw ArgumentError('order needs items'); // invariant (bug if violated)
  }
  int get totalCents => items.fold(0, (s, i) => s + i.subtotalCents);
}

// REPOSITORY INTERFACES — what the domain needs (defined here, implemented in data)
abstract class OrderRepository { Future<Result<Order>> place(Order order); }
abstract class Clock { DateTime now(); }   // abstract even for "framework" values

// USE CASE — one application action; orchestrates + applies rules; returns Result
class PlaceOrder {
  final OrderRepository orders;
  PlaceOrder(this.orders);
  Future<Result<Order>> call(Order order) async {
    if (order.totalCents <= 0) return Failure(ValidationFailure('empty order')); // app rule
    return orders.place(order);          // orchestrate via the abstraction
  }
}
```

```
// Testable with a FAKE repo — no device/network
test('PlaceOrder rejects empty totals', () async {
  final uc = PlaceOrder(FakeOrderRepo());
  final r = await uc(Order('1', [LineItem('p', 0, 0)]));
  expect(r, isA<Failure>());
});
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Framework imports in domain	Breaks purity/testability	Plain Dart + abstractions (Clock/Id)
Anemic entities (data bags only)	Rules leak elsewhere	Put invariants/behavior on entities
Business logic in repositories/widgets	Untestable, scattered	Put it in use cases/entities
Use cases returning DTOs/JSON	Leaks data concerns inward	Return entities/ `Result`
One giant "service" instead of use cases	Poor cohesion/naming	One use case per action
Throwing for expected failures	Callers may miss them	Return `Result` /typed failures
Depending on concrete repos	Violates dependency rule	Depend on interfaces

Best Practices

- Keep the domain **pure** (no framework/IO); model **entities with invariants/behavior** (not anemic bags); prefer **immutability**.
- **One use case per application action** (callable `call`), **orchestrating** entities + repository **interfaces**, returning `Result` **/typed failures**.
- **Define repository interfaces in the domain**; abstract even "framework" needs (time/ids) behind interfaces so the core stays pure.
- Put **business rules in the domain** (entities/use cases), never in widgets/repos; **unit-test** use cases with fakes (fast, no device).

Performance

Pure-Dart domain tests are the fastest tier (no binding/IO). Immutable entities are cheap; use cases add negligible orchestration cost. The design cost is more classes/files — offset by test speed and change-resilience.

Advantages / Disadvantages

- **+** Fast/exhaustive unit tests (no device), reusable across UIs/platforms, business rules centralized + stable, framework-independent.
- **−** More classes (use cases/interfaces), mapping to/from data types, risk of anemic/ceremonial use cases if overdone.

Interview Questions

1. 🟢 **What lives in the domain layer?** — Entities (business objects + invariants), use cases (application actions), and repository interfaces — all pure Dart.
2. 🟢 **What is a use case?** — A single-responsibility class for one application action (`call(...)`) that orchestrates entities/repositories and returns a `Result`.
3. 🟡 **Why must the domain be framework-free?** — So business rules are testable without a device/network and independent of UI/DB/network changes.
4. 🟡 **Where are repository interfaces defined and why?** — In the domain (implemented in data), so dependencies point inward (Dependency Inversion).
5. 🟡 **How do you handle a "framework" need like current time in a pure domain?** — Abstract it (a `Clock` interface) and inject an implementation, keeping the domain pure and testable.
6. 🔴 **Entities vs anemic models — why does it matter?** — Entities carry invariants/behavior so rules live in one testable place; anemic bags push logic into repos/widgets where it tangles and can't be tested.
7. 🔴 **Why return `Result` from use cases instead of throwing?** — Expected failures become explicit, compiler-checkable outcomes callers must handle; bugs still throw.

Senior Engineer Tips

- Keep a hard "no Flutter/dio/sqflite import" rule on the domain folder and test it in pure Dart; that single constraint delivers most of Clean Architecture's value.
- Make use cases thin-but-meaningful — real orchestration/validation, not empty pass-throughs; if a use case only forwards to a repo, ask whether it earns its place.
- Put invariants on entities (constructors/methods) so illegal states are unrepresentable, and abstract even time/ids behind interfaces so tests are deterministic.

Architect Perspective

The domain layer is where the app's essential complexity lives, protected from accidental complexity (frameworks/IO). Entities encode enterprise rules, use cases encode application rules, and interfaces declare needs — all pure, testable, and stable. This is the asset the whole architecture exists to protect and the seam DDD later deepens (aggregates/value objects) (Module 46); the data layer merely satisfies its interfaces (03_data_layer.md).

Summary

- Domain = pure Dart: entities (invariants/behavior), use cases (one action each, orchestrate + return `Result`), repository interfaces (defined here).
- No framework/IO imports; abstract even time/ids; business rules live here; return typed failures.
- Unit-testable with fakes (fastest tier); depends on nothing, everything depends on it.

Revision Notes

- Entities: business objects + invariants (immutable, behavior on them); use cases: one action, `call()`, orchestrate entities + interfaces, return `Result`.
- Repository interfaces defined in domain (impl in data — DIP); no Flutter/dio/sqflite; abstract Clock/IdGenerator for purity.
- Business rules in domain; expected failures as `Result`/typed; unit-test with fakes (pure Dart, no device).

Practice Questions

1. What distinguishes an entity from an anemic model?
2. Why is a use case better than putting logic in the repository?
3. How do you keep the domain pure when it needs the current time?

Coding Questions

1. Model an entity with an invariant and a use case that enforces an app rule.
2. Define a repository interface in the domain and a `Clock` abstraction.
3. Unit-test the use case with a fake repository (no device).

Mini Project

Pure domain slice (Flutter): Model a feature's domain — entities with invariants/behavior, repository interfaces (+ a `Clock` / `IdGenerator` abstraction), and use cases (one per action) that orchestrate and return `Result`/typed failures — in pure Dart. Unit-test the use cases with fakes. Acceptance: zero framework/IO imports in domain; entities carry invariants; use cases orchestrate + return typed results; interfaces defined in domain; use cases unit-tested with fakes (no device); business rules live in the domain only.

The Data Layer (Repositories, Data Sources, DTOs, Mapping)

> The data layer **implements the domain's repository interfaces** using concrete details: **data sources** (remote/`dio`, local/`sqflite`/secure storage), **DTOs** (data-transfer objects that mirror the wire/DB shape and handle JSON/columns), and **mappers** (DTO ↔ domain entity) — coordinating sources (cache-first, offline fallback), converting exceptions into typed `Result`/`Failure`**s** at the boundary, and returning **entities** (never DTOs) upward. It's the only layer allowed to know about HTTP, SQL, and JSON; it satisfies the domain's needs without the domain ever knowing *how*.

Introduction

This file covers the outward-facing layer that turns the domain's abstract needs into real I/O: repository implementations, data sources, DTOs vs entities, mapping, source coordination, and error conversion. It's the concrete side of the interfaces defined in the domain (02_domain_layer.md).

Why this concept exists

The domain declares *what* it needs (interfaces) but must stay pure. Something has to actually fetch/persist data — that's the data layer. Keeping DTOs and I/O here (behind the interface) means the domain is insulated from API/DB shape changes: rename a JSON field or swap `sqflite` for Drift, and only the data layer's mappers/sources change. It's the Repository pattern (Module 05) realizing Dependency Inversion.

Real-world analogy

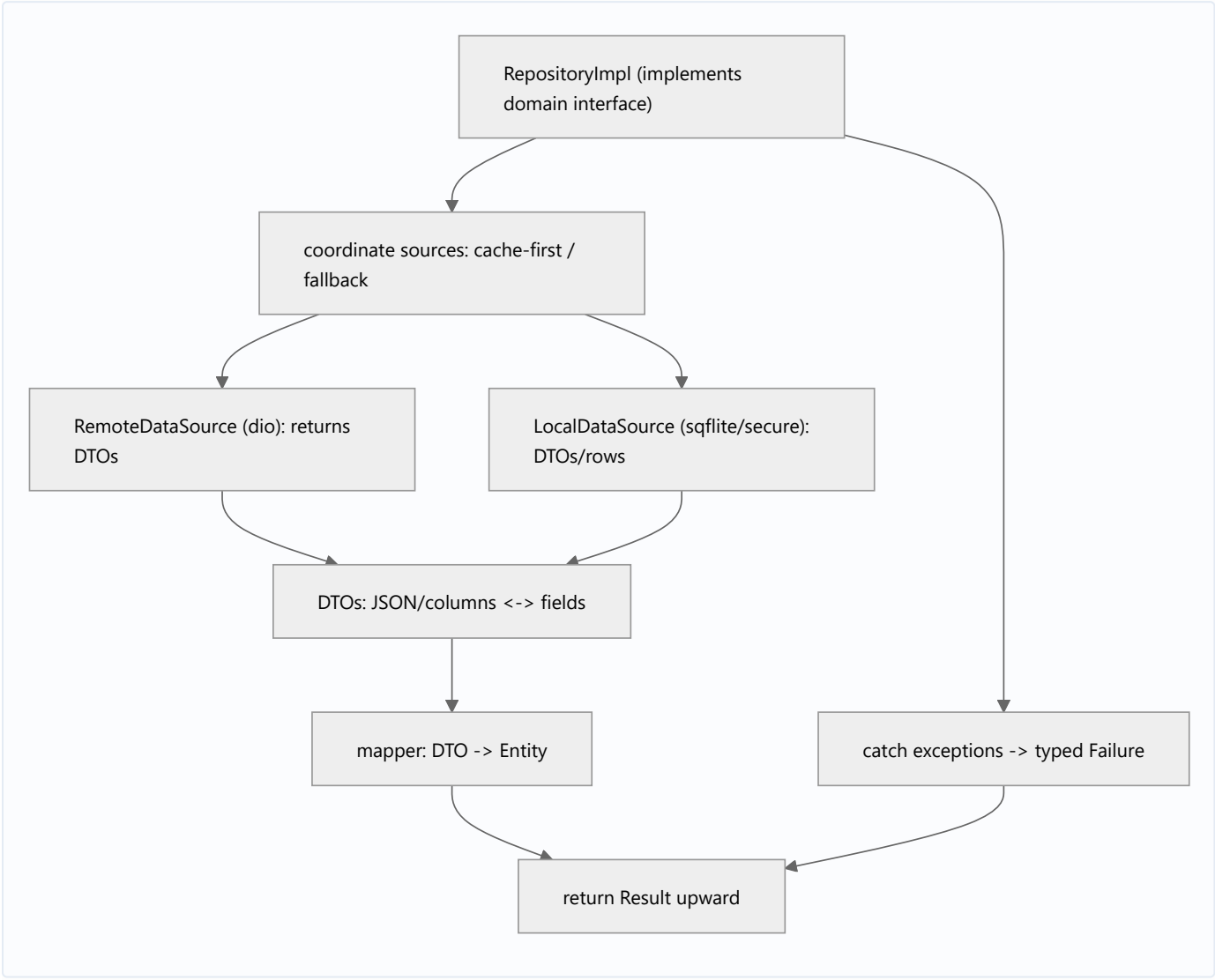
The data layer is the **warehouse + shipping department**: the domain (store) orders "one Order entity"; the department knows the messy details — which supplier (remote API), which shelf (local DB), the packing format (DTO/JSON) — and **unpacks it into the clean product** (entity) the store expects. The store never sees the packing slips (DTOs) or supplier contracts (HTTP); it just receives the product.

Problem Statement

Implement `OrderRepository`: fetch from a remote API (DTO ↔ entity), cache locally, serve cache-first with offline fallback, and convert network/DB exceptions into typed `Failure`s — returning `Result<Order>` (entities) to the domain. You'll build data sources, DTOs, mappers, and the

repository impl.

Internal Working



- **Repository implementation**: implements the **domain interface**; it's the coordinator — decides **cache-first vs network-first**, **offline fallback**, write-through, and combines sources. Returns `Result<Entity>`; the domain never sees sources/DTOs.
- **Data sources** (single-responsibility): **RemoteDataSource** (talks `dio`/GraphQL — Module 16) and **LocalDataSource** (`sqflite`/Drift/Hive/secure storage — Module 20/Module 15). Each returns **DTOs/raw**; they don't coordinate or map to entities (the repo does).
- **DTOs (data transfer objects)**: classes mirroring the **wire/DB shape** with `fromJson` / `toJson` / `fromRow` — they absorb API/DB quirks (naming, nullability, nesting). **Keep DTOs separate from entities**: the entity is the clean domain shape; the DTO is the serialization shape. Coupling them makes the domain fragile to API changes.
- **Mapping (DTO ↔ Entity)**: explicit **mappers** convert at the boundary (`dto.toEntity()`, `entity.toDto()`). This is where wire/DB representation → domain representation; the domain gets only entities.

- **Error conversion at the boundary**: the data layer is where I/O **throws** (`DioException`, `SocketException`, DB errors) — **catch here** and convert to **typed** `Failure`**s** (`NetworkFailure`, `NotFoundFailure`, `CacheFailure`), returning `Result` (Module 38). Domain/UI stay `try/catch`-free.
- **Source coordination patterns**: cache-first (return cache, refresh in background), network-first with cache fallback, write-through (write local + remote), offline outbox (Module 19). The repo owns this policy.
- **Swappability**: because the repo implements a domain interface, you can swap remote↔GraphQL, `sqflite`↔Drive, or add caching, without touching the domain.

Memory Representation

DTOs hold serialization-shaped data (may differ from entities — extra/nested/nullable fields); entities are the clean domain shape. Mappers produce entities from DTOs. The repo may hold source references + cache handles.

Compiler Behavior

The data layer imports frameworks (`dio` / `sqflite`) freely — it's the outer layer. It depends **inward** on the domain interface it implements; the domain doesn't import it.

Runtime Behavior

The repo coordinates async source calls (cache-first/fallback), maps DTOs→entities, and converts exceptions→ `Failure`. Cache hits are fast/offline; misses hit the network; failures become typed results.

Flutter Engine Behavior

None directly (pure Dart + plugins); networking/DB via their plugins.

Dart VM Behavior

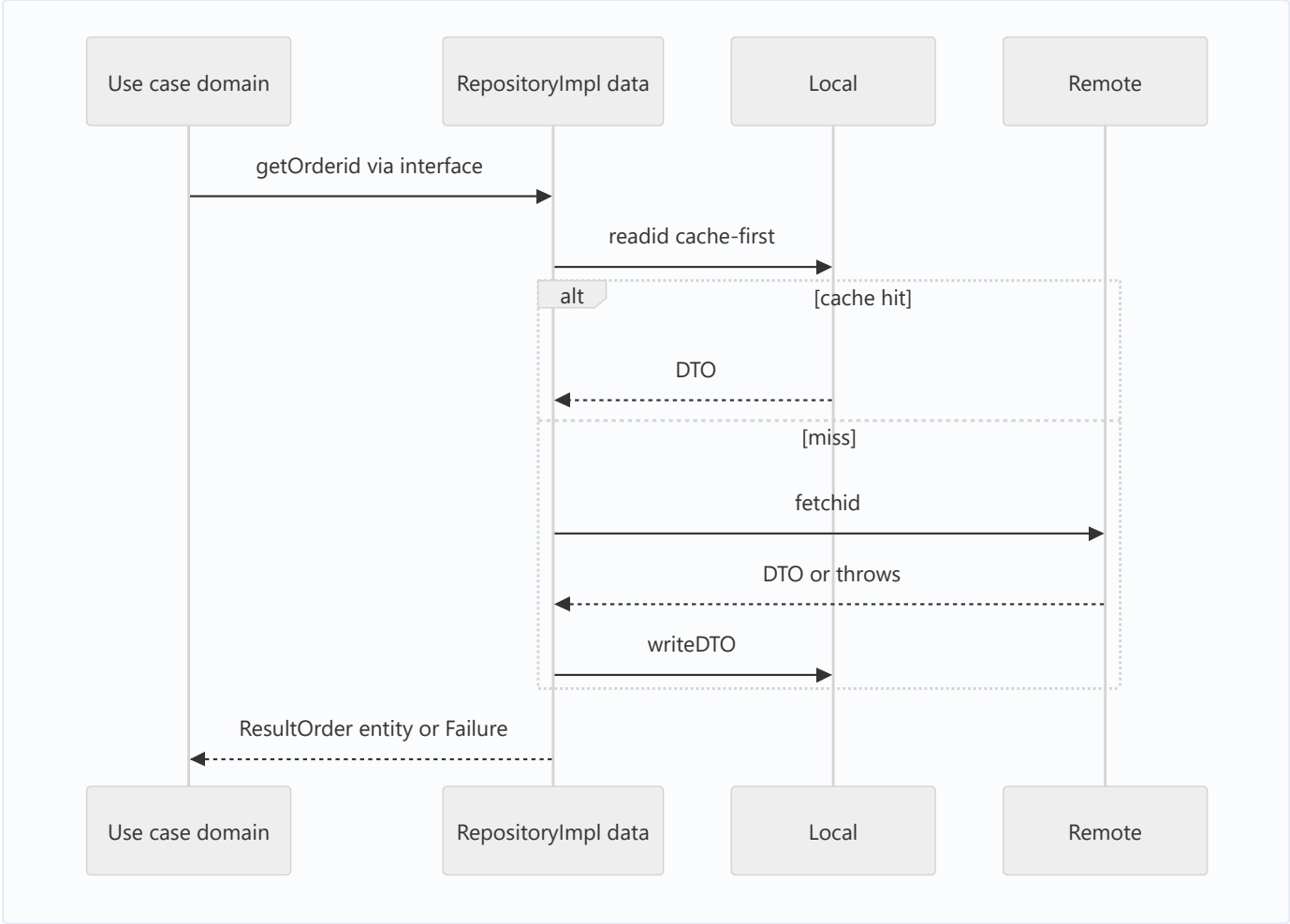
Async I/O on the I/O pool; heavy mapping/parsing can be isolate-offloaded for large payloads (Module 02).

Examples

```
// DTO — mirrors the API shape (JSON quirks live here), separate from the entity
class OrderDto {
  final String id; final List<Map<String, dynamic>> line_items;
  OrderDto(this.id, this.line_items);
```

```
  factory OrderDto.fromJson(Map<String, dynamic> j) =>
      OrderDto(j['id'] as String, (j['line_items'] as List).cast<Map<String, dynamic>>());
  Order toEntity() => Order(id, line_items                    // MAP DTO -> Entity
      .map((m) => LineItem(m['product_id'], m['qty'], m['unit_price_cents']))
      .toList());
}

// Data sources — single responsibility, return DTOs/raw
abstract class OrderRemote { Future<OrderDto> fetch(String id); }   // dio inside impl
abstract class OrderLocal { Future<OrderDto?> read(String id); Future<void> write(OrderDto d); }

// Repository impl — coordinate, map, convert errors -> Result<Entity>
class OrderRepositoryImpl implements OrderRepository {          // domain interface
  final OrderRemote remote; final OrderLocal local;
  OrderRepositoryImpl(this.remote, this.local);

  @override
  Future<Result<Order>> getOrder(String id) async {
    final cached = await local.read(id);                        // cache-first
    if (cached != null) return Success(cached.toEntity());
    try {
      final dto = await remote.fetch(id);                       // network
      await local.write(dto);                                   // write-through cache
      return Success(dto.toEntity());                           // map -> entity
    } on DioException catch (e) {
      if (e.response?.statusCode == 404) return Failure(NotFoundFailure('order'));
      return Failure(NetworkFailure());                         // convert exception -> Failure
    }
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Returning DTOs to the domain	Leaks wire/DB shape inward	Map DTO→entity; return entities
One class = entity + DTO	API change breaks domain	Separate DTO from entity
Letting exceptions escape to domain/UI	Breaks `Result`/purity	Convert to `Failure` at the boundary
Data sources coordinating/mapping	Muddled responsibilities	Sources return raw; repo coordinates/maps
Business logic in the repository	Belongs in domain	Keep repo to coordination/mapping/errors
No cache/offline policy	Poor UX/perf	Repo owns cache-first/fallback
Repo depends on concrete domain	Wrong direction	Repo implements the domain interface

Best Practices

- Repository impls **implement domain interfaces**, **coordinate sources** (cache-first/fallback/write-through), **map DTO↔entity**, **convert exceptions → typed `Failure`s**, and return `Result<Entity>`.

- Keep **DTOs separate from entities** (DTO = wire/DB shape); keep **data sources single-responsibility** (return raw/DTOs, no coordination/mapping).
- Put **no business logic** in the data layer (that's the domain) — only I/O, mapping, and source-policy; **offload heavy parsing** to isolates.
- Make sources **swappable** (remote↔GraphQL, `sqflite`↔Drift) behind the interface; own **offline/caching** policy here (Module 19).

Performance

Cache-first serves fast/offline; write-through keeps cache warm; isolate heavy JSON/DB mapping for large payloads. The repo's source-policy is the main perf lever (avoid redundant network). Mapping overhead is negligible vs I/O.

Advantages / Disadvantages

- **+** Domain insulated from API/DB changes, swappable sources, centralized caching/offline/error-conversion, testable (fake sources).
- **−** DTO/entity duplication + mapping boilerplate, more classes, discipline to keep logic out and errors converted.

Interview Questions

1. 🟢 **What does the data layer do?** — Implements domain repository interfaces using concrete data sources, mapping DTOs↔entities and converting exceptions to typed `Result`/`Failure`.
2. 🟢 **Why keep DTOs separate from entities?** — DTOs mirror the volatile wire/DB shape; separating them means API/DB changes don't ripple into the domain.
3. 🟡 **Where are exceptions converted to failures, and why there?** — At the data boundary (repo/sources), so the domain/UI stay `try/catch`-free and work with `Result`.
4. 🟡 **What's the difference between a data source and a repository?** — Data sources do single-responsibility I/O (return raw/DTOs); the repository coordinates sources, maps to entities, and applies cache/offline policy.
5. 🟡 **Why does the repo implement the domain interface (not vice versa)?** — To keep dependencies pointing inward (DIP) — the domain defines the need, the data layer satisfies it.
6. 🔴 **How does this layer enable swapping the backend/DB?** — Because it hides sources behind the domain interface, swapping remote/local implementations doesn't touch the domain.
7. 🔴 **Should the repository contain business rules?** — No — only coordination/mapping/error-conversion/caching; business rules belong in the domain (entities/use cases).

Senior Engineer Tips

- Always return entities (never DTOs) from repositories and convert exceptions to `Failure` at the boundary — those two rules keep the domain pure and the UI simple.
- Keep DTOs and entities separate even when they look identical today; the day the API adds a field or renames one, you'll be glad the domain didn't move.
- Let the repository own cache/offline policy and keep data sources dumb (raw I/O); muddling coordination into sources is where data layers rot.

Architect Perspective

The data layer is the adapter between the pure domain and the messy outside world — the Repository pattern realizing Dependency Inversion. By hiding sources/DTOs/errors behind domain interfaces and returning entities/ `Result`, it makes the backend/DB/caching a swappable detail and keeps the domain stable and testable. It's where offline-first, caching, and error-conversion policies live, feeding clean results inward (02_domain_layer.md, Module 19, Module 38).

Summary

- Data layer implements domain interfaces: coordinate sources (cache-first/fallback), map DTO↔entity, convert exceptions→typed `Failure`, return `Result<Entity>`.
- DTOs (wire/DB shape) separate from entities; data sources single-responsibility (raw I/O); no business logic here.
- Swappable sources behind the interface; owns caching/offline; heavy parsing → isolate.

Revision Notes

- RepositoryImpl implements domain interface; coordinates Remote/Local sources; maps DTO→entity; converts exceptions→ `Failure`; returns `Result<Entity>`.
- DTO ≠ entity (DTO = JSON/DB shape, `fromJson` / `toEntity`); sources return raw/DTOs (no coordination); no business logic in data layer.
- Boundary = error conversion + mapping; cache-first/fallback/write-through/outbox here; swappable behind interface; isolate heavy parse.

Practice Questions

1. Why return entities, not DTOs, from a repository?
2. Where and why do you convert exceptions to failures?
3. What's the division of labor between data sources and the repository?

Coding Questions

1. Implement a repository over remote + local sources with DTO→entity mapping.
2. Add cache-first with write-through and offline fallback.
3. Convert `DioException`/DB errors into typed `Failure`s returning `Result`.

Mini Project

Data layer slice (Flutter): Implement a domain repository interface with a `RepositoryImpl` coordinating a remote (`dio`) and local (`sqflite`/secure) source, separate DTOs with `fromJson`/`toEntity`, cache-first + write-through + offline fallback, and exception→`Failure` conversion returning `Result<Entity>`. Unit-test with fake sources. Acceptance: implements the domain interface (dependency inward); DTOs separate from entities + mapped at boundary; exceptions converted to typed failures; cache/offline policy in the repo; no business logic in data; sources single-responsibility; unit-tested with fakes.

The Presentation Layer (State, View Models, Domain→UI Mapping)

> The presentation layer is the outermost ring facing the user: **state holders** (bloc/cubit/notifier/view model) **call use cases**, receive domain **entities/** `Result` **s**, and **map them to immutable view state** (loading/data/empty/error + display-formatted fields) that **widgets render**. Widgets stay dumb (render state, emit events); the state holder holds no business rules (those are in the domain) and no I/O (that's in data) — it **orchestrates use cases and translates domain → UI**. This keeps UI swappable, testable without a device, and free of tangled logic.

Introduction

This file covers the UI-facing layer: how state holders call use cases, map domain results to view state, keep widgets thin, and handle errors/loading — closing the loop from domain (02_domain_layer.md) and data (03_data_layer.md) to the screen. State-management mechanics are in Module 11; here we focus on its role in Clean Architecture.

Why this concept exists

Widgets are volatile and hard to test; business rules and I/O must not live in them. The presentation layer provides a **thin translation layer** between the pure domain and the framework UI: it invokes application actions (use cases) and shapes their results into exactly what the view needs. This makes the UI a replaceable detail and the logic testable, upholding the dependency rule (presentation depends on domain, not vice versa).

Real-world analogy

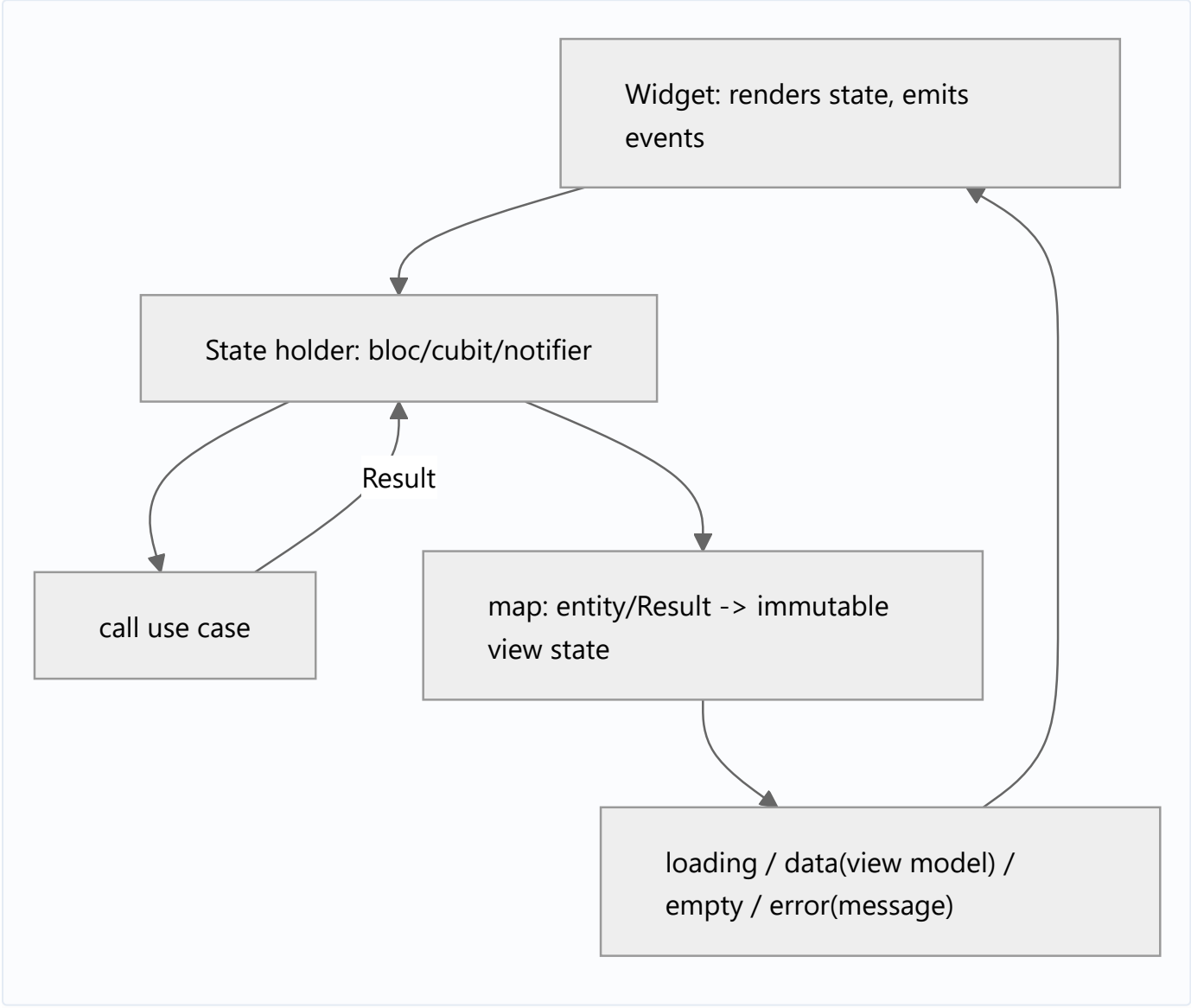
The state holder is a **translator + stage manager**: the audience (widgets) only sees the polished performance (view state). The manager takes the script's outcomes (domain results), decides what the audience sees at each moment (loading/data/error), and translates domain language into the audience's language (formatted strings, display models) — but never rewrites the script (business rules stay in the domain).

Problem Statement

Wire a screen: a state holder calls `GetProfile`, maps the `Result` to view state (loading → data with formatted fields / empty / error with a message + retry), and a thin widget renders it — with the state holder testable without a device and containing no business rules or I/O. You'll build the

state holder + view state + widget.

Internal Working



- **State holder** (bloc/cubit/riverpod notifier/VM): depends on **use cases** (injected via DI — Module 14), **calls** them on events, and **emits immutable view state**. It's the seam between domain and UI. It holds **no business rules** (domain) and **no I/O** (data) — only orchestration + mapping.
- **Domain → view state mapping**: convert **entities/** `Result` into **exactly what the UI needs** — often a **view model** with **display-formatted** fields (currency/date via `intl`, computed labels) and explicit **UI states** (loading/data/empty/error). Don't render entities directly if the view needs formatting/derived fields; don't leak `Result` / `Failure` types into widgets — map failures to **messages** (Module 38).
- **Thin widgets**: widgets **render state** and **emit events/intents** — no logic, no use-case calls, no formatting decisions beyond layout. Every async view handles **loading/data/empty/error** (Module 38).

- **Events → use cases**: user intents (tap/submit) become events the state holder handles by invoking the right **use case**, then mapping the result to new state. Retry re-invokes the use case.
- **Immutability**: view state is immutable (copyWith) so rebuilds are predictable and diffable (Module 11).
- **Testability**: the state holder is unit-tested by injecting **fake use cases** and asserting the **state sequence** (loading → data/error) — no widgets/device needed. Widget tests verify rendering per state.
- **View model vs entity**: entities are domain shape; **view models** are UI shape (formatted, presentation-only fields). Keep them distinct so the UI's needs don't pollute the domain.

Memory Representation

The state holder keeps the current immutable view state (+ references to injected use cases). View models hold display-ready, presentation-only data derived from entities. Widgets hold no logic state beyond ephemeral UI.

Compiler Behavior

Presentation imports Flutter + domain (use cases/entities) but **not** the data layer directly (it gets use cases via DI). Dependency points inward (presentation → domain).

Runtime Behavior

Event → state holder calls use case → maps `Result` → emits state → widget rebuilds. Loading shown while awaiting; errors mapped to messages; retry re-runs the use case.

Flutter Engine Behavior

Only this layer touches the engine (widgets/build/rebuild). State changes trigger rebuilds of listening widgets (Module 07/Module 11).

Dart VM Behavior

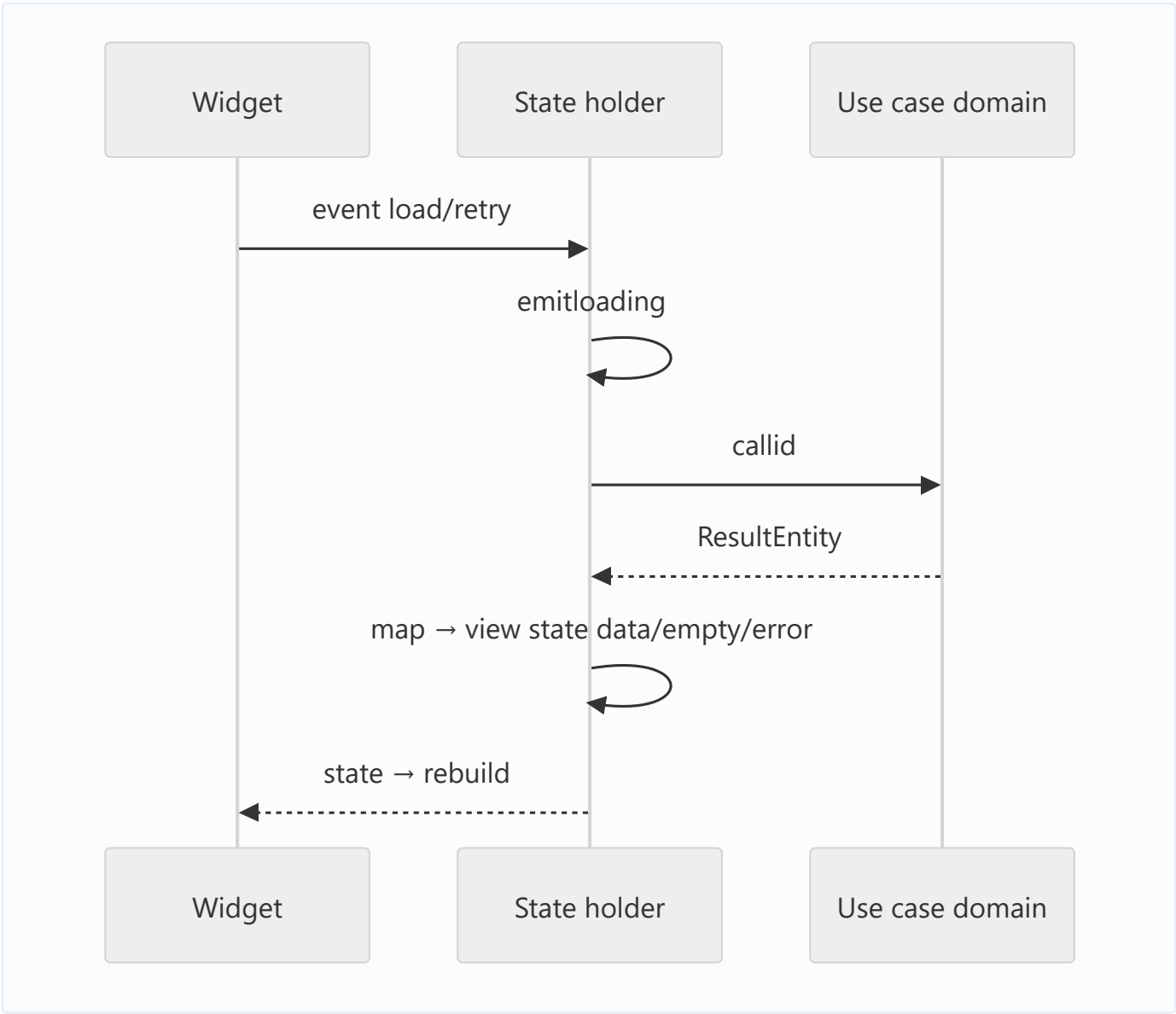
State-holder logic is plain Dart (testable without binding); only widget tests need the Flutter test binding.

Examples

```
// Immutable view state (UI shape) + presentation-only formatting
sealed class ProfileState {}
class ProfileLoading extends ProfileState {}
class ProfileData extends ProfileState { final ProfileVm vm; ProfileData(this.vm); }
class ProfileError extends ProfileState { final String message; ProfileError(this.message); }

class ProfileVm {                                  // view model: display-formatted, UI-only
  final String name; final String joinedLabel;
  ProfileVm(this.name, this.joinedLabel);
  factory ProfileVm.from(Profile p) =>             // map ENTITY -> view model
      ProfileVm(p.name, 'Joined ${DateFormat.yMMM().format(p.joinedAt)}');
}

// State holder (cubit): calls the USE CASE, maps Result -> state; no rules/IO
class ProfileCubit extends Cubit<ProfileState> {
  final GetProfile getProfile;                     // injected domain use case
  ProfileCubit(this.getProfile) : super(ProfileLoading());

  Future<void> load(String id) async {
    emit(ProfileLoading());
    final result = await getProfile(id);           // domain call
    emit(switch (result) {
      Success(:final value) => ProfileData(ProfileVm.from(value)),
      Failure(:final error) => ProfileError(messageFor(error)),   // failure -> message
    });
  }
}
```

```
// Thin widget: renders state, emits events (retry) — no logic
BlocBuilder<ProfileCubit, ProfileState>(builder: (c, s) => switch (s) {
  ProfileLoading() => const CenteredSpinner(),
  ProfileData(:final vm) => ProfileView(vm),       // renders view model
  ProfileError(:final message) =>
      ErrorView(message: message, onRetry: () => c.read<ProfileCubit>().load(id)),
});
```

Diagrams


Widget
State holder
Use case domain
event load/retry
emitloading
callid
ResultEntity
map → view state data/empty/error
state → rebuild
Widget
State holder
Use case domain

Common Mistakes

Mistake	Why	Fix
Business logic in widgets/state holder	Untestable, tangled	Rules in domain; holder only orchestrates/maps
I/O in the state holder	Wrong layer	Call use cases; I/O in data
Rendering entities directly (need formatting)	Domain shape ≠ UI needs	Map to a view model
Leaking `Result` / `Failure` into widgets	Couples UI to domain internals	Map failures to messages/state
Fat widgets with logic	Untestable, rebuild-heavy	Thin widgets: render + emit events
Missing loading/empty/error states	Poor UX / infinite spinner	Handle all UI states + retry
State holder depends on data layer	Violates dependency rule	Depend on use cases (domain) via DI

Best Practices

- State holders **call use cases** and **map domain results → immutable view state**; hold **no business rules** (domain) and **no I/O** (data).
- Map **entities → view models** (display-formatted, UI-only) and **failures → messages/states**; don't leak `Result` / `Failure` /entities' raw shape into widgets when the UI needs formatting.
- Keep **widgets thin** (render state, emit events); handle **loading/data/empty/error + retry**; use **immutable** view state.
- **Inject use cases via DI**; depend inward on the **domain**, not the data layer; **unit-test** the state holder with fake use cases (state-sequence assertions).

Performance

Immutable state + granular rebuilds keep the UI efficient (Module 11/Module 21). Mapping is cheap. The thin-widget/state-holder split localizes rebuilds and keeps logic off the widget tree. No architectural runtime cost.

Advantages / Disadvantages

- **+** Testable UI logic (no device), swappable UI/state lib, thin reusable widgets, clean loading/error UX, domain insulated from UI.
- **−** Mapping/view-model boilerplate, more files, discipline to keep logic out of widgets/holders.

Interview Questions

1. 🟢 **What is the presentation layer's job?** — Call use cases, map domain results to immutable view state, and render it via thin widgets — no business rules or I/O.
2. 🟢 **Why map entities to view models?** — The UI often needs formatted/derived fields; a view model shapes data for the view without polluting the domain entity.
3. 🟡 **Why shouldn't widgets call use cases or hold logic?** — Widgets are volatile/hard to test; keeping them thin (render + emit events) makes logic testable and UI swappable.
4. 🟡 **How do you handle failures in the UI?** — Map `Result` / `Failure` to a message/error state in the state holder (don't leak domain failure types into widgets); render error state + retry.
5. 🟡 **What does the state holder depend on, and why?** — Use cases (domain), injected via DI — dependencies point inward; it must not depend on the data layer.
6. 🔴 **How do you test the presentation layer without a device?** — Unit-test the state holder with fake use cases, asserting the emitted state sequence (loading→data/error); widget tests cover rendering.
7. 🔴 **Where does formatting (currency/date) belong?** — In the presentation mapping (view model), not the domain — the domain holds raw values; the view shapes them.

Senior Engineer Tips

- Keep the state holder a thin translator (use case → view state) and widgets dumb; the instant business logic or `dio` creeps into either, testability and swappability erode.
- Map failures to messages and entities to view models at this boundary so widgets never import domain/data types — that keeps the UI a true, replaceable detail.
- Test state holders by asserting the state sequence with fake use cases; it's fast, device-free, and catches the loading/error regressions users actually hit.

Architect Perspective

The presentation layer completes the dependency-rule loop: it depends inward on the domain (use cases), translating pure results into UI state while widgets stay a replaceable detail. This makes the UI swappable (Material↔Cupertino, bloc↔riverpod) and the logic testable, and it's where the app's chosen state-management pattern (MVVM etc.) plugs in — the same domain/data underneath (Module 11, Module 43, 05_clean_architecture_integration.md).

Summary

- Presentation = state holders calling use cases and mapping domain results → immutable view state; thin widgets render + emit events.

- Map entities→view models (formatted) and failures→messages; handle loading/data/empty/error + retry; no rules/I/O here.
- Inject use cases via DI (depend inward); unit-test the state holder with fakes (state-sequence).

Revision Notes

- State holder (bloc/cubit/notifier/VM) calls use cases → maps `Result`/entity → immutable view state (loading/data/empty/error); no business rules/IO.
- Entity → view model (display-formatted, UI-only); failure → message; widgets thin (render + events); handle all UI states + retry.
- Inject use cases via DI (depend on domain, not data); unit-test holder with fake use cases (assert state sequence).

Practice Questions

1. What belongs in the state holder vs the widget vs the domain?
2. Why map entities to view models and failures to messages?
3. How do you test presentation logic without a device?

Coding Questions

1. Build a cubit/notifier calling a use case and emitting loading/data/error state.
2. Map an entity to a formatted view model and a failure to a message.
3. Unit-test the state holder with a fake use case (assert state sequence).

Mini Project

Presentation slice (Flutter): Build a state holder (cubit/notifier) that calls `GetProfile`, maps the `Result` to immutable view state (loading/data(view model)/empty/error(message)) with `intl` formatting, and a thin widget rendering each state with retry. Inject the use case via DI; unit-test the state holder with a fake use case. Acceptance: holder only orchestrates/maps (no rules/IO); entity→view model + failure→message mapping; thin widget with all UI states + retry; depends inward on domain via DI; state-sequence unit-tested without a device.

Clean Architecture Integration (Capstone: A Full Vertical Slice)

> Put it together as a **vertical slice per feature**: `domain/` (entities, use cases, repository interfaces) + `data/` (DTOs, data sources, repository impls) + `presentation/` (state holders, view models, widgets), wired by **DI** that binds concrete data impls to domain interfaces and injects use cases into state holders — so a request flows **widget → state holder → use case → repository interface → impl → data source**, and every layer is **unit-testable in isolation** with fakes. Organize **by feature, not by layer-across-the-app**, and **right-size** the ceremony to the app's complexity.

Introduction

This module capstone assembles the domain, data, and presentation layers into one working, testable feature slice, wired by DI, with an honest tradeoff discussion. It shows the folder structure, the wiring, the end-to-end flow, and how to test each layer — the practical payoff of the dependency rule.

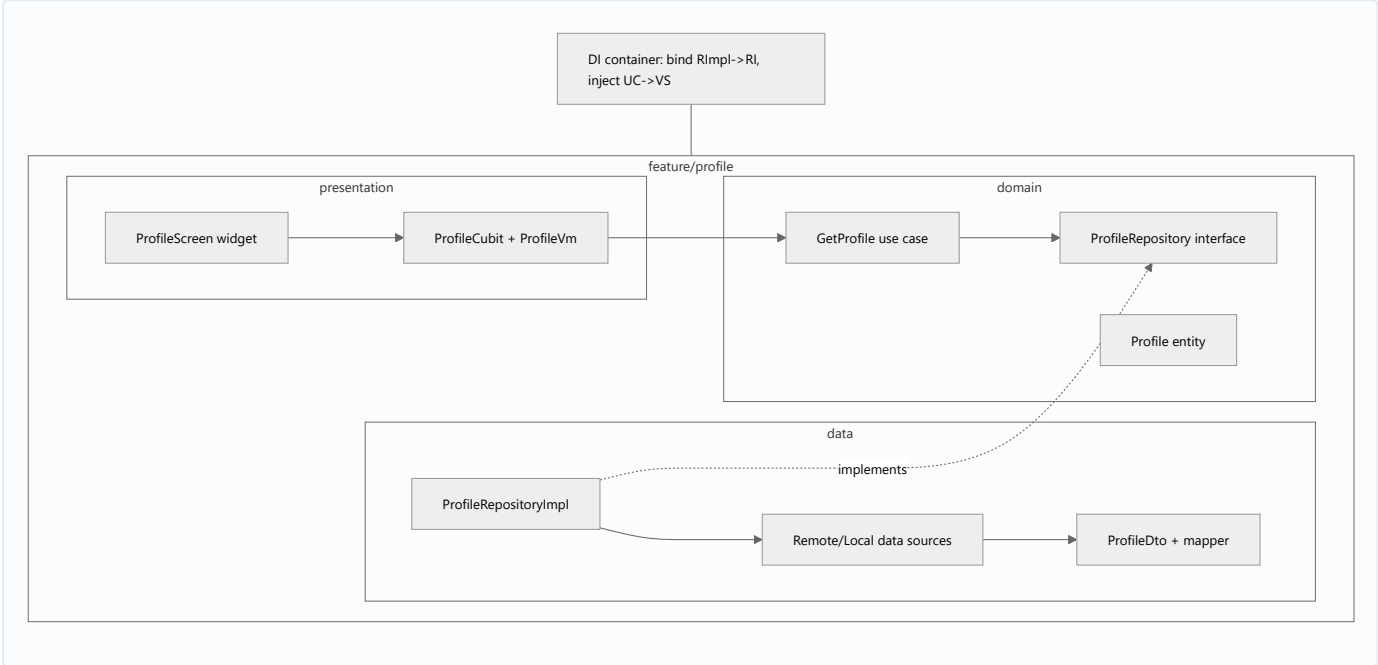
Why this concept exists

Understanding layers individually isn't enough; the value appears when they compose into a slice where dependencies point inward, DI supplies details, and each layer is independently testable. Organizing **by feature** (not global `data/` / `domain/` folders) keeps related code together and slices independently evolvable — the bridge to feature-first/modular architecture (Module 44/Module 45).

Real-world analogy

A vertical slice is a **fully-staffed department for one product line**: it has its own strategy (domain), suppliers/warehouse (data), and storefront (presentation), plus a **switchboard (DI)** connecting them. You can inspect/replace any part (test a layer with a stand-in) without shutting the company, and each department is organized around *its product*, not around "all strategy teams in one building."

Internal Working



- **Folder structure (feature-first)**: group by **feature**, each with `domain/`, `data/`, `presentation/` subfolders. Shared/cross-cutting (core `Result`, DI, theming) lives in a `core/` / `shared/` module. This keeps a slice cohesive and independently evolvable (Module 44).
- **DI wiring** (Module 14): register data sources → repository impl **bound to the domain interface** → use cases → state holder. Presentation gets **use cases** (not repos/sources); the domain interface is the seam. `get_it` / `injectable` / `riverpod` providers.
- **Request flow (runtime)**: widget event → state holder → **use case** → **repository interface** → **impl** (coordinate sources, map DTO→entity, convert errors → `Result`) → back up → state holder maps entity→view model → widget renders. **Dependencies** (compile-time) point inward throughout.
- **Testing per layer** (the payoff — Module 49):
  - **Domain**: use cases with **fake repositories** (pure Dart, fastest).
  - **Data**: repository impl with **fake data sources** (mapping/coordination/error-conversion).
  - **Presentation**: state holder with **fake use cases** (state-sequence); widget tests per state.
  - Each isolated, no device/network needed until integration tests.
- **Tradeoffs / right-sizing (be honest)**: this structure shines for **complex, long-lived, team** apps (testability, swap details, parallel work). For **small/prototype** apps it's **over-engineering** — collapse layers (e.g., skip use cases, repo returns directly) and grow into it. Avoid **anemic use cases** (pure pass-throughs) and **ceremony without payoff**.
- **Consistency**: one slice's structure is the template for all — predictable navigation, onboarding, and review.

Memory Representation

At runtime: the DI container holds singletons/factories; a slice's objects (sources, repo, use cases, state holder) are wired per the graph. Compile-time: the source-dependency graph is strictly inward.

Compiler Behavior

Domain compiles pure (no framework); data/presentation import frameworks but depend inward on domain interfaces/use cases. Import boundaries can be linted per feature.

Runtime Behavior

DI resolves the graph on startup/first use; requests flow outward→inward and results inward→outward; failures surface as `Result` →view state. Swapping a bound implementation (e.g., a fake in tests) changes behavior without touching callers.

Flutter Engine Behavior

Only presentation touches the engine; domain/data are plain Dart, so most tests run without the widget binding.

Dart VM Behavior

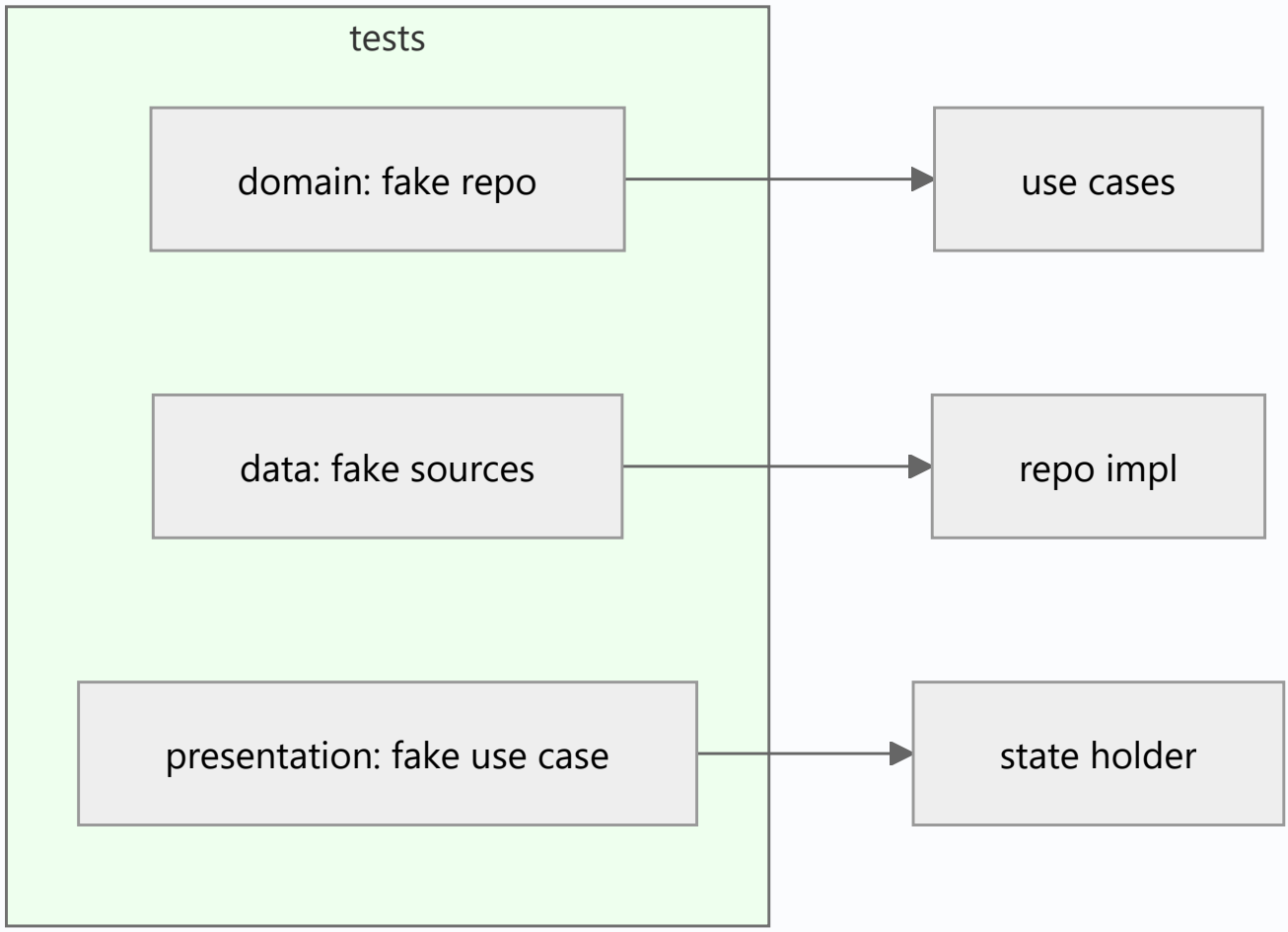
Domain/data/state-holder tests are fast pure-Dart; only widget tests use the Flutter test binding.

Examples

```
// DI wiring (get_it): bind impl -> domain interface; inject use case -> state holder
final getIt = GetIt.instance;
void registerProfileFeature() {
  getIt.registerLazySingleton<ProfileRemote>(() => ProfileRemoteImpl(getIt()));   // data source
  getIt.registerLazySingleton<ProfileLocal>(() => ProfileLocalImpl(getIt()));
  getIt.registerLazySingleton<ProfileRepository>(                                  // interface <- impl
      () => ProfileRepositoryImpl(getIt(), getIt()));
  getIt.registerFactory(() => GetProfile(getIt<ProfileRepository>()));             // use case
  getIt.registerFactory(() => ProfileCubit(getIt<GetProfile>()));                  // state holder
}
// Presentation depends on GetProfile only; never on ProfileRepositoryImpl/sources.
```

```
// Per-layer tests — each layer isolated with fakes, no device
test('use case: maps repo result', () async {
  final uc = GetProfile(FakeRepo(Success(Profile('1', 'Ada'))));
  expect(await uc('1'), isA<Success>());
});
test('repo: converts 404 -> NotFoundFailure', () async {
  final repo = ProfileRepositoryImpl(FakeRemote(throws404: true), FakeLocal());
  expect(await repo.getProfile('x'), isA<Failure>());
});
blocTest<ProfileCubit, ProfileState>('cubit: loading -> data',
  build: () => ProfileCubit(FakeGetProfile(Success(Profile('1','Ada')))),
  act: (c) => c.load('1'),
  expect: () => [isA<ProfileLoading>(), isA<ProfileData>()]);
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Organizing by layer across the whole app	Slices tangle; poor cohesion	Feature-first (domain/data/presentation per feature)
Presentation depending on data/impl	Violates dependency rule	Inject use cases; bind impl→interface in DI
Anemic pass-through use cases	Ceremony without value	Add real orchestration or collapse layers
Full ceremony on a tiny app	Over-engineering	Right-size; grow into it
Skipping per-layer tests	Loses the main payoff	Test each layer with fakes
Inconsistent slice structure	Hard to navigate/onboard	One template for all features
Business logic in data/presentation	Untestable, tangled	Rules in domain only

Best Practices

- Organize **by feature** (each with `domain/` / `data/` / `presentation/`) + a shared `core/`; keep the **slice structure consistent** across features.
- **Wire via DI**: bind data impls to **domain interfaces**, inject **use cases** into state holders; presentation never sees repos/sources.
- **Unit-test each layer in isolation** with fakes (domain/data pure Dart; presentation with fake use cases) — the concrete payoff of the dependency rule.
- **Right-size** the ceremony to complexity (collapse layers for simple apps; avoid anemic use cases); keep **business rules in the domain** only.

Performance

No runtime overhead beyond negligible DI/indirection. The payoff is engineering velocity: fast isolated tests, safe swaps, parallel team work. Right-sizing avoids paying boilerplate cost where it doesn't return value.

Advantages / Disadvantages

- **+** Fully testable slices (per-layer, no device), swappable details, feature cohesion, team-scalable, consistent + evolvable.
- **−** Boilerplate/mapping/DI wiring, more files, over-engineering risk on small apps, discipline to keep boundaries clean.

Interview Questions

1. 🟢 **How do you organize a Clean Architecture Flutter app?** — By feature (each with domain/data/presentation) + a shared core, wired via DI — not global layer folders.
2. 🟢 **What does DI bind in a slice?** — Data impls to domain repository interfaces, and use cases into state holders — so presentation depends only on the domain.
3. 🟡 **Trace a request through the layers.** — Widget event → state holder → use case → repository interface → impl (coordinate/map/convert) → `Result` back → state holder maps to view model → widget renders.
4. 🟡 **How is each layer tested in isolation?** — Domain with fake repos, data with fake sources, presentation with fake use cases — mostly pure Dart, no device.
5. 🟡 **Why feature-first over layer-first folders?** — Cohesion and independent evolvability of slices; the bridge to feature-first/modular architecture.
6. 🔴 **When is this structure over-engineering, and what do you do?** — For small/short-lived apps; collapse layers (skip use cases, repo returns directly) and grow into it — right-size to complexity.
7. 🔴 **What's an anemic use case and why avoid it?** — A pure pass-through with no orchestration/rules — ceremony without value; add real logic or remove the layer.

Senior Engineer Tips

- Build one exemplary slice end-to-end (domain→data→presentation + DI + per-layer tests) and use it as the template; consistency is what makes a Clean codebase navigable and onboardable.
- Bind implementations to interfaces in DI and inject use cases into state holders; if presentation can see a repository impl or data source, the wiring leaked.
- Be ruthless about right-sizing: collapse layers for simple features and expand for complex ones — the goal is testable, evolvable software, not maximal layering.

Architect Perspective

The vertical slice is Clean Architecture made real: the dependency rule enforced by DI, each layer independently testable, organized by feature for cohesion and evolvability. It's the foundation the rest of the architecture band refines — MVVM plugs into presentation, feature-first/modular scales the slicing, DDD deepens the domain — all sharing this boundary discipline. Right-sized, it yields software that teams can build in parallel, test thoroughly, and evolve for years (Module 43, Module 44, Module 46, Module 49).

Summary

- A feature is a vertical slice: `domain/` (entities/use cases/interfaces) + `data/` (DTOs/sources/impls) + `presentation/` (state holders/view models/widgets), wired by DI.
- DI binds impls→domain interfaces and injects use cases into state holders; requests flow outward→inward; dependencies point inward.
- Each layer unit-tested with fakes (the payoff); organize feature-first; right-size ceremony to complexity.

Revision Notes

- Structure: feature-first, each with domain/data/presentation + shared core; DI binds data impl → domain interface, injects use cases → state holders.
- Flow: widget → state holder → use case → repo interface → impl (coordinate/map/convert→ `Result`) → view model → widget; deps inward.
- Test per layer with fakes (domain/data pure Dart; presentation fake use cases); right-size (collapse for simple apps); no anemic use cases; rules in domain only.

Practice Questions

1. How does DI enforce the dependency rule in a slice?
2. How do you test each layer without a device?
3. When would you collapse layers, and how?

Coding Questions

1. Wire a full slice via DI (sources → repo impl → interface → use case → state holder).
2. Write per-layer tests with fakes (domain, data, presentation).
3. Refactor an over-layered simple feature to a right-sized version.

Mini Project

Full vertical slice (capstone — Flutter): Build a feature (e.g., "profile") end-to-end: `domain/` (entity + `GetProfile` use case + `ProfileRepository` interface), `data/` (DTO + remote/local sources + `ProfileRepositoryImpl` with mapping + `Result`), `presentation/` (cubit + view model + thin widget with loading/data/error+retry), wired via DI (impl→interface, use case→cubit), organized feature-first. Unit-test each layer with fakes and write a tradeoff note on right-sizing. Acceptance: dependency rule holds (domain pure; presentation depends on use cases only); DI binds impl→interface; request flows correctly; each layer unit-tested with fakes (no device); feature-first structure; tradeoffs/right-sizing documented; runs end-to-end.