38 · Error Handling

Flutter Practice Notes

Contents

38 · Error Handling

Introduction

This module covers handling failure well in Flutter/Dart: the **errors-vs-exceptions distinction** (bugs you shouldn't catch vs conditions you should), **functional error handling** (`Result` / `Either` /sealed classes — making failure explicit in the type system), **global error handling** (`FlutterError.onError`, `PlatformDispatcher.onError`, `runZonedGuarded`, widget error boundaries), and **recovery & user-facing errors** (retry/backoff, graceful degradation, mapping technical failures to helpful messages), tied together in a capstone. It threads through networking (Module 16), state management (Module 11), monitoring (Module 52), and clean architecture (Module 40).

Why this module exists

Real apps fail constantly — networks drop, APIs 500, files vanish, inputs are bad. The difference between a fragile app and a resilient one is a **deliberate error strategy**: knowing which failures are recoverable, modeling them explicitly so they can't be forgotten, catching uncaught errors globally (so nothing crashes silently), and turning failures into clear user experiences instead of red screens or infinite spinners. Ad hoc `try/catch` scattered everywhere is how apps become unmaintainable and users get stuck.

Module map

#	File	Topic	Level
1	01_errors_vs_exceptions.md	Dart `Error` vs `Exception`, throw/catch, custom exceptions, when to catch	🔵
2	02_result_and_either.md	`Result` / `Either` /sealed classes, explicit failure in types	🔴
3	03_global_error_handling.md	`FlutterError.onError`, `PlatformDispatcher.onError`, `runZonedGuarded`, error boundaries	🔴
4	04_recovery_and_user_errors.md	Retry/backoff, graceful degradation, error→message mapping, error UX	🟡
5	05_error_integration.md	Capstone: an error strategy across layers	🔴

> **Cross-references:** Networking failures/retries: Module 16. State (error states): Module 11. Monitoring/crash reporting: Module 52. Logging: Module 39. Clean architecture (where errors map): Module 40. Offline (failure as normal): Module 19.

Prerequisites

02 Advanced Dart (async/futures/streams), 11 State Management, 16 Networking.

What you'll be able to do after this module

- Distinguish errors (bugs) from exceptions (conditions) and catch the right things.
- Model expected failures explicitly with `Result` / `Either` /sealed classes.
- Catch all uncaught errors globally and add widget-level error boundaries.
- Implement retry/backoff, graceful degradation, and clear user-facing error messages.
- Compose a coherent, layered error strategy that's observable and testable.

Capstone

Error strategy slice: A feature where the data layer returns typed `Result` /failures (no leaking exceptions), the domain maps them to user-meaningful errors, the UI renders error states with retry, a global handler + `runZonedGuarded` capture anything uncaught (reported to monitoring), and a widget error boundary prevents a single widget crash from taking down the screen — all consistent and testable.

Summary

Error handling = a deliberate strategy, not scattered `try/catch` : distinguish bugs from conditions, model expected failures in the type system (`Result` / `Either`), catch everything uncaught globally, and turn failures into recovery + clear UX. Layer it (data→domain→UI), make it observable (monitoring/logging), and test the failure paths.

Errors vs Exceptions (What to Catch, What to Fix)

> Dart splits failures into two intents: `Error` signals a **programming bug** (a broken contract — `assert` failure, null deref, `StateError`, `ArgumentError`) that you should **fix, not catch**; `Exception` signals an **expected runtime condition** (network down, file missing, bad input) that you **should** handle. Both can be `throw`n and `catch`-caught (they implement `Throwable`-like behavior via `Object`), but the **convention** guides you: catch exceptions and recover; let errors crash in dev so you find the bug, and capture them globally in prod (03_global_error_handling.md).

Introduction

Before choosing a strategy, you must know *what kind* of failure you're dealing with. This file covers Dart's `Error` / `Exception` distinction, `throw` / `catch` / `on` / `finally`, custom exceptions, and the crucial judgment of **when to catch vs when to let it crash** — the semantic foundation for everything else.

Why this concept exists

Not all failures are equal. A `null` where you assumed non-null is a **bug** — swallowing it hides the defect and corrupts state. A failed HTTP call is a **normal condition** — crashing on it is user-hostile. Dart encodes this intent in two base types so code (and reviewers) can tell "fix this" from "handle this," preventing the twin anti-patterns of catching bugs and crashing on conditions.

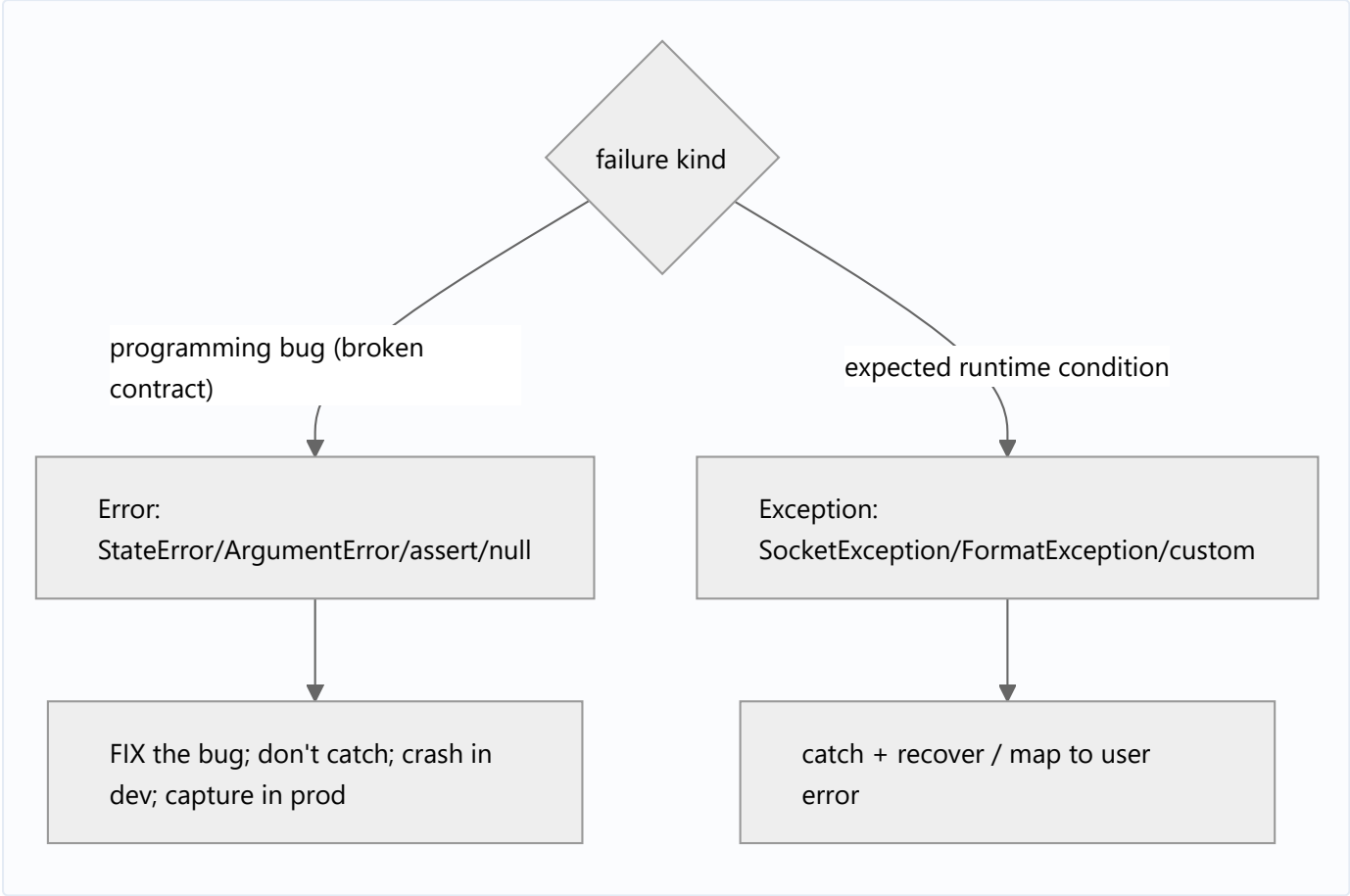
Real-world analogy

An `Exception` is a **flat tire on a road trip** — expected enough that you carry a spare and handle it (retry, reroute). An `Error` is the **engine was assembled wrong at the factory** — you don't "handle" it on the roadside; you fix the manufacturing defect. Wrapping the whole trip in "ignore all problems and keep driving" (catch-all that swallows) means you drive on a seized engine and crash worse later.

Problem Statement

For a data-fetch function, decide which failures to catch (timeout, 404, offline) and which to let propagate (a null you assumed present, a programmer misuse), define a meaningful custom exception, and use `try/on/catch/finally` correctly. You'll classify failures and write disciplined handling.

Internal Working



- `Error` (bugs): `StateError`, `ArgumentError`, `RangeError`, `TypeError`, `UnimplementedError`, `AssertionError`. These mean **the code is wrong** — throwing them is a signal to fix the caller/logic. **Generally don't catch** them; let them surface (in prod, capture globally + report — Module 52). Catching to "recover" usually hides a defect.
- `Exception` (conditions): `SocketException`, `TimeoutException`, `FormatException`, `HttpException`, and **your custom exceptions**. These are **expected** — **catch and handle** (retry, fallback, user message).
- `throw` / `catch`: `throw` any object (idiomatically an `Exception` / `Error`). `try { } on FooException catch (e) { } on Exception catch (e) { } catch (e, st) { }` — catch **specific** types first; `catch (e, st)` gets the stack trace. `finally` runs regardless (cleanup: close files/sinks/controllers).
- `rethrow`: preserve the original stack when re-throwing after partial handling (`catch (e) { log(e); rethrow; }`).
- **Custom exceptions**: implement `Exception`, add fields + a `toString()`; model domain failures (`NetworkException`, `NotFoundException`, `ValidationException`) so callers can branch and map to UX (04_recovery_and_user_errors.md).
- **When to catch**: catch **exceptions you can act on** at the level that can act; **don't** catch-all-and-swallow (hides bugs), and **don't** catch `Error`s to "handle" a bug. Catch broadly **only** at boundaries (a global handler / an isolate entry) to log + fail gracefully.

- **Async**: `Future` errors propagate to `await` (catch with try/catch) or `.catchError`; **stream** errors go to `onError`. Uncaught async errors need zone/global handling (03_global_error_handling.md).

Memory Representation

Exceptions/errors are objects with a message + (when caught with `catch (e, st)`) a `StackTrace`. `finally` guarantees cleanup runs. `rethrow` keeps the original stack; `throw e` inside a catch loses it.

Compiler Behavior

Dart has **no checked exceptions** — the compiler doesn't force you to declare/handle throwables; discipline (or `Result` types — 02_result_and_either.md) is on you. `assert`s are **debug-only** (stripped in release).

Runtime Behavior

An uncaught exception/error on the current call stack propagates up; if never caught it reaches the zone/global handler (or crashes). `finally` runs during unwinding. Async errors surface at the await/listen point.

Flutter Engine Behavior

Uncaught errors during build/layout/paint go to `FlutterError.onError`; uncaught async/platform errors go to `PlatformDispatcher.onError`/zone (03_global_error_handling.md).

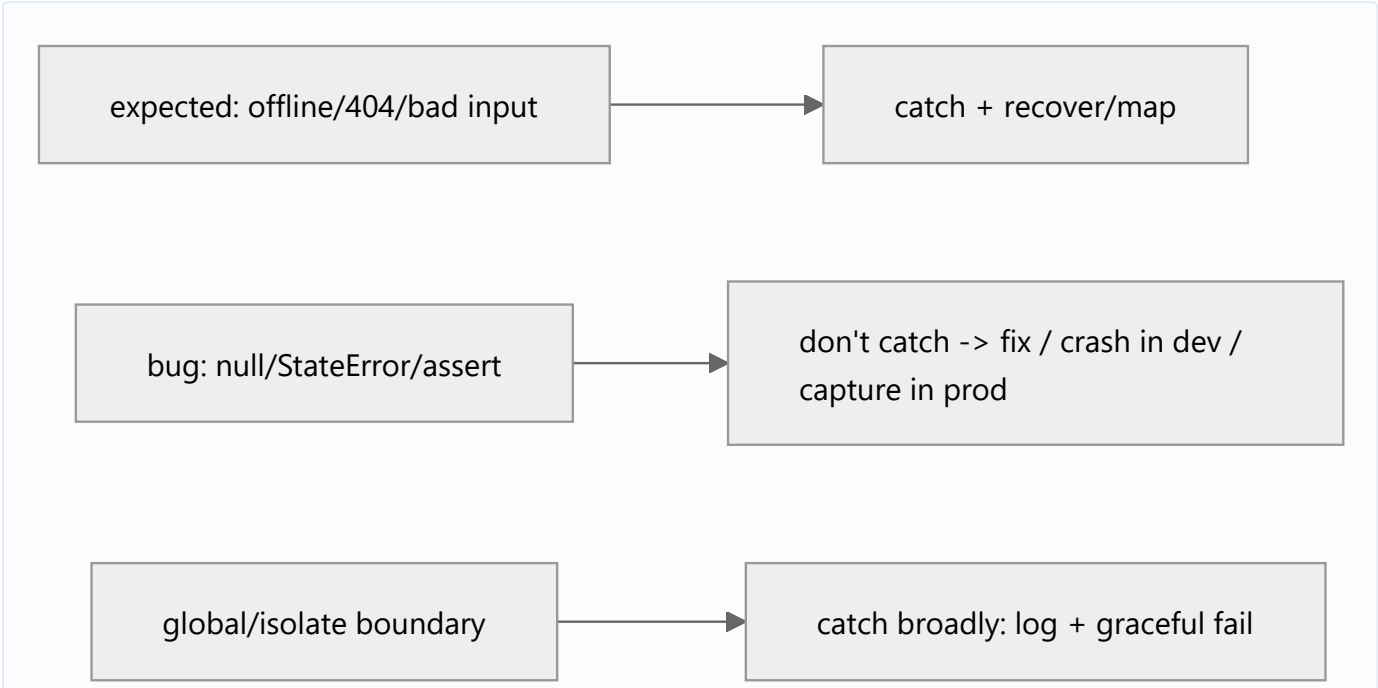
Dart VM Behavior

Errors/exceptions unwind the stack of the current isolate; an unhandled one in a background isolate terminates that isolate (not the app) unless handled.

Examples

```
// Custom exception (expected condition) — model domain failures
class NotFoundException implements Exception {
  final String resource;
  NotFoundException(this.resource);
  @override
  String toString() => '$resource not found';
}

Future<User> fetchUser(String id) async {
  try {
    final res = await api.get('/users/$id');
    if (res.statusCode == 404) throw NotFoundException('user $id');
    return User.fromJson(res.data);
  } on SocketException {
    throw NetworkException('offline');          // condition -> handle upstream
  } on FormatException catch (e, st) {
    log('bad JSON', error: e, stackTrace: st);  // condition (bad response)
    rethrow;                                     // preserve stack
  } finally {
    // cleanup that must always run (e.g., close a resource)
  }
}

// BUG (Error): don't "handle" it — fix the caller. Let it crash in dev.
int lastOf(List<int> xs) {
  if (xs.isEmpty) throw StateError('lastOf on empty list'); // programmer contract
  return xs.last;
}
// Anti-pattern: try { risky(); } catch (_) {}  // swallows bugs AND conditions silently
```

Diagrams



Common Mistakes

Mistake	Why it's bad	Fix
`catch (_) {}` swallowing everything	Hides bugs + conditions; silent failure	Catch specific types; act or rethrow
Catching `Error`s to "recover"	Masks a defect, corrupts state	Fix the bug; don't catch
Crashing on expected conditions	User-hostile	Catch exceptions + handle
`throw e` in a catch	Loses original stack	`rethrow`
No `finally` for cleanup	Leaks (files/sinks/controllers)	`finally` or try-with-cleanup
Assuming checked exceptions	Dart has none	Discipline / `Result` types
Ignoring async/stream errors	Uncaught → crash	try/catch await, stream `onError`, zones

Best Practices

- **Classify** failures: **catch** `Exception`**s** you can act on (offline/404/bad input); **don't catch** `Error`**s** — fix the bug (crash in dev, capture in prod).
- Catch **specific types first**; use `catch (e, st)` for stack traces, `rethrow` to preserve them, and `finally` for cleanup.
- Model **custom exceptions** for domain failures (with `toString`) so callers branch + map to UX; **never** `catch (_) {}` **silently**.

- Handle **async/stream** errors (await try/catch, stream `onError`); catch **broadly only at boundaries** (global/isolate) to log + degrade gracefully.

Performance

Throwing/catching is cheap relative to I/O but **not free** — don't use exceptions for **normal control flow** in hot paths (prefer `Result` / nullable returns — 02_result_and_either.md). `finally` is negligible.

Advantages / Disadvantages

- **+** Clear intent (bug vs condition), stack traces, cleanup via `finally`, custom domain modeling, propagation.
- **−** No compiler enforcement (easy to forget), easy to misuse (swallow/over-catch), exceptions-as-control-flow is a smell, async error handling has extra rules.

Interview Questions

1. 🟢 **What's the difference between `Error` and `Exception` in Dart?** — `Error` signals a programming bug (fix it, don't catch); `Exception` signals an expected condition (catch and handle).
2. 🟢 **Should you catch `Error`s?** — Generally no — catching a bug hides the defect; let it crash in dev and capture globally in prod.
3. 🟡 **Why is `catch (_) {}` an anti-pattern?** — It silently swallows both bugs and conditions, hiding failures; catch specific types and act or `rethrow`.
4. 🟡 **`throw e` vs `rethrow` in a catch?** — `rethrow` preserves the original stack trace; `throw e` resets it.
5. 🟡 **Does Dart have checked exceptions?** — No — the compiler doesn't force handling; you rely on discipline or `Result` types.
6. 🔴 **When is a broad catch-all appropriate?** — Only at boundaries (global handler, isolate entry) to log and fail gracefully — not sprinkled through business logic.
7. 🔴 **Why avoid exceptions for normal control flow?** — They're costlier than returns and obscure the happy path; use `Result`/nullable for expected outcomes.

Senior Engineer Tips

- Treat `catch (_) {}` as a code-review blocker — silent catches are where bugs go to hide and users get stuck spinners.
- Let programming errors crash loudly in development (asserts, no catch); you want to *find* them, not paper over them, and prod captures them globally.

- Use `rethrow` + logging when you handle partially, and always `finally` your cleanup; lost stack traces and leaked resources are the recurring pain points.

Architect Perspective

The error/exception distinction is the vocabulary the whole strategy is built on: it decides what's modeled as recoverable (exceptions → `Result`/UX) vs what's a defect (errors → crash/capture). Getting this classification right at each layer — and forbidding silent catches — is what lets a codebase handle failure deliberately rather than accidentally, feeding the explicit-failure modeling and global handling that follow (02_result_and_either.md, 03_global_error_handling.md).

Summary

- `Error` = bug (fix, don't catch; crash dev, capture prod); `Exception` = condition (catch + handle).
- Catch specific types first, `catch (e, st)` for stacks, `rethrow` to preserve, `finally` for cleanup; never `catch (_) {}` silently.
- Model custom exceptions; handle async/stream errors; broad catch only at boundaries; don't use exceptions for normal control flow.

Revision Notes

- `Error` (StateError/ArgumentError/assert/null) = programming bug → don't catch; `Exception` (Socket/Timeout/Format/custom) = condition → catch.
- `try { } on T catch (e, st) { } catch (e, st) { } finally { }`; `rethrow` preserves stack; `catch (_) {}` = anti-pattern.
- No checked exceptions; async → await try/catch, stream `onError`, uncaught → zone/global; don't use exceptions as control flow.

Practice Questions

1. Which failures in a fetch should you catch vs let crash?
2. Why use `rethrow` instead of `throw e`?
3. Why is exceptions-as-control-flow discouraged?

Coding Questions

1. Write a custom `Exception` + a function that throws it on a 404.
2. Use `on` / `catch` / `finally` to handle offline + cleanup correctly.
3. Fix a `catch (_) {}` anti-pattern into specific handling + rethrow.

Mini Project

Disciplined try/catch (Flutter): Refactor a data function to classify failures — catch `SocketException` / `TimeoutException` /404 into typed custom exceptions (handled upstream), let programmer errors propagate, use `rethrow` + `finally` correctly, and remove any silent `catch (_) {}` . Acceptance: exceptions vs errors correctly classified; specific `on` clauses; stack traces preserved (`rethrow` / `catch (e, st)`); cleanup in `finally` ; no silent catches; async errors handled.

Result / Either (Explicit Failure in the Type System)

> Instead of throwing for **expected** failures (which the compiler can't force callers to handle), return the outcome as a value: a `Result<T>` / `Either<Failure, T>` — usually a **sealed class** with `Success(value)` / `Failure(error)` variants — so failure is **part of the return type**, callers **must** handle both cases (exhaustive `switch`), and business logic stays free of `try/catch`. Exceptions still exist for bugs and at boundaries; `Result` is for the *modeled, recoverable* failures that flow through your domain.

Introduction

This file covers functional error handling: representing success-or-failure as a typed value (`Result` / `Either`) via sealed classes, why it beats throwing for expected failures, and how to compose it (map/flatMap/fold) across layers. It's the type-system complement to the errors/exceptions distinction (01_errors_vs_exceptions.md).

Why this concept exists

Dart has **no checked exceptions**, so a thrown exception is invisible in a function's signature — callers forget to handle it and the app crashes. Encoding failure in the **return type** makes it impossible to ignore: the compiler forces you to handle both branches. This turns "hope everyone remembers the try/catch" into "the types guarantee it," and keeps happy-path logic clean.

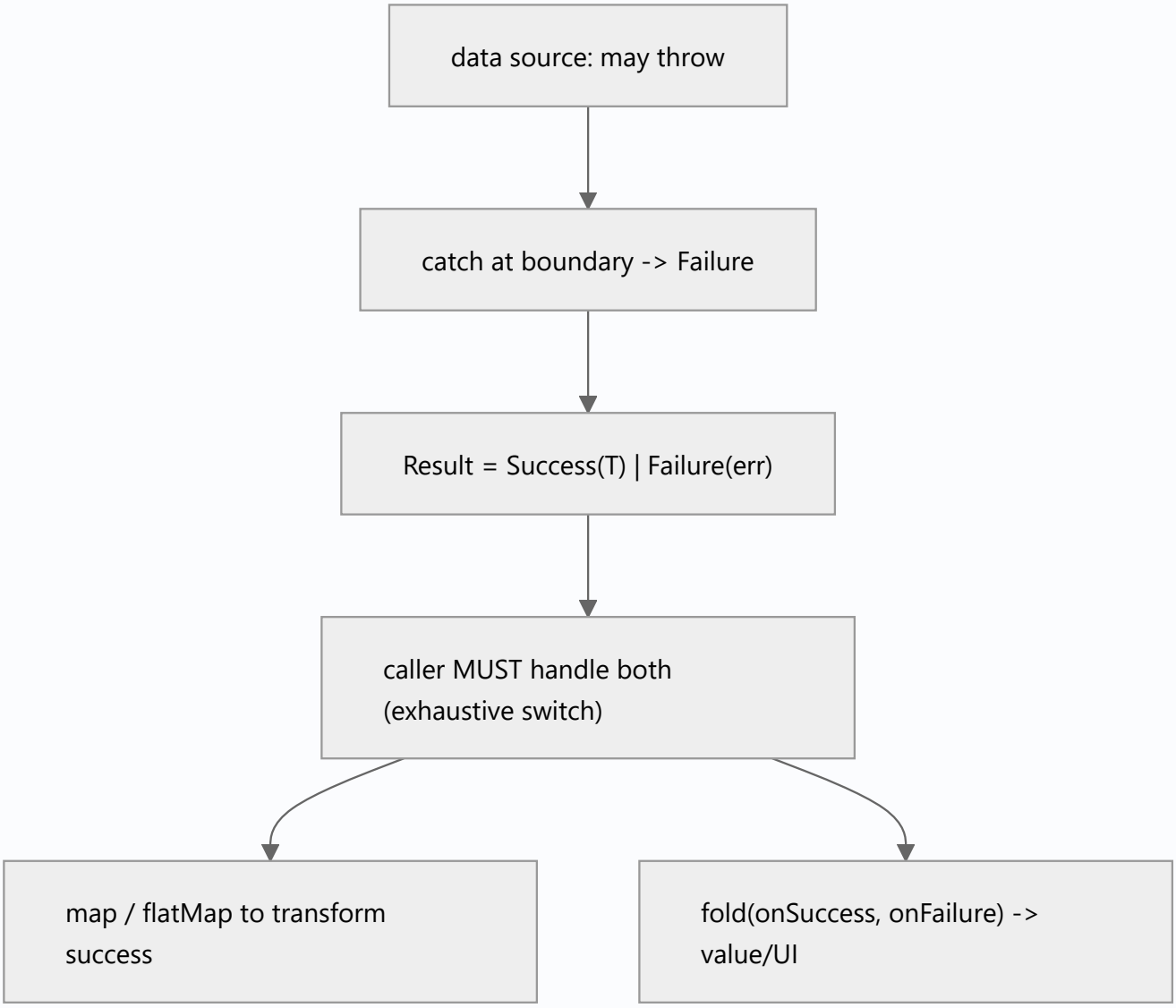
Real-world analogy

Throwing is like a **fire alarm** — it interrupts everything and someone up the chain must catch it (but nothing forces them to). A `Result` is like a **delivery that arrives either as the package or as a "delivery failed" slip in the same box** — you literally cannot open the box without seeing which one you got, so you always decide what to do. Failure is handed to you, not hurled past you.

Problem Statement

Make a repository method's failures impossible to ignore: return `Result<User>` (or `Either<Failure, User>`) instead of throwing, map network/parse exceptions into typed `Failure`s at the boundary, and force the caller (bloc/UI) to handle success and each failure explicitly. You'll define a sealed `Result`, convert exceptions at the edge, and consume it exhaustively.

Internal Working



- **Sealed `Result` / `Either`**: a **sealed class** with two subtypes — `Success<T>(value)` and `Failure(error)` (or `Either<L,R>` with `Left` =failure, `Right` =success). Sealed → the compiler enforces **exhaustive** handling in `switch` (Dart 3 patterns). Use a package (`dartz` , `fpdart` , `result_dart` , `multiple_result`) or a tiny hand-rolled type.
- **Convert at the boundary**: the **data source** (which genuinely throws — network/DB/file) is wrapped so its exceptions become **typed `Failure` s** (`NetworkFailure` , `NotFoundFailure` , `ValidationFailure`) — a small `try/catch` at the edge, then `Result` flows inward. Domain/UI never `try/catch` .
- **Typed failures**: model failures as a **sealed `Failure` hierarchy** so callers branch and map to UX (04_recovery_and_user_errors.md) — richer than a string.
- **Composition**: `map` (transform success), `flatMap` / `then` (chain fallible steps, short-circuiting on the first failure), `fold` / `when` / `switch` (collapse to a value/UI). Chains read linearly without nested try/catch.

- **When to use which**: `Result` / `Either` for **expected, recoverable** failures crossing layers (the domain's normal outcomes); **exceptions** for **bugs** and unforeseeable errors, caught at boundaries/globally. Don't wrap *everything* in `Result` (bugs should crash) and don't throw for *expected* failures.
- **Async**: return `Future<Result<T>>`; still awaited normally, but the value carries the outcome. No uncaught-exception surprises.
- **Tradeoff**: more ceremony/boilerplate vs guaranteed handling + clean happy path; teams pick a convention and apply it consistently.

Memory Representation

A `Result` is a small object holding either a value or an error (one of two variants). Sealed hierarchies enable exhaustive pattern matching at compile time. No stack unwinding (unlike throwing).

Compiler Behavior

Sealed classes + `switch` give exhaustiveness checking — omit a case and it won't compile. This is the whole point: the type system *forces* failure handling that thrown exceptions can't.

Runtime Behavior

No exception propagation for modeled failures — control flows normally, carrying the outcome. Chains short-circuit on the first `Failure` (with `flatMap`). Faster/predictable vs throw/catch for expected paths.

Flutter Engine Behavior

Not applicable; pure Dart pattern.

Dart VM Behavior

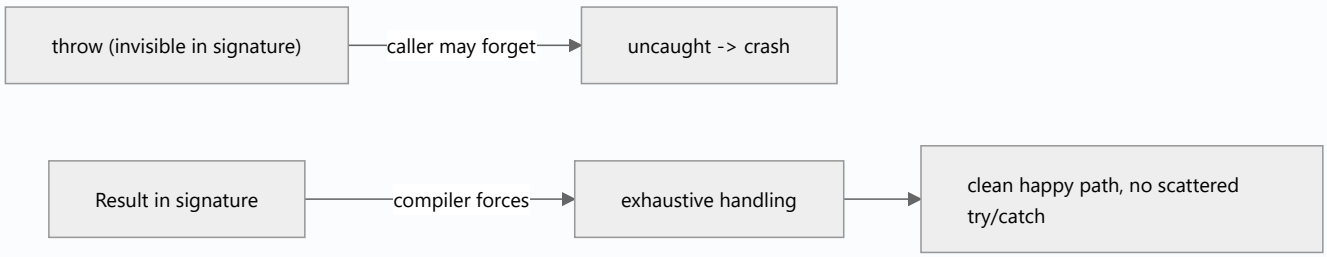
No stack unwinding for `Result` failures (cheaper than throwing). Bugs still throw as `Error`s.

Examples

```
// Minimal sealed Result (or use dartz/fpdart/result_dart)
sealed class Result<T> {
  const Result();
  R fold<R>(R Function(T) onOk, R Function(Failure) onErr) => switch (this) {
        Success<T>(:final value) => onOk(value),
```

```
        Failure(:final error) => onErr(error as Failure), // (typed in real code)
      };
}
class Success<T> extends Result<T> { final T value; const Success(this.value); }
class Failure<T> extends Result<T> { final Object error; const Failure(this.error); }

// Sealed failure hierarchy (branchable, UX-mappable)
sealed class AppFailure {}
class NetworkFailure extends AppFailure {}
class NotFoundFailure extends AppFailure { final String what; NotFoundFailure(this.what); }

// Repository: convert exceptions -> Result AT THE BOUNDARY
Future<Result<User>> getUser(String id) async {
  try {
    final res = await api.get('/users/$id');
    if (res.statusCode == 404) return Failure(NotFoundFailure('user $id'));
    return Success(User.fromJson(res.data));
  } on SocketException {
    return Failure(NetworkFailure());       // expected failure as a value
  }
}

// Caller MUST handle both — no forgotten try/catch, exhaustive switch
final result = await repo.getUser(id);
final ui = switch (result) {
  Success(:final value) => ProfileView(value),
  Failure(error: NetworkFailure()) => const OfflineBanner(),
  Failure(error: NotFoundFailure(:final what)) => NotFound(what),
  Failure() => const GenericError(),
};
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Throwing for expected failures	Callers forget to catch → crash	Return `Result` / `Either`
Wrapping bugs in `Result` too	Hides defects that should crash	`Result` for conditions; let bugs throw
`try/catch` in domain/UI	Leaks boundary concern inward	Convert to `Result` at the data boundary
String errors, not typed failures	Can't branch/map to UX	Sealed `Failure` hierarchy
Non-sealed result type	No exhaustiveness check	Sealed classes + `switch`
Ignoring the `Failure` branch	Silent wrong behavior	Handle both (compiler-enforced if sealed)
Over-abstracting (heavy FP)	Team friction	Keep it simple + consistent

Best Practices

- Return **`Result` / `Either`** (sealed) for **expected, recoverable** failures so the **compiler forces** handling; keep **exceptions for bugs** and boundaries.
- **Convert exceptions → typed `Failure`s at the data boundary**; keep domain/UI **free of `try/catch`**; model failures as a **sealed hierarchy** for branching/UX.
- **Compose** with `map` / `flatMap` / `fold` (short-circuit on failure) for linear, clean flows; consume with **exhaustive `switch`**.
- Pick **one convention** (package or tiny type) and apply consistently; don't over-engineer or wrap *everything*.

Performance

`Result` avoids stack unwinding — cheaper/more predictable than throwing for expected paths (relevant in hot/loopy code). Slight allocation per result; negligible vs I/O. The bigger win is correctness (no forgotten handling), not raw speed.

Advantages / Disadvantages

- **+** Failure explicit in the type; compiler-enforced handling; clean happy path; branchable typed failures; no uncaught surprises; cheaper than throwing.
- **−** Boilerplate/ceremony, a new convention to learn, easy to over-apply (wrapping bugs), composition operators have a learning curve.

Interview Questions

1. 🟢 **Why return `Result` / `Either` instead of throwing?** — Dart has no checked exceptions, so thrown failures are invisible and forgettable; encoding failure in the return type forces callers

to handle it.

2. 🟢 **What makes handling exhaustive?** — A sealed `Result` / `Failure` hierarchy + `switch` — the compiler errors if a case is missing.
3. 🟡 **Where do you convert exceptions to `Result`?** — At the data boundary (where I/O genuinely throws); domain/UI then stay `try/catch`-free.
4. 🟡 **When do you still use exceptions?** — For programming bugs and truly unforeseeable errors, caught at boundaries/globally — `Result` is for expected, recoverable failures.
5. 🟡 **What do `map` / `flatMap` / `fold` do?** — Transform success, chain fallible steps (short-circuiting on failure), and collapse to a value/UI — enabling linear flows without nested try/catch.
6. 🔴 **Why is `Result` cheaper than throwing for expected paths?** — No stack unwinding — control flows normally carrying the outcome.
7. 🔴 **What's the downside, and how do you manage it?** — Boilerplate/ceremony; manage by choosing one library/convention, applying it consistently, and not wrapping bugs.

Senior Engineer Tips

- Draw the line clearly: `Result` for expected domain failures, exceptions for bugs — wrapping everything in `Result` (or throwing everything) both defeat the purpose.
- Convert at the boundary once and keep the inner layers throw-free; a `try/catch` deep in a bloc is a sign the boundary leaked.
- Model failures as a sealed hierarchy from day one so the UI can branch (offline vs not-found vs generic); string errors force stringly-typed UX later.

Architect Perspective

`Result` / `Either` is how clean architecture makes failure a first-class, type-checked concern: data sources convert exceptions to typed failures at the boundary, the domain composes `Result`s, and the UI exhaustively renders outcomes — no scattered `try/catch`, no forgotten handling. It pairs with the errors/exceptions distinction (bugs still throw) and feeds directly into user-facing error mapping and testable failure paths (01_errors_vs_exceptions.md, 04_recovery_and_user_errors.md, Module 40).

Summary

- Return sealed `Result` / `Either` for expected failures so the compiler forces exhaustive handling; keep exceptions for bugs/boundaries.
- Convert exceptions → typed `Failure`s at the data boundary; domain/UI stay `try/catch`-free; compose with map/flatMap/fold.

- Cheaper than throwing for expected paths; pick one convention; don't wrap bugs or over-engineer.

Revision Notes

- Sealed `Result<T>` = `Success(value)` | `Failure(error)` (or `Either<L,R>`); `switch` → exhaustive (compiler-enforced).
- Convert exceptions → typed `Failure` at data boundary; domain/UI `try/catch`-free; sealed `Failure` hierarchy for UX branching.
- `map` / `flatMap` / `fold` (short-circuit on failure); `Future<Result<T>>` for async; `Result` for expected failures, exceptions for bugs; one convention (dartz/fpdart/result_dart/hand-rolled).

Practice Questions

1. Why is a thrown exception "invisible" and a `Result` not?
2. Where should exception→`Result` conversion happen?
3. When should you still throw instead of returning `Result`?

Coding Questions

1. Define a sealed `Result<T>` + sealed `Failure` hierarchy.
2. Convert a throwing repository method to return `Future<Result<User>>`.
3. Consume it with an exhaustive `switch` rendering per-failure UI.

Mini Project

Typed failures with `Result` (Flutter): Refactor a repository to return `Future<Result<T>>` with a sealed `Failure` hierarchy (Network/NotFound/Validation/Generic), converting exceptions at the data boundary, keeping domain/UI `try/catch`-free, composing with `map` / `flatMap`, and consuming via exhaustive `switch` that renders per-failure UI. Acceptance: failures explicit in return types (compiler-enforced handling); exceptions converted at boundary; typed sealed failures branched in UI; no domain/UI `try/catch`; bugs still throw; one consistent convention.

Global Error Handling (Catch-All Safety Nets)

> No matter how careful your local handling, some errors slip through — so install **global catch-alls**: `FlutterError.onError` (errors inside the framework: build/layout/paint/gesture callbacks), `PlatformDispatcher.instance.onError` (uncaught async/platform errors), and optionally wrap `runApp` in `runZonedGuarded` (catches uncaught errors in that zone). Route all of them to your **logging/crash reporter** (Module 52) so nothing fails silently. Separately, use a **widget `ErrorWidget.builder` /error boundary** so a single widget's build error shows a fallback instead of crashing the whole screen.

Introduction

Global handlers are the safety net beneath local `try/catch` and `Result`s. This file covers the three Flutter/Dart global hooks, how they differ, wiring them to crash reporting, and widget-level error boundaries — ensuring uncaught errors are captured, reported, and (where possible) contained rather than crashing or vanishing.

Why this concept exists

Local handling can't cover everything — a bug in a build method, an unawaited future's error, a platform callback failure. Without global hooks these either crash the app (bad UX, if uncaught) or disappear silently (worse — you never learn about them). Flutter/Dart provide layered hooks so **every** uncaught error is caught somewhere, logged, and reported.

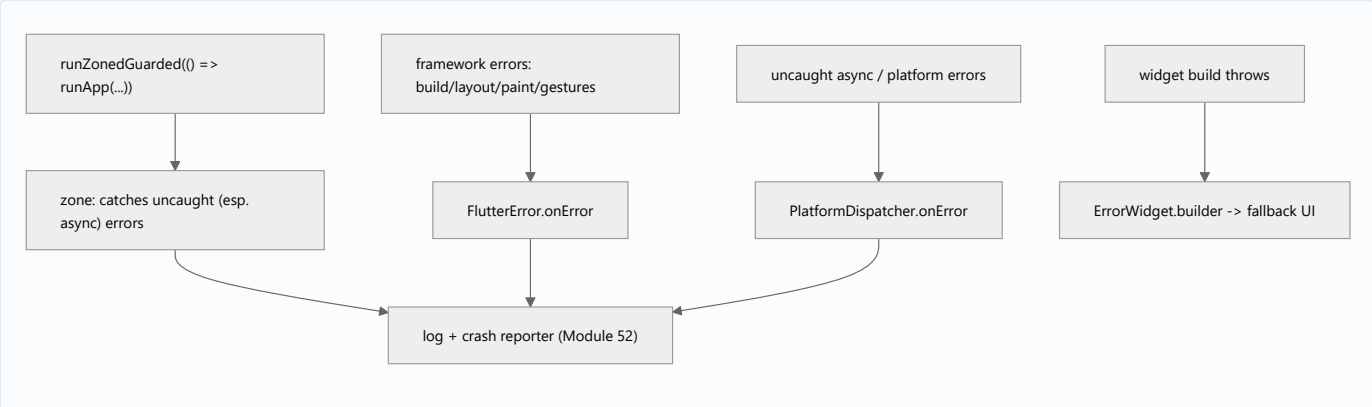
Real-world analogy

Local `try/catch` is each worker handling problems at their station. Global handlers are the **building's fire alarm + sprinklers + security desk**: if something gets past everyone, the alarm still logs it and the sprinklers contain the damage. The widget error boundary is a **firebreak wall** — a fire in one room (widget) doesn't burn down the whole floor (screen); that room shows "under maintenance" instead.

Problem Statement

Ensure that a build-method exception, an unawaited failing future, and a platform-channel error are all **captured and reported** (not silently lost or crashing), and that one broken widget shows a friendly fallback instead of a red screen taking down the page. You'll wire `FlutterError.onError`, `PlatformDispatcher.onError`, `runZonedGuarded`, and `ErrorWidget.builder`.

Internal Working



- `FlutterError.onError`: called for errors **within the Flutter framework** — exceptions in `build`/layout/paint, gesture handlers, etc. Default prints to console (red screen in debug). Override to **forward to your reporter** (call `FlutterError.presentError` in debug to keep the console output). This is the **framework** channel.
- `PlatformDispatcher.instance.onError`: the top-level hook for **uncaught async errors and platform-dispatched errors** (Flutter 3.3+). Return `true` to mark handled. Simpler than a zone for many cases; covers errors the framework hook doesn't.
- `runZonedGuarded`: run `runApp` inside a zone whose `onError` catches **any uncaught error in that zone** (including some async). Historically the way to catch everything; still used, but ensure `WidgetsFlutterBinding.ensureInitialized()` is inside the same zone and don't double-report (pick a consistent combo — commonly `PlatformDispatcher.onError` + `FlutterError.onError`, or the zone).
- **Wire to reporting**: all three should funnel to **logging + a crash reporter** (Crashlytics/Sentry — Module 52) with the error + stack, so production failures are visible.
- **Widget error boundary** (`ErrorWidget.builder`): override to render a **friendly fallback** instead of the default red error box; scope-limited boundaries (a wrapper widget catching build errors of a subtree) prevent one widget from crashing the whole screen. In release, the default is a grey box — customize it.
- **Isolates**: background isolates need their **own** error handling (`Isolate.addErrorListener` / handle in the entry) — the main zone doesn't catch them (02 · isolates).
- **Don't swallow**: global handlers **log/report and (optionally) show a friendly screen** — they're a net, not an excuse to ignore local handling.

Memory Representation

Handlers are callbacks holding references to the reporter/logger. Errors arrive as `(error, stackTrace)`. No persistent state beyond what you log/report.

Compiler Behavior

Not applicable.

Runtime Behavior

Uncaught framework errors → `FlutterError.onError`; uncaught async/platform → `PlatformDispatcher.onError` (or the zone); a build throw → red/grey box or your `ErrorWidget.builder`. In debug, framework errors also show the red screen (keep `presentError`).

Flutter Engine Behavior

The engine/binding routes framework errors to `FlutterError.onError` and platform/async errors to `PlatformDispatcher.onError`; `ErrorWidget` is substituted into the tree where a build fails.

Dart VM Behavior

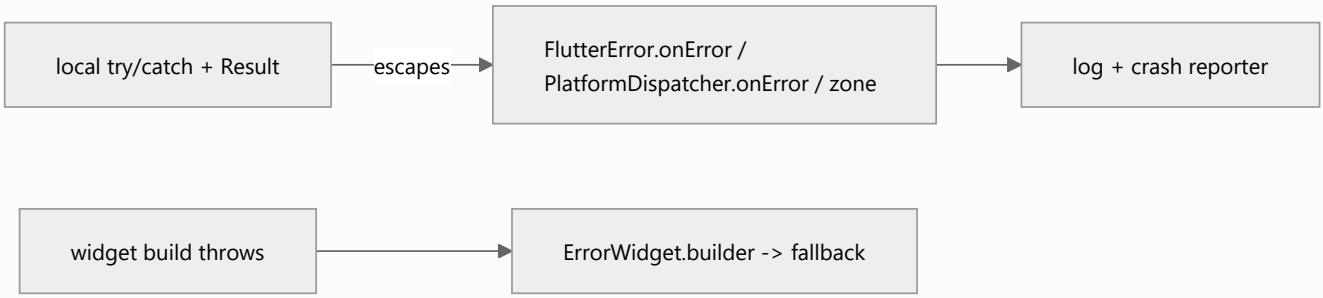
Zones (`runZonedGuarded`) intercept uncaught errors in their execution context; each isolate has independent error handling.

Examples

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

void main() {
  // Framework errors (build/layout/paint/gestures) -> reporter (+ console in debug)
  FlutterError.onError = (FlutterErrorDetails details) {
    FlutterError.presentError(details);            // keep debug red screen
    crashReporter.record(details.exception, details.stack); // Module 52
  };

  // Uncaught async / platform errors -> reporter
  PlatformDispatcher.instance.onError = (error, stack) {
    crashReporter.record(error, stack);
    return true;                                    // handled
  };

  // Friendly fallback instead of the default error box
  ErrorWidget.builder = (FlutterErrorDetails details) => const _FallbackTile();

  runApp(const MyApp());
}

// Alternative/complement: catch everything in a guarded zone
// runZonedGuarded(() {
//   WidgetsFlutterBinding.ensureInitialized();     // inside the same zone
//   runApp(const MyApp());
// }, (error, stack) => crashReporter.record(error, stack));
```

```
// Scoped widget error boundary: contain a subtree's build error to a fallback
class ErrorBoundary extends StatelessWidget {
  final Widget child; final Widget fallback;
  const ErrorBoundary({super.key, required this.child, required this.fallback});
  @override
  Widget build(BuildContext context) {
    ErrorWidget.builder = (_) => fallback;          // (illustrative; real impls wrap build)
    return child;
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
No global handler	Errors crash or vanish silently	Wire `FlutterError.onError` + `PlatformDispatcher.onError`
Not reporting to a crash tool	Prod failures invisible	Funnel handlers to Crashlytics/Sentry
`ensureInitialized` outside the zone	Zone doesn't catch startup errors	Put it inside `runZonedGuarded`
Double-reporting (zone + hooks)	Duplicate reports	Pick a consistent combo
Default red/grey error box in prod	Ugly UX	Custom `ErrorWidget.builder`
Ignoring isolate errors	Uncaught in background	Handle per-isolate
Treating global as a substitute for local	Loses context/recovery	Global is a net, not primary handling

Best Practices

- Install `FlutterError.onError` (framework) + `PlatformDispatcher.onError` (async/platform), and/or `runZonedGuarded` — a **consistent** combo (avoid double-reporting), with `ensureInitialized` inside the zone if used.
- **Funnel all** uncaught errors to **logging + a crash reporter** (Module 52) with stack traces; keep `presentError` for the debug red screen.
- Provide a **custom** `ErrorWidget.builder` / scoped **error boundaries** so one widget's build error shows a fallback, not a crashed screen.
- Handle **isolate** errors separately; treat global handlers as a **safety net**, not a replacement for local handling/ `Result`.

Performance

Negligible — handlers only fire on errors. Reporting is async/batched by the crash tool. Error boundaries add a trivial wrapper. The value is visibility + containment, not runtime cost.

Advantages / Disadvantages

- **+** No silent/uncaught crashes, production visibility (crash reports), contained widget failures, better UX on error.
- **−** Setup nuance (which hook, zone vs dispatcher, double-reporting), isolates need separate handling, not a substitute for local handling.

Interview Questions

1. 🟢 **What does** `FlutterError.onError` **catch?** — Errors inside the Flutter framework (build/layout/paint, gesture callbacks) — the framework channel.
2. 🟢 **What's** `PlatformDispatcher.instance.onError` **for?** — A top-level hook for uncaught async and platform-dispatched errors (return `true` when handled).
3. 🟡 **What does** `runZonedGuarded` **add?** — It runs `runApp` in a zone that catches uncaught errors (incl. some async) in that zone — ensure `ensureInitialized` is inside it.
4. 🟡 **How do you avoid double-reporting?** — Choose a consistent combination of hooks/zone rather than reporting the same error from multiple handlers.
5. 🟡 **What is** `ErrorWidget.builder` **for?** — Customizing the fallback UI shown when a widget's `build` throws (instead of the default red/grey box).
6. 🔴 **Why must background isolates handle their own errors?** — The main zone/handlers don't catch them; use `Isolate.addErrorListener`/handle in the entry.
7. 🔴 **Are global handlers a replacement for local handling?** — No — they're a safety net for what escapes; local handling/ `Result` keeps context and enables recovery.

Senior Engineer Tips

- Wire all global hooks to your crash reporter on day one; the worst production state is failures you never hear about because nothing was capturing them.
- Pick one consistent capture strategy (commonly `FlutterError.onError` + `PlatformDispatcher.onError`) and keep `presentError` in debug — juggling a zone on top often causes double reports or missed startup errors.
- Add a branded `ErrorWidget.builder` and scoped boundaries around risky subtrees; a contained "something went wrong, retry" beats a dead screen every time.

Architect Perspective

Global handling is the observability + resilience backstop of the error strategy: it guarantees no uncaught error is silent (reported to monitoring) and, via error boundaries, that failures are contained. It sits beneath local `try/catch` and `Result` (which handle expected failures with context) and feeds monitoring/logging — completing the layered defense so the app degrades gracefully and you *learn* about every failure (02_result_and_either.md, Module 52, Module 39).

Summary

- Install `FlutterError.onError` (framework) + `PlatformDispatcher.onError` (async/platform), and/or `runZonedGuarded`, in a consistent combo; funnel to logging + crash reporter.
- Customize `ErrorWidget.builder` / use scoped error boundaries so a widget error shows a fallback, not a crash.
- Handle isolate errors separately; global handlers are a net, not a substitute for local handling/ `Result`.

Revision Notes

- `FlutterError.onError` = framework errors (build/layout/paint/gestures; keep `presentError` in debug); `PlatformDispatcher.instance.onError` = uncaught async/platform (return true).
- `runZonedGuarded(() { ensureInitialized(); runApp(); }, onError)` — inside-zone init; avoid double-reporting; funnel all → crash reporter (Module 52).
- `ErrorWidget.builder` / scoped boundary = fallback UI (not red/grey box); isolates handle own errors; global = safety net, not primary handling.

Practice Questions

1. Which hook catches a build-method exception vs an unawaited future error?
2. How do you avoid duplicate crash reports?

3. How do you keep one broken widget from crashing the screen?

Coding Questions

1. Wire `FlutterError.onError` + `PlatformDispatcher.onError` to a crash reporter.
2. Add a custom `ErrorWidget.builder` fallback.
3. Set up `runZonedGuarded` with in-zone initialization (no double-report).

Mini Project

Global safety net (Flutter): Wire `FlutterError.onError` + `PlatformDispatcher.onError` (and/or `runZonedGuarded`) to a (stub) crash reporter, keep the debug red screen, add a branded `ErrorWidget.builder`, and wrap a risky subtree in a scoped error boundary. Verify a build exception, an unawaited failing future, and a deliberate widget error are all captured/contained. Acceptance: all three uncaught cases captured + reported (no silent loss/crash); debug red screen preserved; friendly fallback for widget errors; consistent combo (no double-report); isolate handling noted.

Recovery & User-Facing Errors

> Handling a failure isn't done until the app **recovers or communicates** gracefully: **retry with backoff** for transient failures (network blips, 503) — but only for **idempotent/safe** operations and with a cap; **degrade gracefully** (cached data, reduced features, offline mode) rather than blocking; and **map technical failures to clear, actionable user messages** ("You're offline — retry" not "SocketException"). The golden rules: **never leak raw exceptions/stack traces to users, never leave an infinite spinner, always offer a next step** (retry/back/support).

Introduction

This file is the UX + resilience end of error handling: turning caught failures (01_errors_vs_exceptions.md, 02_result_and_either.md) into recovery (retry/degrade) and clear messages. It covers retry/backoff, graceful degradation, error→message mapping, and error-state UI patterns.

Why this concept exists

A failure the code "handles" but the user experiences as a frozen screen or cryptic text is still a bad experience. Transient failures often succeed on retry; missing data can fall back to a cache; and users need to understand what happened and what to do. Good recovery/UX turns inevitable failures into minor bumps instead of dead ends.

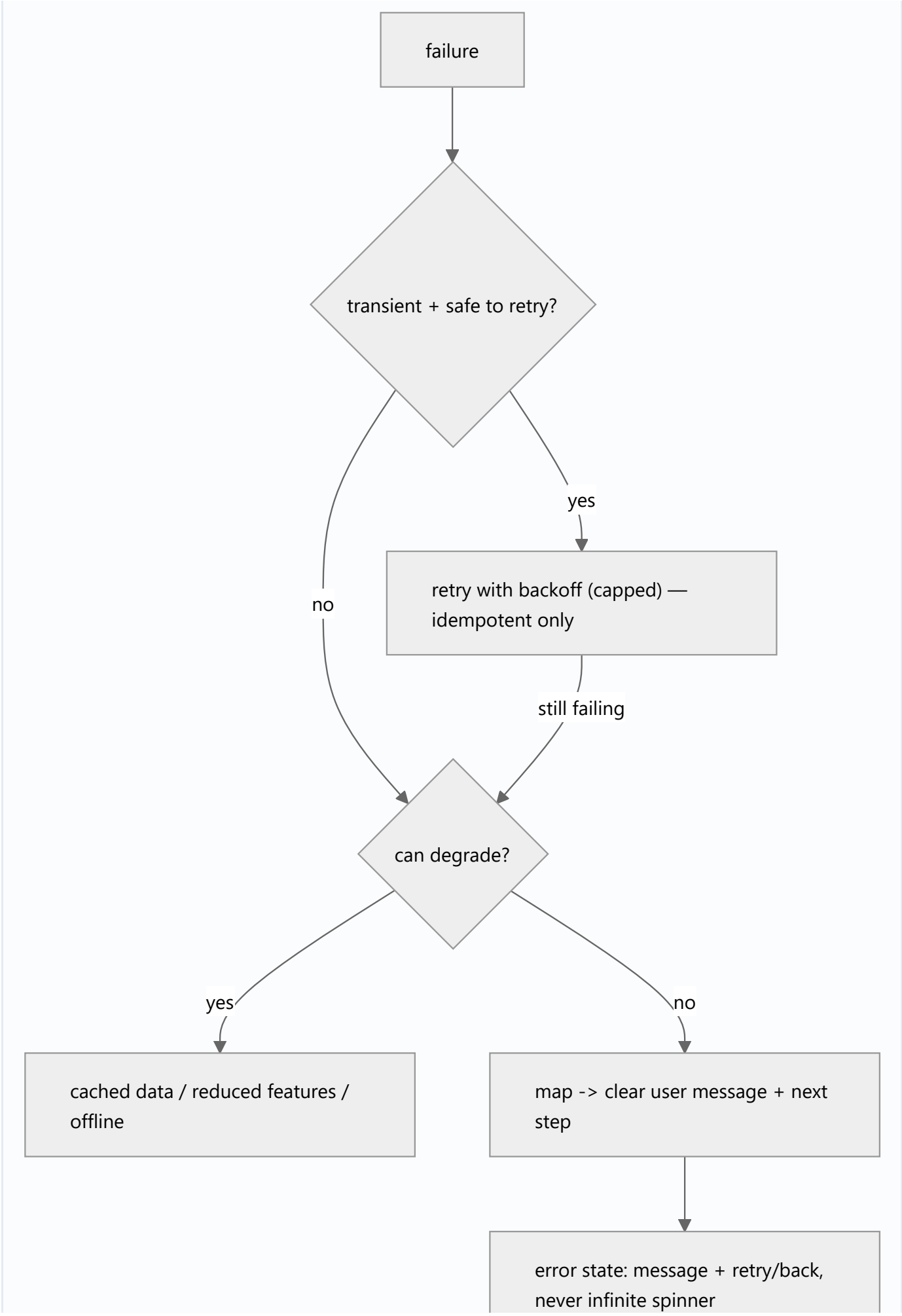
Real-world analogy

A good waiter (the app) doesn't tell you "kitchen exception 500" — they say "the special's sold out, may I suggest the pasta?" (clear message + next step). If the kitchen is briefly slammed they **quietly retry** getting your order in (backoff retry); if an ingredient's out they **substitute** (degrade gracefully). They never just **vanish and leave you staring at an empty table** (infinite spinner).

Problem Statement

For a data screen: retry a transient fetch failure automatically (capped, backoff) but not a duplicate-charge POST, fall back to cached data when offline, show "You're offline — Retry" (not `SocketException`) with a working retry button, and never leave a spinner forever. You'll implement retry/backoff, degradation, and error→message mapping + UI.

Internal Working


failure
transient + safe to retry?
yes
retry with backoff (capped) — idempotent only
still failing
no
can degrade?
yes
cached data / reduced features / offline
no
map -> clear user message + next step
error state: message + retry/back, never infinite spinner

- **Retry with backoff**: for **transient** failures (timeouts, 502/503, connection reset), retry a **capped** number of times with **exponential backoff + jitter** (avoid thundering herd). **Only retry idempotent/safe operations** (GET, or POSTs made idempotent via a key — Module 31); **never blindly retry** non-idempotent writes (double-charge/double-post). `dio` retry interceptors or a small helper.
- **Graceful degradation**: when the operation can't succeed, **fall back** — serve **cached/stale data** (with a "stale" indicator), disable just the broken feature (not the app), enter **offline mode** (Module 19). Failure of one thing shouldn't break everything.
- **Error → message mapping**: translate **typed failures** (02_result_and_either.md) into **user-friendly, actionable** copy: offline → "You're offline. Check your connection and retry."; not-found → "This item is no longer available."; server → "Something went wrong on our side. Try again shortly." **Never** show `Exception`/stack traces/error codes as the primary message (a support code is fine, secondary).
- **Error-state UI**: every async view has **loading / data / empty / error** states; the **error state** shows the message + a **retry/back** action. **No infinite spinners** (time out → error), no dead ends. Use snackbars for transient/inline errors, full-screen for blocking ones, banners for degraded/offline.
- **Recovery hooks**: retry re-runs the fetch; on auth failure → refresh token / re-login; on validation → highlight the field. Preserve user input across retries.
- **Report while recovering**: still **log/report** the underlying failure (Module 52) even when you show a friendly message — the user's experience and your telemetry are separate concerns.

Memory Representation

Retry state (attempt count, next delay) is transient. Cached fallback data lives in your cache (Module 34/Module 19). Error UI is driven by the state's error variant.

Compiler Behavior

Not applicable.

Runtime Behavior

Backoff spaces retries (growing delay + jitter); after the cap it surfaces an error state. Degradation swaps in cached data. The UI reflects loading→error/data; a timeout converts a hung request into a handled error.

Flutter Engine Behavior

Not applicable.

Dart VM Behavior

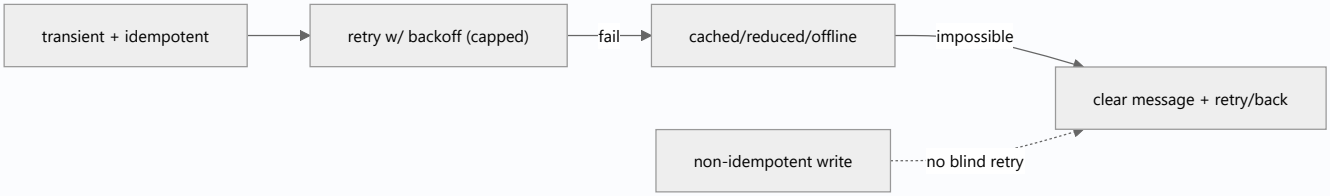
Not applicable.

Examples

```
// Retry with exponential backoff + jitter — IDEMPOTENT operations only, capped
Future<T> retry<T>(Future<T> Function() op, {int maxAttempts = 3}) async {
  var attempt = 0;
  while (true) {
    try {
      return await op();
    } on TransientException {
      attempt++;
      if (attempt >= maxAttempts) rethrow;            // give up -> error state
      final delay = Duration(milliseconds: (200 * (1 << attempt)) + _jitter());
      await Future.delayed(delay);                    // backoff
    }
    // Non-transient / non-idempotent failures: do NOT retry here.
  }
}

// Map typed failures -> user-friendly, actionable messages (never raw exceptions)
String messageFor(AppFailure f) => switch (f) {
  NetworkFailure() => "You're offline. Check your connection and retry.",
  NotFoundFailure() => "This item is no longer available.",
  ServerFailure() => "Something went wrong on our side. Please try again shortly.",
  _ => "Something went wrong. Please try again.",
};
```

```
// Error-state UI: message + retry, graceful degradation, NO infinite spinner
switch (state) {
  Loading() => const CenteredSpinner(),
  Data(:final items) => ItemList(items, stale: state.fromCache), // degraded/stale banner
  Empty() => const EmptyState('Nothing here yet'),
  Error(:final failure) => ErrorView(
      message: messageFor(failure),
      onRetry: () => bloc.add(Retry()),            // actionable next step
    ),
};
```

Diagrams



Common Mistakes

Mistake	Why it's bad	Fix
Retrying non-idempotent writes	Double-charge/double-post	Retry only idempotent/keyed ops
Retry without cap/backoff	Hammering, battery, herd	Capped exponential backoff + jitter
Showing raw exceptions/stacks to users	Confusing/leaky	Map to friendly, actionable copy
Infinite spinner on failure	User stuck	Timeout → error state with retry
No degradation	One failure breaks everything	Cached fallback / offline / reduced features
Dead-end error (no action)	User can't proceed	Always offer retry/back/support
Not reporting while recovering	Blind telemetry	Log/report + show friendly message

Best Practices

- **Retry transient, idempotent** failures with **capped exponential backoff + jitter**; **never blindly retry** non-idempotent writes (use idempotency keys).
- **Degrade gracefully** (cached/stale data with an indicator, offline mode, disable only the broken feature) instead of blocking the whole app.

- **Map typed failures to clear, actionable messages** (never raw exceptions/stacks); show a **support code** secondarily if needed.
- Give every async view **loading/data/empty/error** states with a **retry/next-step** action — **no infinite spinners**; **report** the underlying failure while showing friendly UX.

Performance

Backoff prevents retry storms (server + battery friendly); jitter avoids synchronized retries. Serving cache is faster than failing/blocking. Timeouts free hung requests. Good recovery improves both perceived performance and reliability.

Advantages / Disadvantages

- **+** Transient failures auto-recover, app stays usable (degradation), users understand + can act, better retention, telemetry preserved.
- **−** Must classify transient/idempotent correctly, backoff/state-machine complexity, message-mapping upkeep, cache/degradation infra.

Interview Questions

1. 🟢 **When is it safe to retry, and how?** — For transient failures on idempotent/safe operations, with a capped exponential backoff + jitter — never blindly for non-idempotent writes.
2. 🟢 **Why never show raw exceptions to users?** — They're confusing and can leak internals; map failures to clear, actionable messages (with an optional support code).
3. 🟡 **What is graceful degradation?** — Falling back (cached/stale data, offline mode, disabling just the broken feature) so a failure doesn't block the whole app.
4. 🟡 **Why is an infinite spinner a bug?** — It leaves users stuck; add timeouts and always resolve to a data/empty/error state with a next step.
5. 🟡 **How do you retry safely for a payment/POST?** — Make it idempotent (idempotency key) so a retry can't double-charge; otherwise don't auto-retry.
6. 🔴 **What UI states must an async view handle?** — Loading, data, empty, and error (with retry) — never just loading→data.
7. 🔴 **Should you still report errors you recover from?** — Yes — user-facing recovery and telemetry are separate; log/report the underlying failure regardless.

Senior Engineer Tips

- Classify failures as transient vs permanent and idempotent vs not before adding retries; a retry on a non-idempotent write is a money/data bug, not a resilience feature.

- Kill infinite spinners with timeouts and a mandatory error state + retry; "stuck loading" is the single most common user-reported failure.
- Keep a central error→message map so UX copy is consistent and no `toString()` of an exception ever reaches a user.

Architect Perspective

Recovery/UX is where the error strategy pays off for users: typed failures (02_result_and_either.md) become retry/degradation decisions and clear messages, driven by explicit UI states, while the underlying failures still flow to monitoring. Centralizing retry policy, degradation, and error→message mapping (behind services/state) makes resilience consistent and testable — the user-facing complement to global capture (03_global_error_handling.md, Module 19, Module 52).

Summary

- Retry transient, idempotent failures with capped backoff + jitter; never blindly retry non-idempotent writes.
- Degrade gracefully (cache/offline/reduced) and map typed failures to clear, actionable messages — never raw exceptions or infinite spinners.
- Every async view has loading/data/empty/error states with a next step; report underlying failures while showing friendly UX.

Revision Notes

- Retry: transient + idempotent only, capped exponential backoff + jitter; non-idempotent → idempotency key or no auto-retry.
- Degrade: cached/stale (indicator), offline mode, disable only broken feature; error→message map (friendly + actionable, optional support code, never raw exception/stack).
- UI states: loading/data/empty/error + retry; no infinite spinner (timeouts); report underlying failure to monitoring while showing friendly UX.

Practice Questions

1. Which failures should you retry, and how do you avoid a double-charge?
2. What does graceful degradation look like for an offline data screen?
3. Why is an infinite spinner considered an error-handling bug?

Coding Questions

1. Implement capped exponential-backoff retry for idempotent ops only.
2. Build an error→message mapper over a sealed `Failure` hierarchy.
3. Render loading/data/empty/error states with a working retry.

Mini Project

Resilient data screen (Flutter): For a fetch screen, implement capped exponential-backoff retry (idempotent GET only), graceful degradation to cached data (with a stale banner) when offline, a typed-failure→message mapper, and full loading/data/empty/error UI with a retry action — no infinite spinner, and report underlying failures to a (stub) monitor. Acceptance: transient failures auto-retry (backoff+jitter, capped); non-idempotent ops not blindly retried; offline degrades to cache; friendly actionable messages (no raw exceptions); all UI states + retry; no infinite spinner; failures reported.

Error Integration (Capstone: A Layered Error Strategy)

> The maintainable shape: error handling is a **layered strategy**, not scattered `try/catch` — the **data layer** converts exceptions to typed `Result` / `Failure` at the boundary, the **domain** maps/propagates them, the **presentation** (bloc/state) turns failures into explicit **error states**, the **UI** renders them with recovery (retry/degrade/message), and a **global safety net** (`FlutterError.onError` + `PlatformDispatcher.onError` /zone) captures anything uncaught and reports it to monitoring. Each layer has one job; together they make failure explicit, contained, recoverable, observable, and testable.

Introduction

This module capstone composes the errors/exceptions distinction, `Result` / `Either`, global handling, and recovery/UX into one coherent strategy mapped to your architecture. Ad hoc error handling drifts into inconsistency and silent failures; a layered strategy makes failure a designed, first-class concern. This file shows the per-layer responsibilities and an end-to-end failure flow.

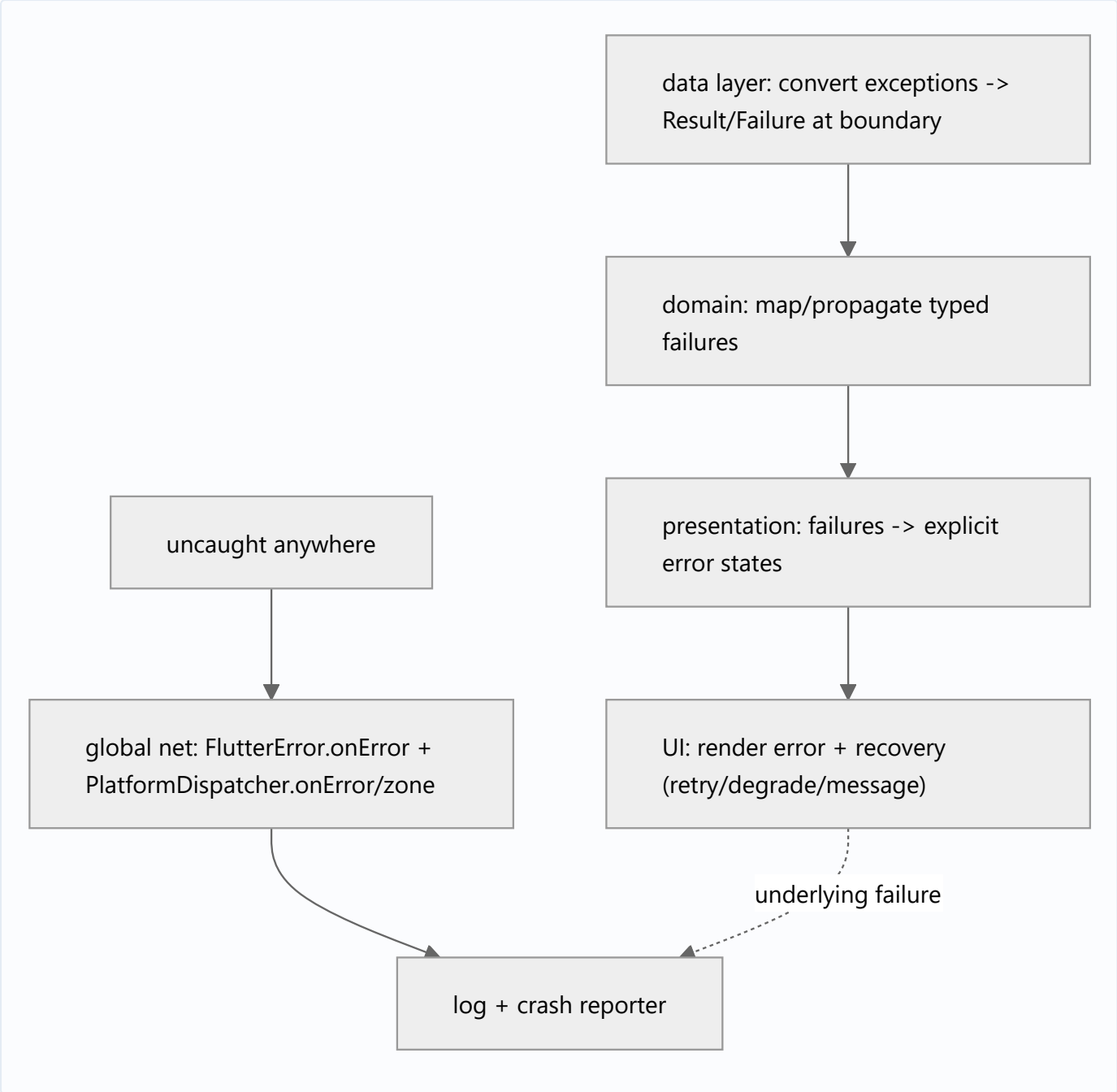
Why this concept exists

Error handling touches every layer (I/O, domain, state, UI, observability) and every layer has a *different* right answer (convert, map, model, render, capture). Without a strategy, teams mix concerns (try/catch in UI, exceptions leaking to users, silent swallows, unreported crashes). A layered assignment of responsibilities — like clean architecture for correctness — makes failure handling consistent, testable, and observable (Module 40).

Real-world analogy

A resilient organization handles problems in layers: the **front line** converts raw incidents into standard reports (data→ `Result`), **operations** routes and interprets them (domain), the **service desk** decides the customer response (state), the **customer-facing staff** communicate clearly with a next step (UI), and a **central incident log** records everything even if it slipped past the process (global capture + monitoring). Each role does its part; nothing is silently dropped, and the customer always gets a clear answer.

Internal Working



- **Data layer** (02_result_and_either.md): the *only* place with `try/catch` around I/O; converts exceptions → **typed** `Failure` and returns `Result<T>`. Inner layers stay throw-free for expected failures.
- **Domain**: composes/maps `Result`s (network `Failure` → domain `Failure`), applies business-rule failures; no UI concerns.
- **Presentation (bloc/state)** (Module 11): consumes `Result`, emits **explicit states** (loading/data/empty/error(failure)); decides **retry/degradation** policy (04_recovery_and_user_errors.md).
- **UI**: renders each state; error state = **friendly mapped message + retry/next step**; **no infinite spinner**, no raw exceptions; scoped **error boundary** contains widget build errors

(03_global_error_handling.md).

- **Global safety net**: `FlutterError.onError` + `PlatformDispatcher.onError` (and/or `runZonedGuarded`) capture **anything uncaught → logging + crash reporter** (Module 52). Bugs (`Error`s) crash in dev, are captured in prod.
- **Observability everywhere**: even recovered failures are **logged/reported** (Module 39) — user UX and telemetry are separate.
- **Classification discipline** (01_errors_vs_exceptions.md): expected → `Result` + recover; bugs → crash/capture; never silent swallow.
- **Testability**: because failures are typed values flowing through layers, you **unit-test failure paths** (repo returns `Failure` → state → error UI) without real I/O.

Memory Representation

Failures are typed values (`Result` / `Failure`) flowing up; error states hold the failure; global handlers hold the reporter. No hidden control flow — the failure is data.

Compiler Behavior

Sealed `Result` / `Failure` + `switch` give **exhaustive** handling (can't forget a case). Bugs still throw as `Error`s.

Runtime Behavior

Expected failures flow as values → error states → recovery UI; uncaught errors hit the global net → reported. Retries/degradation happen at the presentation layer; nothing crashes silently.

Flutter Engine Behavior

Framework/async errors route to the global hooks; widget build errors to the error boundary — as per 03_global_error_handling.md.

Dart VM Behavior

`Result` avoids unwinding for expected failures; isolates handle their own errors + report.

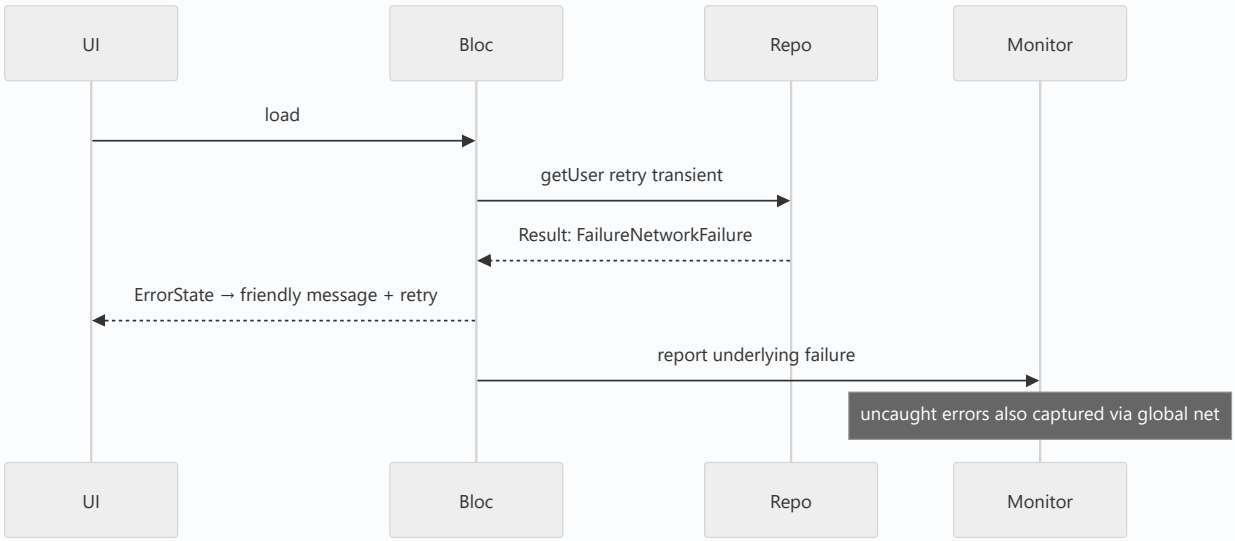
Examples

```
// DATA: convert exceptions -> typed Result at the boundary
class UserRepository {
  Future<Result<User>> getUser(String id) async {
```

```
    try {
      final res = await api.get('/users/$id');
      if (res.statusCode == 404) return Failure(NotFoundFailure('user'));
      return Success(User.fromJson(res.data));
    } on SocketException { return Failure(NetworkFailure()); }
  }
}

// PRESENTATION: Result -> explicit states, with retry policy
class UserBloc {
  Future<void> load(String id) async {
    emit(Loading());
    final r = await retryIdempotent(() => repo.getUser(id)); // backoff for transient
    emit(switch (r) {
      Success(:final value) => Data(value),
      Failure(:final error) => ErrorState(error),           // typed failure
    });
  }
}

// UI: friendly message + retry; no raw exceptions / infinite spinner
Widget build(BuildContext c) => switch (state) {
  Loading() => const CenteredSpinner(),
  Data(:final user) => Profile(user),
  ErrorState(:final failure) => ErrorView(message: messageFor(failure), onRetry: () => bloc.load(id)),
};

// GLOBAL: capture anything uncaught -> monitoring
void main() {
  FlutterError.onError = (d) { FlutterError.presentError(d); crash.record(d.exception, d.stack); };
  PlatformDispatcher.instance.onError = (e, s) { crash.record(e, s); return true; };
  runApp(const MyApp());
}
```

```
// Testable failure path — no real I/O
test('offline -> error state -> friendly message', () async {
  final bloc = UserBloc(FakeRepo(returns: Failure(NetworkFailure())));
  await bloc.load('1');
  expect(bloc.state, isA<ErrorState>());
  expect(messageFor((bloc.state as ErrorState).failure), contains('offline'));
});
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`try/catch` scattered across layers	Mixed concerns, inconsistent	Convert at data boundary; `Result` inward
Exceptions leaking to UI/users	Crashes / cryptic text	Typed failures → mapped messages
No global net	Uncaught errors silent/crash	`FlutterError` + `PlatformDispatcher` → monitor
Recovered failures not reported	Blind telemetry	Log/report even on graceful recovery
Swallowing bugs as "handled"	Hides defects	Bugs crash (dev) / captured (prod)
Untested failure paths	Regressions	Unit-test repo→state→UI failure flow
Inconsistent error strategy	Drift, confusion	One documented layered convention

Best Practices

- Assign **one job per layer**: data **converts** exceptions → typed `Result` / `Failure`; domain **maps**; presentation **models** error states + retry/degradation; UI **renders** friendly messages + recovery; global net **captures** the rest.
- Keep `try/catch` **at the data boundary** (inner layers throw-free for expected failures); **bugs crash/capture**, expected failures flow as **typed values**.
- **Report every failure** (even recovered) to logging/monitoring; **never** leak raw exceptions to users or leave infinite spinners.
- Make failure **testable** (typed values → unit-test each layer's failure path); document the **one** strategy and apply it consistently.

Performance

The strategy adds no runtime cost (typed values are cheap; global handlers fire only on errors) and improves resilience/perceived performance (retry, degradation, no hangs). The investment is design consistency; the payoff is fewer crashes, better UX, and visible telemetry.

Advantages / Disadvantages

- **+** Consistent, layered, testable failure handling; explicit + recoverable; nothing silent (observable); clean happy path; friendly UX.
- **−** Upfront convention/boilerplate (`Result`, states, mapping), discipline to keep `try/catch` at the boundary, cross-layer coordination.

Interview Questions

1. 🟢 **What's each layer's error-handling job?** — Data converts exceptions→`Result`; domain maps; presentation models error states + retry; UI renders friendly messages + recovery; global net captures the rest.
2. 🟢 **Where should `try/catch` live?** — At the data boundary (I/O); inner layers stay throw-free for expected failures, which flow as typed `Result`s.
3. 🟡 **How do you ensure no failure is silent?** — A global net (`FlutterError.onError` + `PlatformDispatcher.onError`/zone) reports uncaught errors, and even recovered failures are logged to monitoring.
4. 🟡 **How do bugs vs expected failures flow differently?** — Expected failures become typed `Result`s handled/recovered; bugs (`Error`s) crash in dev and are captured in prod — not swallowed.
5. 🟡 **Why does this make failure testable?** — Failures are typed values flowing through layers, so you can unit-test repo→state→UI failure paths without real I/O.

6. 🔴 **How do retry/degradation fit the layers?** — At the presentation layer, deciding policy over the `Result` (retry transient/idempotent, degrade to cache) before emitting the error state.
7. 🔴 **Why report failures you've recovered from?** — User UX and telemetry are separate concerns; you still need visibility into failure rates/causes.

Senior Engineer Tips

- Write the error strategy down (one page: which layer does what) and enforce "`try/catch` only at the data boundary" in review — it's the single rule that keeps error handling from sprawling.
- Wire the global net + monitoring first so you have visibility from day one; then layer typed failures + recovery UX on top.
- Test the unhappy paths explicitly (offline, 404, 500, timeout) as first-class cases — they're where real apps break and where most bugs hide.

Architect Perspective

Error integration is the correctness-and-resilience counterpart to clean architecture: failure is a designed, typed concern with a clear per-layer responsibility, contained by boundaries and a global net, made visible by monitoring, and recoverable in the UI. This turns "the app crashes / spins / shows cryptic errors" into "failures are explicit, handled at the right layer, observed, and communicated" — the same layered discipline the whole handbook applies to structure, now applied to failure (Module 40, Module 11, Module 52).

Summary

- Layer the strategy: data converts exceptions→typed `Result`; domain maps; presentation models error states + retry/degradation; UI renders friendly recovery; global net captures the rest → monitoring.
- `try/catch` at the boundary; bugs crash/capture, expected failures flow as typed values; report even recovered failures; never leak exceptions or hang.
- Failure becomes explicit, contained, recoverable, observable, and testable; document + apply one convention.

Revision Notes

- Layers: data → `Result`/`Failure` (only `try/catch` here); domain → map; presentation → error states + retry/degrade; UI → friendly message + recovery; global → `FlutterError` + `PlatformDispatcher`/zone → monitor.

- Bugs crash (dev)/capture (prod); expected failures = typed values; report even recovered; no raw exceptions/infinite spinners.
- Testable failure paths (repo→state→UI); one documented convention; observability everywhere.

Practice Questions

1. What is each layer responsible for in the error strategy?
2. Why keep `try/catch` only at the data boundary?
3. How do you guarantee no failure is silent?

Coding Questions

1. Wire a full path: repo `Result` → bloc error state → UI message + retry.
2. Add the global net reporting to monitoring.
3. Unit-test the offline/404/500 failure paths across layers.

Mini Project

Layered error strategy (capstone — Flutter): For a feature, implement the full strategy: data layer returns typed `Result` / `Failure` (converting exceptions at the boundary), presentation maps to loading/data/empty/error states with retry/degradation, UI renders friendly messages (no raw exceptions/infinite spinner), a global net (`FlutterError.onError` + `PlatformDispatcher.onError`) reports uncaught errors + a scoped error boundary contains widget crashes, and all failures (even recovered) go to a (stub) monitor. Add unit tests for offline/404/500 paths. Acceptance: `try/catch` only at the data boundary; typed failures flow to explicit states + friendly UI; retry/degradation applied; uncaught errors captured + reported; widget errors contained; failure paths unit-tested; one consistent documented strategy.