39 · Logging

Flutter Practice Notes

Contents

39 · Logging

Introduction

This module covers logging done right in Flutter: **fundamentals** (why logging matters, log levels, structured vs string logs, `print` vs `dart:developer` vs a logging package), **the `logger` package & setup** (formatting, filters, dev-vs-prod configuration behind an abstraction), **production logging & redaction** (never log PII/secrets, sampling, performance, security), and **remote logging & observability** (remote sinks, crash-report integration, correlation ids, breadcrumbs/tracing) — tied together in a capstone. It complements error handling (Module 38) and monitoring (Module 52).

Why this module exists

You can't fix what you can't see. Logging is how you understand behavior in dev *and* — critically — in production where you can't attach a debugger. But naive logging causes real harm: `print` everywhere bloats and slows apps, logging tokens/PII is a **security and compliance breach**, and unstructured string logs are useless at scale. Good logging is deliberate: leveled, structured, redacted, environment-aware, and wired to observability — so you can debug production without leaking data or degrading performance.

Module map

#	File	Topic	Level
1	01_logging_fundamentals.md	Why log, log levels, structured logging, `print` / `log` /package	🔵
2	02_logger_package_and_setup.md	`logger` package, formatting, filters, dev-vs-prod behind an abstraction	🟡
3	03_production_logging_and_redaction.md	PII/secret redaction, what not to log, sampling, performance, security	🔴
4	04_remote_logging_and_observability.md	Remote sinks, crash-report integration, correlation ids, breadcrumbs/tracing	🔴
5	05_logging_integration.md	Capstone: a logging facade across environments	🔴

> **Cross-references:** Error handling (log on failure): Module 38. Monitoring/crash reporting: Module 52. Security (never log secrets): Module 37. Networking (request/response logging): Module 16. DI (inject the logger): Module 14. App size/perf (strip logs): Module 21.

Prerequisites

38 Error Handling, 37 Security (redaction), 14 Dependency Injection, basic app architecture.

What you'll be able to do after this module

- Use log levels and structured logging instead of scattered `print`s.
- Configure the `logger` package with dev-vs-prod behavior behind an abstraction.
- Redact PII/secrets, sample, and keep logging cheap and compliant in production.
- Ship logs/breadcrumbs to remote sinks and correlate them with crash reports and traces.
- Wire a single logging facade across environments that's testable and observable.

Capstone

Logging slice: A logging facade (interface) with a pretty console logger in dev and a redacting, sampled, remote-shipping logger in prod, integrated with the crash reporter (breadcrumbs + correlation ids), used across layers (with a `dio` request logger) — PII-safe, performant, and testable.

Summary

Logging = deliberate visibility: leveled + structured logs behind a facade, environment-aware (verbose dev, redacted/sampled/remote prod), never leaking PII/secrets, and correlated with crash reporting and traces. It's how you debug production safely — the observability complement to error handling.

Logging Fundamentals (Levels, Structure, `print` vs `log`)

> Logging is deliberate, **leveled**, **structured** visibility — not scattered `print`s. Use **severity levels** (trace/debug/info/warning/error/fatal) so you can filter noise from signal, and prefer **structured logs** (event + key/value fields) over freeform strings so logs are searchable/parseable at scale. In Flutter, **`print` / `debugPrint` are for throwaway debugging** (they ship to release and can be throttled/dropped); **`dart:developer`'s `log()`** is better (level, name, error/stack, DevTools integration); a **logging package** (02_logger_package_and_setup.md) is best for real apps.

Introduction

Before any package, you need the concepts: why log at all, what the levels mean, why structure beats strings, and the difference between `print`, `debugPrint`, and `dart:developer`'s `log`. This file establishes the vocabulary the rest of the module builds on.

Why this concept exists

In production you can't attach a debugger — logs are your only window into behavior. But logs are useless if everything is the same severity (can't filter), unstructured (can't search/aggregate), or done with `print` (ships to release, no metadata, throttled). Levels + structure + the right API turn logging from noise into a diagnostic tool.

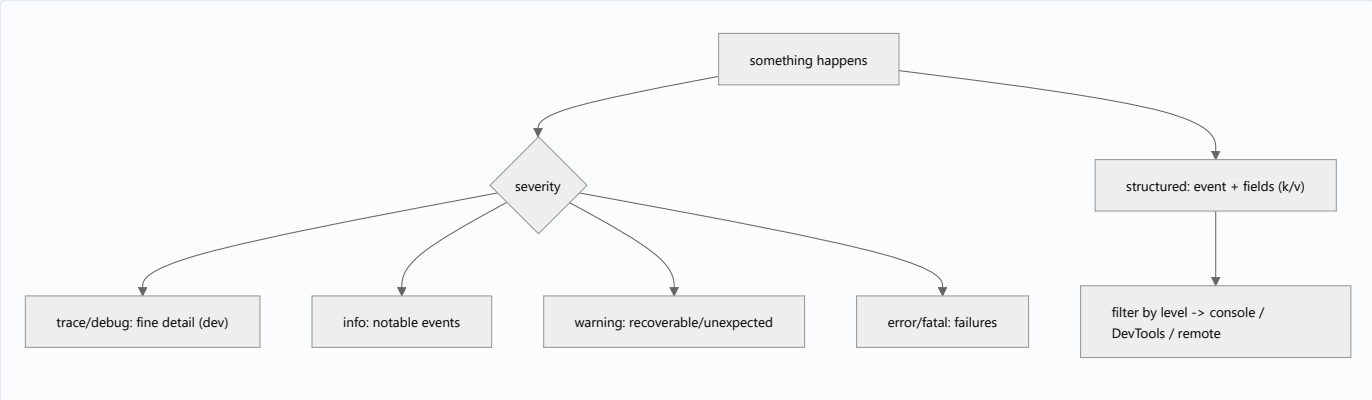
Real-world analogy

Logging is a **flight data recorder**: it must record the right events at the right **severity** (a routine altitude reading vs an engine-fire alarm) in a **structured, machine-readable** format so investigators can search and reconstruct what happened — not a pilot scribbling random notes on napkins (`print`) that get lost.

Problem Statement

Instrument a feature so you can trace normal flow, spot warnings, and capture errors — filterable by severity and searchable by fields (user action, screen, duration) — without using `print` or logging noise. You'll pick levels, structure events, and use `dart:developer`/a logger.

Internal Working



- **Log levels** (severity, low→high): **trace/verbose** (very fine detail), **debug** (dev diagnostics), **info** (notable events: login, screen view), **warning** (unexpected but recoverable), **error** (failures), **fatal** (crash-worthy). Levels let you set a **threshold** (e.g., info+ in prod, debug+ in dev) to separate signal from noise.
- **Structured logging**: log an **event + key/value fields** (`event: 'checkout', step: 'payment', amountCents: 1499, durationMs: 320`) rather than a freeform sentence. Structured logs are **searchable, filterable, aggregatable** by machines (remote sinks/observability — 04_remote_logging_and_observability.md); string logs aren't.
- **`print` / `debugPrint`**: `print` writes to stdout — **ships to release**, no level/metadata, and platform consoles may **throttle/drop** high-volume output; `debugPrint` throttles to avoid loss but is still for **temporary** debugging. **Don't build real logging on `print`.**
- **`dart:developer` `log()`**: `log(message, level:, name:, error:, stackTrace:, time:)` — has **levels, logger names, error/stack**, integrates with **DevTools/observatory**, and can be filtered. A solid built-in step up from `print`.
- **Logging package** (`logger`, etc.): pretty dev output, filters, multiple outputs, structure — the practical choice for apps (02_logger_package_and_setup.md).
- **What to log**: notable state transitions, decisions, external calls (metadata, not bodies), errors with context — **enough to reconstruct behavior**, not everything (noise + cost + PII risk — 03_production_logging_and_redaction.md).
- **Consistency**: consistent event names/fields/levels make logs analyzable; ad hoc strings don't.

Memory Representation

A log record is (level, message/event, fields, timestamp, optional error/stack, logger name). Structured records serialize to JSON for remote sinks; console logs are formatted strings.

Compiler Behavior

`print` / `debugPrint` calls remain in release unless stripped/guarded; wrap dev-only logs behind `kDebugMode` /build config or a no-op prod logger (02_logger_package_and_setup.md).

Runtime Behavior

Below-threshold logs are filtered cheaply; high-volume `print` can be throttled/dropped by the platform console. Structured logs are emitted to sinks (console/DevTools/remote).

Flutter Engine Behavior

`dart:developer` `log` integrates with DevTools' logging view (filter by level/name); `print` goes to the raw console.

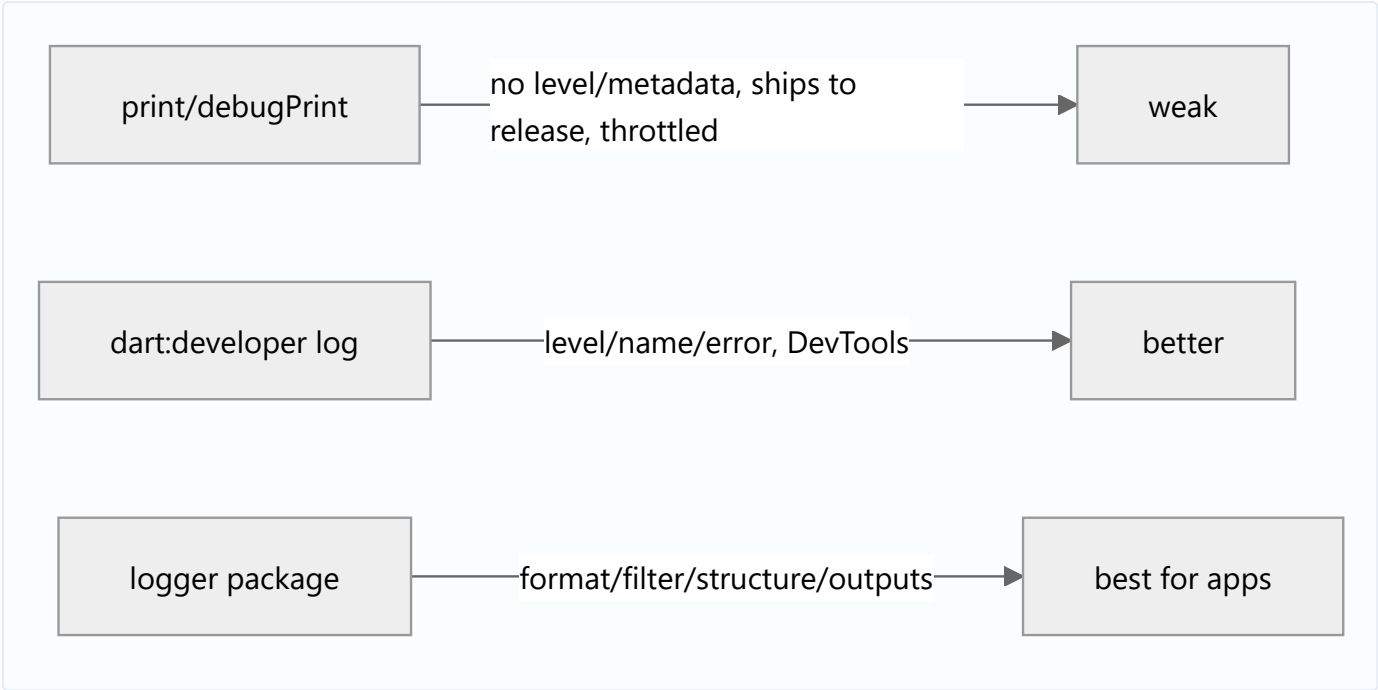
Dart VM Behavior

`log` records carry structured metadata the VM/DevTools understand; guarding by `kDebugMode` lets the compiler tree-shake dev-only logging.

Examples

```
import 'dart:developer' as dev;
import 'package:flutter/foundation.dart';

// ANTI-PATTERN: print() for real logging (ships to release, no level, may be dropped)
// print('user ${user.email} logged in');   // ❌ also logs PII!

// Better: dart:developer log with level, name, error/stack
void logLogin(String userId) {
  dev.log('login', name: 'auth', level: 800 /* INFO */); // structured-ish, filterable
}

// Structured event (searchable fields) — no PII
void logCheckoutStep(String step, int amountCents, int durationMs) {
  dev.log(
    'checkout',
    name: 'checkout',
    level: 800,
    // in a real logger, pass fields as a map; here shown inline
    error: null,
  );
  // logger.i('checkout', {'step': step, 'amountCents': amountCents, 'durationMs': durationMs});
}

// Error with context (level ERROR + stack)
void logFailure(Object e, StackTrace st) =>
    dev.log('fetch_failed', name: 'data', level: 1000, error: e, stackTrace: st);

// Dev-only diagnostic (tree-shaken in release)
if (kDebugMode) dev.log('cache miss for key', name: 'cache', level: 500 /* DEBUG */);
```

Diagrams



Common Mistakes

Mistake	Why it's bad	Fix
`print` everywhere for logging	Ships to release, no levels, throttled/dropped	Use `dart:developer`/a logger
No log levels	Can't filter signal from noise	Use severity levels + threshold
Freeform string logs	Not searchable/aggregatable	Structured event + fields
Logging everything	Noise, cost, PII risk	Log enough to reconstruct behavior
Logging PII/secrets	Security/compliance breach	Redact (production file)
Dev logs shipped to release	Bloat/perf/leak	Guard with `kDebugMode`/prod no-op
Inconsistent event names/levels	Unanalyzable	Consistent conventions

Best Practices

- Log with **severity levels** and a **threshold** (verbose in dev, info+ in prod) to separate signal from noise; use **consistent** event names/levels.
- Prefer **structured logs** (event + key/value fields) over freeform strings for searchability/aggregation.
- Don't build logging on `print` — use `dart:developer` `log` or a **logging package**; **guard dev-only logs** (`kDebugMode`/prod no-op).
- Log **enough to reconstruct behavior** (transitions, decisions, external calls' metadata, errors + context) — **never PII/secrets** (03_production_logging_and_redaction.md).

Performance

Below-threshold logs should be cheap (guard expensive message construction behind the level check / `kDebugMode`). `print` at volume is throttled/dropped (and slow). Structured logging adds minor serialization cost — worth it for searchability; sample high-volume logs in prod (03_production_logging_and_redaction.md).

Advantages / Disadvantages

- **+** Production visibility, filterable by severity, searchable (structured), reconstruct behavior without a debugger.
- **−** Overhead if overdone, PII/security risk, `print` misuse, requires discipline (levels/structure/consistency).

Interview Questions

1. 🟢 **Why not use `print` for real logging?** — It ships to release, has no levels/metadata, and platform consoles may throttle/drop it; use `dart:developer` `log` or a logging package.
2. 🟢 **What are log levels for?** — Severity (trace→fatal) lets you filter and set a threshold, separating signal from noise (verbose dev, info+ prod).
3. 🟡 **Why structured over string logs?** — Event + key/value fields are searchable, filterable, and aggregatable by machines (remote/observability); freeform strings aren't.
4. 🟡 **What does `dart:developer`'s `log` add over `print`?** — Levels, logger names, error/stack, timestamps, and DevTools integration.
5. 🟡 **What should you log (and not)?** — Enough to reconstruct behavior (transitions, decisions, external-call metadata, errors + context); never PII/secrets, never "everything."
6. 🔴 **How do you keep dev-only logs out of release?** — Guard with `kDebugMode`/build config (tree-shaken) or route through a prod no-op logger.
7. 🔴 **How do you keep below-threshold logging cheap?** — Guard expensive message/field construction behind the level check so it isn't computed when filtered out.

Senior Engineer Tips

- Ban `print` in review and standardize on a logger with levels + structured fields; consistent event names are what make logs analyzable later.
- Guard expensive log-message construction behind the level/`kDebugMode` check — building a big string only to drop it is a silent perf cost.
- Decide what's worth logging by "could I reconstruct this incident from the logs?" — not "log everything," which just creates noise, cost, and PII risk.

Architect Perspective

Logging fundamentals set the contract for observability: leveled, structured, consistent events that machines and humans can filter and search. Choosing the right API (`log` /package over `print`) and conventions up front — behind a facade (05_logging_integration.md) — makes production debugging possible without leaking data or degrading performance, and feeds directly into remote sinks and crash correlation (04_remote_logging_and_observability.md, Module 52).

Summary

- Log with severity levels + a threshold (verbose dev, info+ prod) and structured events (fields), consistently.
- Don't use `print` for logging — use `dart:developer` `log` or a package; guard dev-only logs.
- Log enough to reconstruct behavior; never PII/secrets; keep below-threshold logging cheap.

Revision Notes

- Levels: trace/debug/info/warning/error/fatal → filter by threshold (dev verbose, prod info+); structured event + k/v fields (searchable).
- `print` / `debugPrint` = temporary only (ships to release, throttled); `dart:developer` `log(msg, level, name, error, stackTrace)` = levels/DevTools; package = best.
- Log transitions/decisions/external-call metadata/errors+context; never PII/secrets; guard dev logs (`kDebugMode`); keep below-threshold cheap.

Practice Questions

1. Why are log levels essential for production?
2. What makes a log "structured," and why does it matter?
3. When is `print` acceptable, and when is it not?

Coding Questions

1. Replace `print` s with leveled `dart:developer` `log` calls (with name/level).
2. Emit a structured event with fields (no PII).
3. Guard a dev-only debug log with `kDebugMode` .

Mini Project

Leveled, structured logging (Flutter): Instrument a feature with `dart:developer` `log` (or a logger) using severity levels and structured events (event + fields), a dev/prod threshold, guarded dev-only logs, and no `print`/PII. Acceptance: levels used + filterable; events structured with fields; dev-only logs guarded (`kDebugMode`); no `print` ; no PII/secrets; enough logged to reconstruct the feature's flow.

The `logger` Package & Setup

> The `logger` package gives you leveled logging with pluggable **filters** (which levels emit), **printers** (formatting — pretty for dev, compact/JSON for prod), and **outputs** (console, file, remote) — but the key architectural move is to **hide it behind your own logging abstraction** so the rest of the app depends on *your* interface, not the package, and behaves differently per environment (**pretty + verbose in dev, filtered + structured + no-op-or-remote in prod**). Never scatter `Logger()` instances everywhere; inject one facade.

Introduction

This file covers configuring `logger` (filter/printer/output), the dev-vs-prod split, and — most importantly — wrapping it behind an abstraction injected via DI so logging is centralized, swappable, and environment-aware. It's the practical setup layer over the fundamentals (01_logging_fundamentals.md).

Why this concept exists

Apps need readable logs while developing and lean, structured, filtered logs in production — and they shouldn't be coupled to a specific library (you may swap it or add remote shipping). A configured `logger` behind a facade delivers both: rich dev output, controlled prod output, one injection point, and easy testing/swapping.

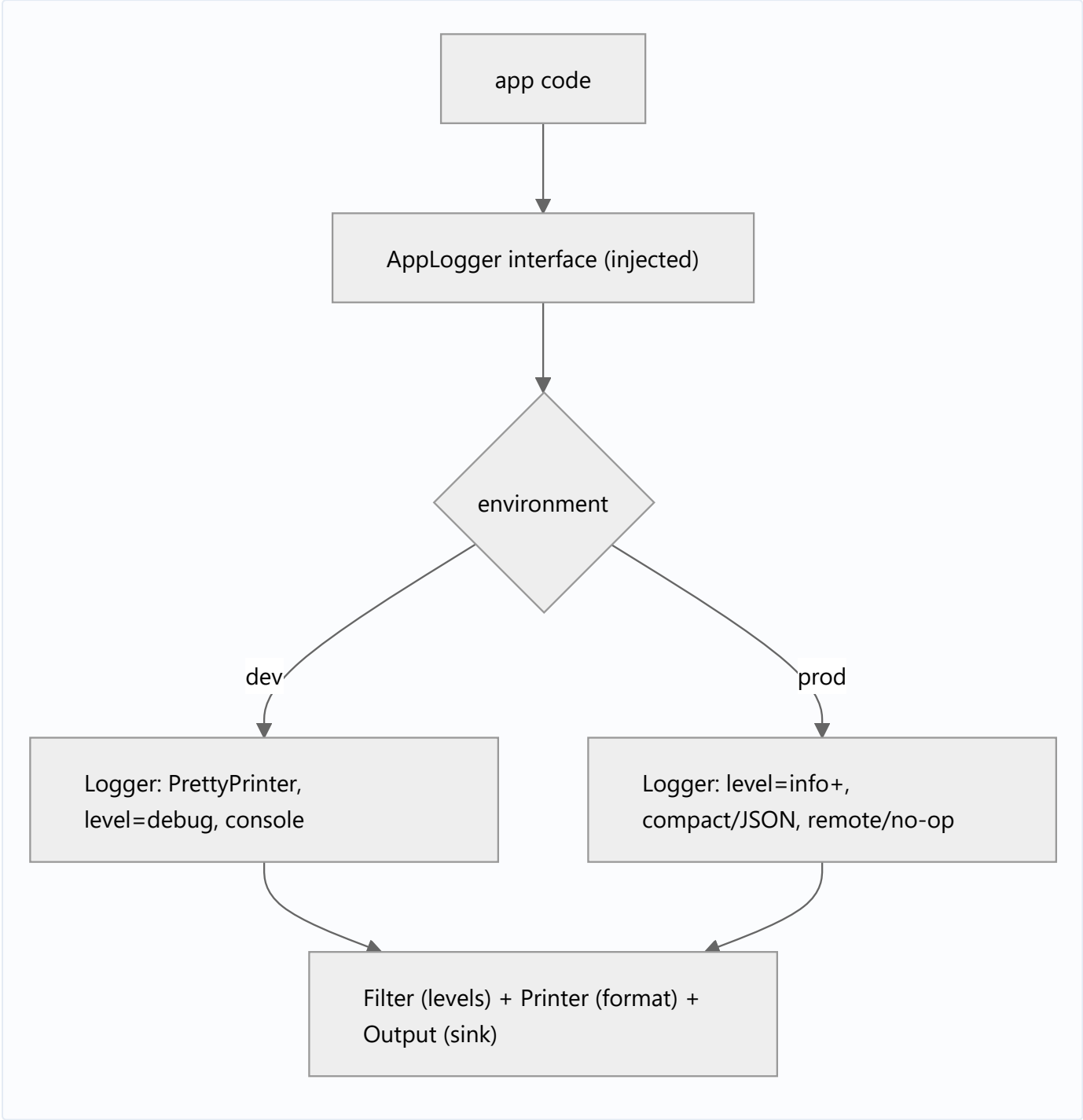
Real-world analogy

`logger` is a **configurable PA system**: a **filter** decides which announcements are loud enough to broadcast (levels), a **printer** decides the wording/format (verbose backstage vs terse public), and **outputs** decide where it plays (backstage speakers vs recorded to tape/remote). Wrapping it behind your own **"announcements service"** means every department calls the same simple interface, and you can re-wire the PA (swap vendor, change format) without changing how departments make announcements.

Problem Statement

Set up logging that's pretty and verbose in debug, filtered/structured (and remote-ready) in release, accessed through a single injected `AppLogger` interface (not raw `Logger()` calls), so you can swap implementations and test easily. You'll configure `logger` and wrap it behind a facade.

Internal Working



- `logger` **anatomy**: a `Logger` composes a `LogFilter` (which levels pass — e.g., `DevelopmentFilter` vs `ProductionFilter`), a `LogPrinter` (formatting — `PrettyPrinter` with boxes/emoji/stack for dev; a compact/JSON printer for prod), and a `LogOutput` (where it goes — `ConsoleOutput`, file, or a custom remote output — 04_remote_logging_and_observability.md). Call `logger.d/i/w/e/f(...)` by level.
- **Dev config**: `PrettyPrinter` (readable, stack traces, colors), filter at **debug** — rich local diagnostics.
- **Prod config**: filter at **info/warning+**, a **compact/structured (JSON) printer**, output to a **remote sink** or a **no-op** (to strip noise) — lean, searchable, PII-safe

(03_production_logging_and_redaction.md).

- **The abstraction (crucial)**: define **your own `AppLogger` interface** (`debug/info/warn/error(event, fields, error, stack)`); implement it over `logger` (dev/prod variants). App code depends on `AppLogger` , **not** `logger` — so you can swap libraries, add redaction/remote, or use a fake in tests without touching call sites.
- **One instance via DI**: register the environment-appropriate `AppLogger` in your DI container (Module 14) and inject it — **don't** `new Logger()` all over.
- **Environment selection**: choose dev vs prod impl by build mode/flavor (`kReleaseMode` /flavor config); dev-only detail is stripped in release.
- **Structured fields**: have your facade accept an **event + fields map**, formatted for humans (dev) or JSON (prod) — consistent structure across environments.

Memory Representation

One shared logger/facade instance (via DI). Filter/printer/output are small configured objects. Log records flow through printer → output; no accumulation unless a buffering output batches for remote.

Compiler Behavior

`kReleaseMode` /flavor branches let dead dev-only config be tree-shaken; the facade indirection is free at runtime.

Runtime Behavior

Below-threshold levels are filtered before formatting (cheap). Dev prints pretty to console; prod emits compact/JSON to sink/remote/no-op. Swapping impls changes behavior app-wide without call-site changes.

Flutter Engine Behavior

Console output shows in the IDE/DevTools; custom outputs (file/remote) bypass the console.

Dart VM Behavior

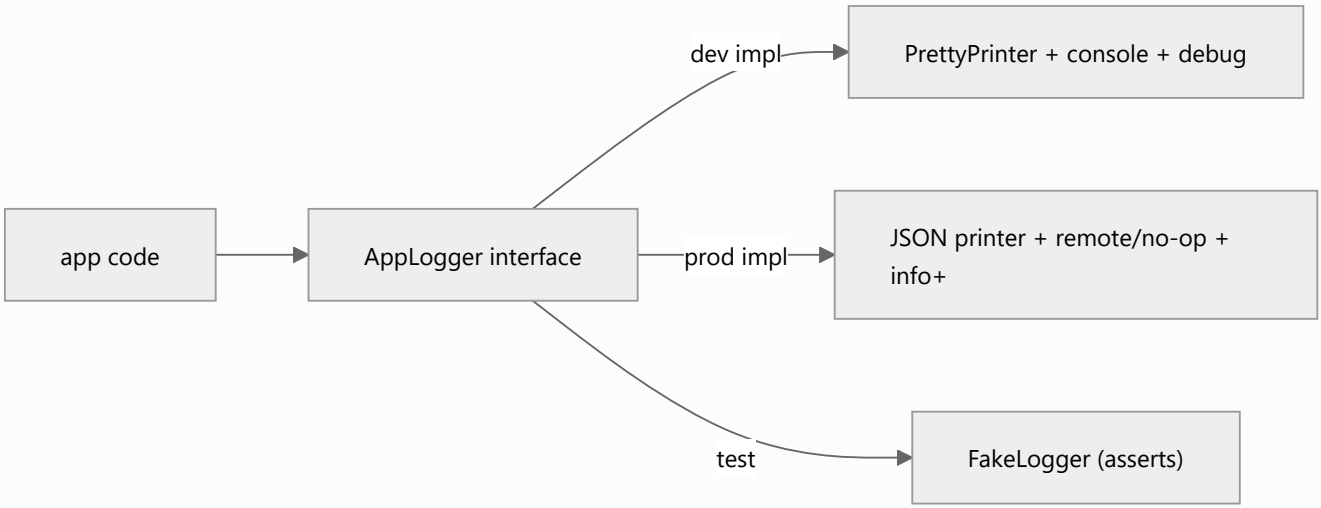
Not applicable beyond tree-shaking dev branches.

Examples

```
import 'package:logger/logger.dart';
import 'package:flutter/foundation.dart';

// Your OWN abstraction — app depends on this, not on `logger`
abstract class AppLogger {
  void debug(String event, [Map<String, Object?>? fields]);
  void info(String event, [Map<String, Object?>? fields]);
  void warn(String event, [Map<String, Object?>? fields]);
  void error(String event, {Object? error, StackTrace? stack, Map<String, Object?>? fields});
}

// Implementation over the `logger` package, configured per environment
class LoggerAppLogger implements AppLogger {
  final Logger _logger;
  LoggerAppLogger._(this._logger);

  factory LoggerAppLogger.dev() => LoggerAppLogger._(Logger(
        filter: DevelopmentFilter(),                // debug+ in dev
        printer: PrettyPrinter(methodCount: 2),     // readable
        output: ConsoleOutput(),
      ));

  factory LoggerAppLogger.prod({required LogOutput remote}) => LoggerAppLogger._(Logger(
        filter: ProductionFilter(),                 // info+ in prod
        printer: _JsonPrinter(),                    // compact/structured
        output: remote,                             // remote sink (or no-op)
      ));

  @override void debug(String e, [Map<String, Object?>? f]) => _logger.d(_fmt(e, f));
  @override void info(String e, [Map<String, Object?>? f]) => _logger.i(_fmt(e, f));
  @override void warn(String e, [Map<String, Object?>? f]) => _logger.w(_fmt(e, f));
  @override void error(String e, {Object? error, StackTrace? stack, Map<String, Object?>? f}) =>
      _logger.e(_fmt(e, f), error: error, stackTrace: stack);
  Object _fmt(String e, Map<String, Object?>? f) => {'event': e, ...?f}; // structured
}

// DI: register the right impl by environment; inject AppLogger everywhere
// getIt.registerSingleton<AppLogger>(kReleaseMode
```

```
//     ? LoggerAppLogger.prod(remote: RemoteLogOutput())
//     : LoggerAppLogger.dev());
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`new Logger()` scattered everywhere	Coupled, inconsistent, untestable	One `AppLogger` facade via DI
Same config dev + prod	Noisy/leaky or unreadable	Environment-specific filter/printer/output
App depends on `logger` directly	Can't swap/redact/remote centrally	Depend on your interface
PrettyPrinter in prod	Verbose/perf, not structured	Compact/JSON prod printer
No prod threshold	Log spam/cost	Filter at info/warning+
Unstructured facade	Not searchable	Event + fields map
Can't fake in tests	Coupled to real logger	Interface → fake impl

Best Practices

- Depend on **your own** `AppLogger` **interface**, not the package; implement it over `logger` with **environment-specific** filter/printer/output.
- **Dev**: PrettyPrinter, debug threshold, console. **Prod**: compact/JSON printer, info/warning+ threshold, remote sink or no-op.
- Register **one instance via DI** and inject it (no scattered `Logger()`); select dev/prod impl by **build mode/flavor**.

- Make the facade **structured** (event + fields), enabling **swap/redaction/remote/testing** without touching call sites.

Performance

Filtering below threshold before formatting keeps prod cheap; PrettyPrinter is dev-only. The facade adds negligible indirection. Remote/buffering outputs should batch (async) so logging never blocks — covered in 04_remote_logging_and_observability.md.

Advantages / Disadvantages

- **+** Readable dev + lean prod logs, one injection point, swappable/testable, structured, environment-aware.
- **−** Setup + facade boilerplate, must pick/maintain config per environment, discipline to route all logging through the facade.

Interview Questions

1. 🟢 **What three parts make up a `logger` `Logger`?** — A filter (which levels emit), a printer (formatting), and an output (sink) — configured per environment.
2. 🟢 **Why wrap `logger` behind your own interface?** — So the app depends on your abstraction (swap library, add redaction/remote, fake in tests) rather than the package, from one place.
3. 🟡 **How should dev vs prod logging differ?** — Dev: PrettyPrinter + debug threshold + console; prod: compact/JSON + info/warning+ threshold + remote/no-op.
4. 🟡 **Why not `new Logger()` throughout the app?** — It's coupled, inconsistent, and untestable; register one `AppLogger` via DI and inject it.
5. 🟡 **How do you select the environment config?** — By build mode/flavor (`kReleaseMode`/flavor), branching to dev/prod impls (dev config tree-shaken in release).
6. 🔴 **How does the facade enable redaction/remote later?** — All logging flows through one implementation, so you add redaction/remote output there without touching call sites.
7. 🔴 **How do you test logging?** — Inject a fake `AppLogger` and assert on captured events/levels — impossible if code calls `Logger()` directly.

Senior Engineer Tips

- Define the `AppLogger` interface first and treat `logger` as an implementation detail; this one indirection is what lets you add redaction, remote shipping, and tests later without a refactor.
- Configure prod deliberately (threshold + structured printer + remote/no-op) rather than shipping dev's PrettyPrinter — the default dev config is a perf/noise/leak liability in release.

- Register one logger in DI and inject it everywhere; scattered `Logger()` instances are the reason logging config drifts and can't be centrally redacted.

Architect Perspective

The facade + environment config is the seam that makes logging manageable: app code speaks one structured interface, while the implementation varies by environment and can grow redaction, sampling, and remote shipping without ripple. Injected via DI and swappable for a fake, it's testable and future-proof — the foundation the production and remote layers extend (03_production_logging_and_redaction.md, 04_remote_logging_and_observability.md, Module 14).

Summary

- `logger` = filter (levels) + printer (format) + output (sink); configure dev (pretty/debug/console) vs prod (JSON/info+/remote or no-op).
- Depend on your own `AppLogger` interface (structured event+fields), inject one instance via DI, select impl by build mode/flavor.
- The facade enables swap/redaction/remote/testing without touching call sites.

Revision Notes

- `Logger(filter, printer, output)`; `DevelopmentFilter` / `ProductionFilter`, `PrettyPrinter` (dev) vs compact/JSON (prod), `ConsoleOutput` /file/remote.
- Wrap in `AppLogger` interface (debug/info/warn/error, event+fields); one instance via DI; select by `kReleaseMode` /flavor.
- Dev: pretty/debug/console; prod: structured/info+/remote or no-op; facade enables redaction/remote/testing (fake) without call-site changes.

Practice Questions

1. What do a `logger` filter, printer, and output each control?
2. Why depend on your own logging interface instead of the package?
3. How do dev and prod logger configs differ?

Coding Questions

1. Define an `AppLogger` interface + a `logger`-backed impl.
2. Provide dev (pretty/console/debug) and prod (JSON/remote/info+) factories.
3. Register the right impl via DI by build mode and inject it.

Mini Project

Logging facade (Flutter): Build an `AppLogger` interface (structured event + fields) implemented over the `logger` package with dev (PrettyPrinter/console/debug) and prod (JSON printer/remote-or-no-op/info+) configurations, selected by build mode and registered once via DI. Add a fake for tests. Acceptance: app depends on `AppLogger` (no direct `Logger()` calls); dev is pretty+verbose, prod is structured+filtered; one injected instance; environment-selected; fake logger enables tests.

Production Logging & Redaction

> In production, logging is a **liability as much as an asset**: logs frequently end up on servers, in crash reports, and in third-party tools, so **logging PII/secrets (tokens, passwords, emails, card/health data) is a security and compliance breach** (GDPR/PCI/HIPAA). The rules: **redact/omit sensitive fields** (allowlist what you log, don't blocklist), **raise the level threshold** (info/warning+), **sample** high-volume logs, keep logging **cheap and async**, and **never log request/response bodies or auth headers** verbatim. Treat every log line as potentially public.

Introduction

This file covers making logging safe and sustainable in production: what you must never log, how to redact, sampling, performance, and the compliance stakes. It's where logging meets security (Module 37) — the discipline that separates helpful telemetry from a data breach.

Why this concept exists

Dev logging is verbose and freewheeling; production logging is shipped off-device to systems (and vendors) you don't fully control, retained, and searchable. A single `log('user $email token $token')` can leak credentials/PII into logs that persist for years — a reportable breach. Production logging must be deliberately minimal, redacted, and controlled.

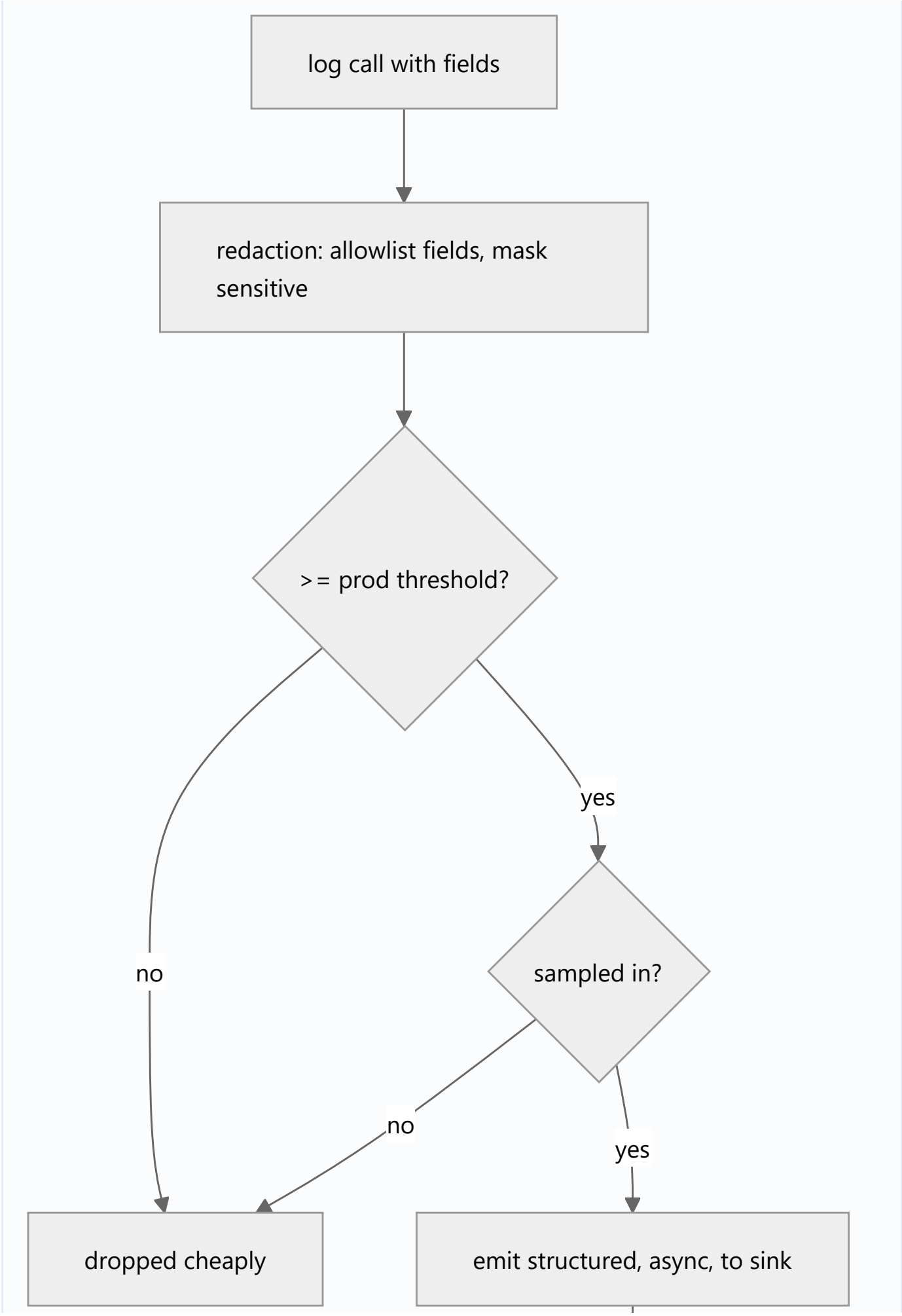
Real-world analogy

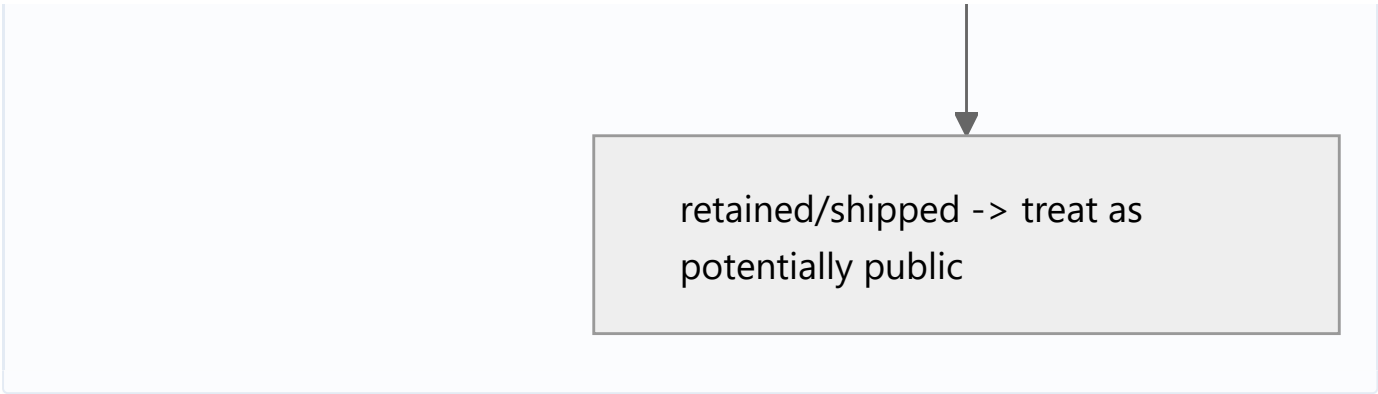
Production logs are like **security-camera footage that gets archived and shared with contractors**: useful for investigations, but if you record people's **PINs and ID cards** (PII/secrets) on tape, you've created a liability that outlives the incident. You **blur faces/mask numbers** (redact), **only keep relevant angles** (allowlist), and **don't record every second** in full (sample) — because the footage will be seen by others and kept for a long time.

Problem Statement

Make your prod logs safe: ensure tokens/passwords/emails/PANs never appear, log only allowlisted non-sensitive fields, cut noise via threshold + sampling, keep logging off the critical path, and log request/response **metadata** (status, duration, endpoint) — never bodies/auth headers. You'll add redaction, an allowlist, and sampling to the logging facade.

Internal Working


log call with fields
redaction: allowlist fields, mask sensitive
>= prod threshold?
no
yes
sampled in?
no
yes
dropped cheaply
emit structured, async, to sink



- **Never log** (hard rule): passwords, tokens/keys/secrets, full card numbers/CVV, government ids, health data, precise location, full emails/phones (unless required + consented), auth headers, request/response **bodies** with PII. These persist and are seen by others.
- **Redaction by allowlist (not blocklist)**: log only an **explicit set** of safe fields; mask/omit everything else. Allowlisting fails safe (a new PII field isn't logged by default); blocklisting fails open (you forget one). Mask partial values where useful (`****1234`), hash identifiers if you must correlate without exposing them.
- **Centralized redaction**: put redaction **in the logging facade** (02_logger_package_and_setup.md) so every log passes through it — not per call site. Redact known-sensitive keys and scrub values matching patterns (emails, long digit strings, JWTs) as a backstop.
- **Threshold + sampling**: prod filters at **info/warning+**; for high-volume events, **sample** (log 1-in-N or on error) to control cost/noise/PII exposure. Always log errors/warnings; sample debug/trace to zero in prod.
- **Performance**: logging must be **cheap + async** — don't block the UI/hot paths; guard expensive field construction behind the level check; batch remote shipping (04_remote_logging_and_observability.md). Excessive logging is a real perf + battery + cost drain.
- **Networking logs**: log **metadata** (method, endpoint, status, duration, correlation id) — **never** bodies or `Authorization` headers (Module 16); redact query params with secrets/tokens.
- **Compliance**: GDPR/PCI/HIPAA/CCPA constrain what you may log/retain about users; document a **data-handling policy** for logs (what, why, retention). Treat logs as regulated data.
- **Retention/access**: minimize retention, restrict who can read prod logs; assume vendors/tools can see them.

Memory Representation

Redaction transforms records before emission (masked/omitted fields). Sampling drops records probabilistically. Only allowlisted, safe fields reach the sink; nothing sensitive is retained.

Compiler Behavior

Dev-only verbose logs stripped in release; redaction/sampling run at runtime in the prod facade.

Runtime Behavior

Each prod log passes redaction → threshold → sampling → async emit. Below-threshold/sampled-out logs are dropped cheaply. Errors always emitted (+ reported to crash tooling — Module 52).

Flutter Engine Behavior

Not applicable.

Dart VM Behavior

Async emission keeps logging off the critical path; guarded construction avoids wasted work.

Examples

```
// Redaction in the facade: ALLOWLIST safe fields; mask/scrub the rest
const _allowed = {'event', 'screen', 'durationMs', 'statusCode', 'endpoint', 'correlationId'};

Map<String, Object?> _redact(Map<String, Object?> fields) {
  final out = <String, Object?>{};
  for (final e in fields.entries) {
    if (!_allowed.contains(e.key)) continue;          // allowlist: drop unknown keys
    out[e.key] = _scrub(e.value);                     // backstop pattern scrub
  }
  return out;
}
Object? _scrub(Object? v) {
  final s = '$v';
  if (RegExp(r'[\w.+-]+@[\w-]+\.\w+').hasMatch(s)) return '[email]';  // mask emails
  if (RegExp(r'\b\d{12,19}\b').hasMatch(s)) return '[card]';          // mask PAN-like
  if (s.startsWith('eyJ')) return '[jwt]';                            // mask tokens
  return v;
}

// Sampling high-volume events (always keep errors)
bool _sampledIn(Level level, int oneInN, int counter) =>
    level.index >= Level.warning.index || counter % oneInN == 0;

// Networking: log METADATA only — never bodies or Authorization
void logHttp(String method, String endpoint, int status, int ms, String corrId) {
  logger.info('http', {'endpoint': endpoint, 'statusCode': status, 'durationMs': ms, 'correlationId':
corrId});
  // NEVER: {'headers': req.headers, 'body': res.data}  // leaks tokens/PII
}
```

Diagrams

Common Mistakes

Mistake	Why it's a breach/bug	Fix
Logging tokens/passwords/PII	Security + compliance breach	Never log; redact/omit
Logging request/response bodies	Leaks PII/secrets	Log metadata only
Logging `Authorization` headers	Token leak	Redact headers
Blocklist redaction	Fails open (forget a field)	Allowlist safe fields
Verbose prod logging	Noise, cost, PII exposure, perf	Threshold + sampling
Synchronous/hot-path logging	Jank/battery	Async + guarded + batched
No retention/access policy	Compliance risk	Minimize retention, restrict access

Best Practices

- **Never log** secrets/PII (tokens, passwords, PAN/CVV, health, precise location, auth headers, PII bodies); treat every log line as **potentially public**.
- **Redact by allowlist** (log only explicit safe fields) with a **pattern-scrub backstop**, **centralized in the facade**; mask/hash where correlation is needed.
- Prod **threshold info/warning+** + **sample** high-volume logs (always keep errors); keep logging **cheap, async, batched**.
- Log **network metadata only** (never bodies/auth); follow **compliance** (GDPR/PCI/HIPAA) with a documented **retention/access** policy.

Performance

Redaction/sampling/threshold keep prod logging lean; async + batching keep it off the critical path; guarded construction avoids wasted work. Over-logging costs CPU, battery, bandwidth, and vendor bills — sampling is the main lever.

Advantages / Disadvantages

- **+** Safe, compliant, affordable production telemetry; incident-debuggable without leaking data; controlled volume/cost.
- **−** Requires disciplined redaction (allowlist upkeep), sampling can hide rare events, compliance overhead, less raw detail than dev.

Interview Questions

1. 🟢 **Why is logging PII/secrets in production dangerous?** — Logs are shipped, retained, and seen by systems/vendors you don't fully control; leaking tokens/PII is a security and

compliance (GDPR/PCI/HIPAA) breach.

2. 🟢 **What must you never log?** — Passwords, tokens/keys, full card numbers/CVV, health/gov ids, auth headers, and request/response bodies with PII.
3. 🟡 **Why allowlist rather than blocklist fields?** — Allowlisting fails safe (new sensitive fields aren't logged by default); blocklisting fails open (you forget one).
4. 🟡 **How do you log network calls safely?** — Metadata only (method, endpoint, status, duration, correlation id) — never bodies or `Authorization` headers.
5. 🟡 **How do you control log volume/cost in prod?** — Raise the threshold (info/warning+) and sample high-volume events (always keeping errors), async + batched.
6. 🔴 **Where should redaction live and why?** — Centralized in the logging facade, so every log passes through it — not scattered per call site (which you'll miss).
7. 🔴 **How do you correlate a user without exposing them?** — Log a hashed/opaque id (or correlation id), never the raw email/phone/PII.

Senior Engineer Tips

- Put redaction in the facade with an allowlist + a pattern-scrub backstop, and treat "does this log contain PII?" as a review checklist item — leaks happen one careless `log(user)` at a time.
- Never log request/response bodies or auth headers, even in dev habits that leak into prod; log metadata + a correlation id instead.
- Sample aggressively in prod (keep errors, sample the rest); the cost and PII exposure of full-volume logging sneaks up on you at scale.

Architect Perspective

Production logging is a security/compliance surface as much as an observability tool. Centralizing redaction (allowlist + scrub), threshold, and sampling in the logging facade makes safety the default — no call site can leak — while keeping telemetry useful and affordable. This is the logging counterpart to the untrusted-client/least-data discipline of security, and it's what makes remote shipping and crash correlation safe to enable (Module 37, 04_remote_logging_and_observability.md, Module 52).

Summary

- Treat prod logs as potentially public: never log secrets/PII/bodies/auth headers; redact by allowlist (+ scrub backstop), centralized in the facade.
- Threshold info/warning+ and sample high-volume logs (keep errors); log network metadata only; keep logging cheap/async/batched.
- Follow compliance (GDPR/PCI/HIPAA) with minimal retention + restricted access.

Revision Notes

- Never log: passwords/tokens/keys, PAN/CVV, health/gov ids, precise location, auth headers, PII bodies; treat logs as public.
- Redact by allowlist (fail safe) + pattern scrub (email/PAN/JWT) centralized in facade; mask/hash for correlation.
- Prod threshold info/warning+; sample high-volume (keep errors); async/batched/guarded; network metadata only; compliance + retention/access policy.

Practice Questions

1. Why is allowlist redaction safer than blocklist?
2. What can and can't you log about a network request?
3. How do threshold + sampling protect cost and privacy?

Coding Questions

1. Add allowlist + pattern-scrub redaction to the logging facade.
2. Implement level-threshold + sampling for prod (always keep errors).
3. Write a safe HTTP logger (metadata only, no bodies/auth).

Mini Project

Safe production logging (Flutter): Extend the logging facade with centralized redaction (allowlist safe fields + scrub emails/PAN/JWT), a prod info/warning+ threshold, sampling of high-volume events (always keeping errors), async emission, and a network logger that records metadata only. Verify no PII/secrets/bodies/auth headers appear. Acceptance: redaction allowlist + scrub in the facade; no secrets/PII/bodies/auth logged; threshold + sampling applied (errors always kept); async/cheap; network metadata only; retention/compliance noted.

Remote Logging & Observability

> On-device logs are invisible in production — so ship logs/events to a **remote sink** (Crashlytics/Sentry breadcrumbs, a logging backend, or your own endpoint) and turn them into **observability**: attach a **correlation/trace id** to tie together the logs, crash reports, and backend traces for a single user action; record **breadcrumbs** leading up to errors; and add **context** (session, app version, screen). Ship **async + batched** (never block the UI), respect **redaction** (03_production_logging_and_redaction.md), and integrate with your **crash reporter** so a crash arrives with the story of what led to it.

Introduction

This file elevates logging from "text on a device" to **observability**: remote sinks, crash-report integration, correlation/trace ids, breadcrumbs, and context — so you can diagnose production incidents across client and server. It's the remote/observability layer over the facade (02_logger_package_and_setup.md) and pairs with monitoring (Module 52).

Why this concept exists

A crash report saying "null in X" is far less useful than one preceded by "user tapped checkout → payment API 500 → retry → crash," correlated with the exact backend request. Observability — logs + traces + context, correlated by ids — is how you reconstruct and fix production incidents you can't reproduce. Remote shipping makes on-device logs visible; correlation makes them actionable.

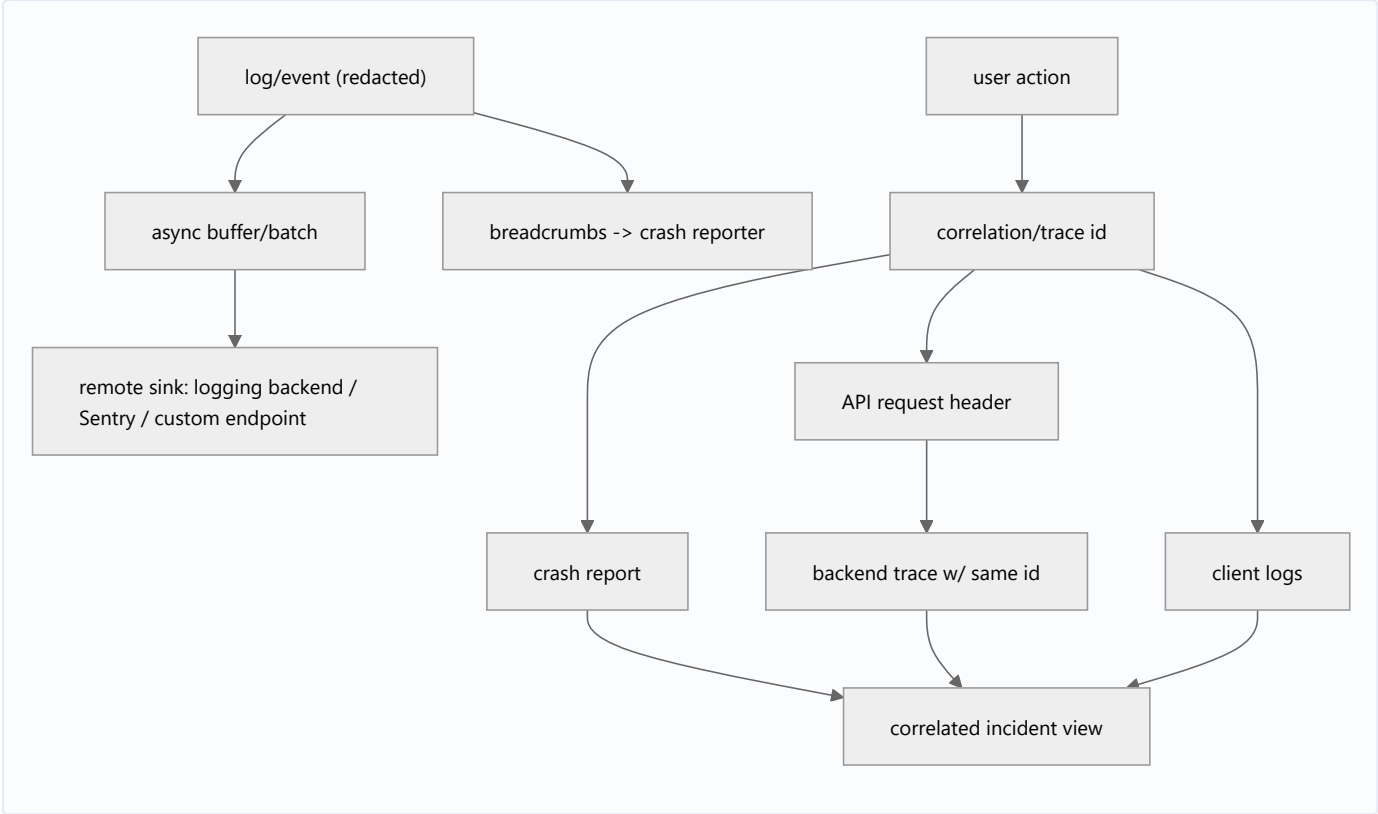
Real-world analogy

Local logs are a **black box that stays in the wreckage** — useless if you can't reach it. Remote logging **transmits the flight data live** to ground control. **Breadcrumbs** are the sequence of events before the incident; a **correlation id** is the **flight number** that lets ground control line up the cockpit recorder (client logs), the tower's records (backend traces), and the incident report (crash) for the *same* flight.

Problem Statement

Make production incidents diagnosable: ship logs/breadcrumbs to a remote sink + crash reporter, tag every log/request/crash for one user action with a shared **correlation id** (so client + server line up), attach context (version/session/screen), and do it **async/batched/redacted** without blocking the UI. You'll add a remote output, correlation ids, and crash integration.

Internal Working



- **Remote sink**: a custom `LogOutput` that **buffers + batches** records and ships them **async** to a backend (your endpoint, a logging service) or into a crash tool as **breadcrumbs**. Never block the UI; drop/limit on backpressure; retry with backoff; respect offline (queue).
- **Crash-report integration** (Module 52): forward logs as **breadcrumbs** so a crash arrives with the trail of events before it; report errors/fatals with the current context. The logging facade and crash reporter share one pipeline.
- **Correlation/trace id**: generate an id per **user action/request** (or per session), **attach it to every log**, **send it as a request header** (so the backend logs/traces carry the same id), and **include it in crash reports**. This is what lets you **stitch client logs + server traces + crash** for one action — the core of distributed tracing.
- **Context/breadcrumbs**: attach **app version, build, session id, screen/route, user (opaque id), device** to logs/reports; record breadcrumbs (navigation, taps, network results) so errors have a **narrative**.
- **Structured + redacted**: ship **structured** records (JSON) with **redaction applied first** (03_production_logging_and_redaction.md) — remote logs are the *most* exposed, so PII discipline is non-negotiable.
- **Sampling/volume**: sample high-volume logs, cap batch size/frequency, and prioritize errors — control cost and bandwidth (03_production_logging_and_redaction.md).
- **Observability trio**: logs (events), **traces** (correlated spans across client/server via the id), and metrics/crashes — together they answer "what happened, where, and why" (Module 52).

Memory Representation

A bounded async buffer holds pending records until flushed/batched to the sink (dropped/capped on overflow). Correlation ids are short strings threaded through logs/requests/crash context. Breadcrumbs are a rolling buffer in the crash tool.

Compiler Behavior

Not applicable.

Runtime Behavior

Logs are buffered and shipped async/batched; correlation ids flow client→server via headers; crashes arrive with breadcrumbs + context + id. Offline logs queue and flush on reconnect; backpressure drops lowest-priority records.

Flutter Engine Behavior

Not applicable; remote shipping is network I/O.

Dart VM Behavior

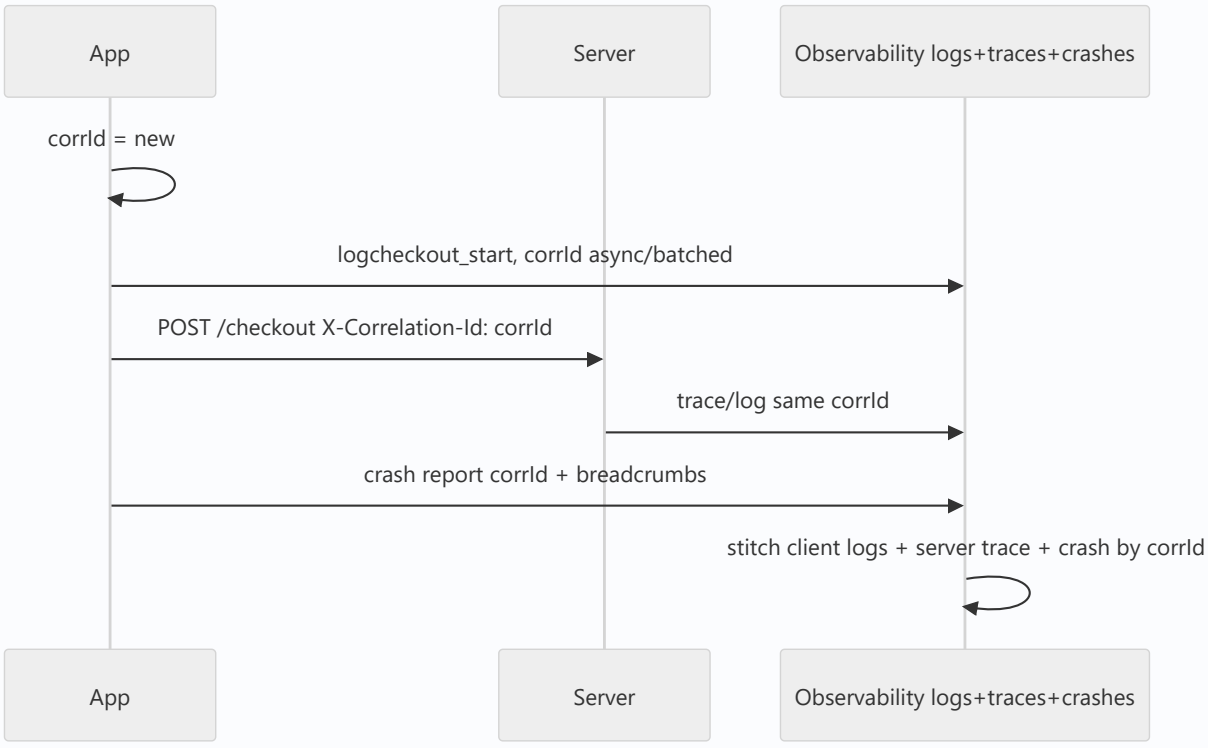
Async batching keeps shipping off the critical path; heavy serialization can be minimized/offloaded.

Examples

```
// Correlation id per user action, threaded through logs + request + crash
final corrId = _newCorrelationId();            // e.g., per checkout tap
logger.info('checkout_start', {'correlationId': corrId, 'screen': 'cart'});
final res = await dio.post('/checkout',
    options: Options(headers: {'X-Correlation-Id': corrId}));  // server logs same id
logger.info('checkout_result', {'correlationId': corrId, 'statusCode': res.statusCode});
// On crash: crashReporter.setCustomKey('correlationId', corrId);  // stitch it all together
```

```
// Remote LogOutput: async, batched, redacted, offline-queued
class RemoteLogOutput extends LogOutput {
  final _buffer = <Map<String, Object?>>[];
  @override
  void output(OutputEvent event) {
    _buffer.add(_redact(_toStructured(event)));   // redact BEFORE shipping
    if (_buffer.length >= 20) _flush();            // batch
  }
  Future<void> _flush() async {
    final batch = List.of(_buffer); _buffer.clear();
    try { await api.postLogs(batch); }             // async; retry/backoff; queue if offline
    catch (_) { /* re-queue or drop on backpressure */ }
  }
}

// Breadcrumbs feed the crash reporter (Module 52)
void breadcrumb(String event, Map<String, Object?> fields) {
  logger.debug(event, fields);
  crashReporter.addBreadcrumb(event, _redact(fields)); // trail before a crash
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Synchronous remote shipping	Blocks UI/jank	Async + batched buffer
No correlation id	Can't stitch client+server+crash	Thread an id through logs/requests/crash
Shipping unredacted logs	Worst-case PII leak (remote/retained)	Redact BEFORE shipping
No breadcrumbs/context	Crashes lack narrative	Breadcrumbs + version/session/screen
Unbounded buffer	Memory/backpressure issues	Bounded buffer, drop/cap on overflow
Ignoring offline	Logs lost	Queue + flush on reconnect
No sampling/caps	Cost/bandwidth blowout	Sample + batch caps (keep errors)

Best Practices

- Ship logs **async + batched** to a **remote sink** and integrate with the **crash reporter** (breadcrumbs + context); **bound** the buffer, **queue offline**, retry with backoff.
- Thread a **correlation/trace id** through **logs** → **request headers** → **crash reports** so client logs, server traces, and crashes stitch together per action.
- Attach **context** (version/build/session/screen/opaque user) and **breadcrumbs** so errors have a narrative; ship **structured + redacted** records.
- **Sample + cap** volume (keep errors); make it part of the **observability trio** (logs/traces/crashes) (Module 52); apply **redaction before shipping**.

Performance

Async batching + caps keep remote logging off the critical path and control bandwidth/cost. A bounded buffer prevents memory growth; offline queuing avoids loss without blocking. Sampling is the main cost lever. Correlation ids are tiny.

Advantages / Disadvantages

- **+** Production visibility, correlated client+server+crash diagnosis, breadcrumbs/context, faster incident resolution.
- **−** Infra + cost, PII risk if unredacted, buffering/backpressure/offline complexity, sampling can miss rare events.

Interview Questions

1. 🟢 **Why ship logs remotely?** — On-device logs are invisible in production; remote shipping (+ crash breadcrumbs) makes production behavior diagnosable.

2. 🟢 **What is a correlation/trace id for?** — To stitch together client logs, backend traces, and the crash report for a single user action (thread it through logs, request headers, and crash context).
3. 🟡 **How should remote shipping be done to avoid jank?** — Async + batched via a bounded buffer, with offline queuing and backoff retry — never synchronously on the UI thread.
4. 🟡 **What are breadcrumbs and why do they matter?** — The trail of events (navigation/taps/network) before an error, giving crash reports a narrative to reconstruct the incident.
5. 🟡 **What must you do before shipping logs remotely?** — Redact PII/secrets — remote/retained logs are the most exposed surface.
6. 🔴 **How do logs, traces, and crashes fit into observability?** — Together (correlated by ids) they answer what happened (logs), where across systems (traces), and why (crashes + context) — the observability trio.
7. 🔴 **How do you control remote-logging cost and volume?** — Sample high-volume logs (keep errors), cap batch size/frequency, and bound the buffer.

Senior Engineer Tips

- Introduce a correlation id per user action and propagate it to the backend header + crash context from day one; retrofitting cross-system correlation later is painful, and it's the single biggest debugging multiplier.
- Ship async/batched with a bounded buffer and offline queue, and redact before shipping; the two classic failures are UI jank from sync shipping and PII leaked into a remote store.
- Treat logs, traces, and crashes as one correlated system (breadcrumbs + context + id), not separate tools — that's what turns "it crashed" into "here's exactly what happened."

Architect Perspective

Remote logging + observability is where logging becomes an operational capability: correlated, contextual, redacted telemetry shipped safely and stitched across client and server. Built on the facade (structured + redacted) and integrated with crash reporting, it closes the loop from error handling (Module 38) to production diagnosis — the observability backbone that lets teams find and fix what they can't reproduce (Module 52, 03_production_logging_and_redaction.md).

Summary

- Ship logs async/batched to a remote sink + crash reporter (breadcrumbs, context), bounded + offline-queued + redacted.

- Thread a correlation/trace id through logs → request headers → crash reports to stitch client + server + crash per action.
- Add context/breadcrumbs, sample/cap volume; logs + traces + crashes = the observability trio.

Revision Notes

- Remote `LogOutput`: async, batched, bounded buffer, offline queue, backoff; feed crash reporter as breadcrumbs; redact BEFORE shipping.
- Correlation/trace id per action → logs + request header (`X-Correlation-Id`) + crash context → stitch client+server+crash.
- Context (version/session/screen/opaque user) + breadcrumbs; sample/cap (keep errors); observability trio (logs/traces/crashes — Module 52).

Practice Questions

1. How does a correlation id help diagnose an incident across client and server?
2. Why must remote shipping be async/batched and redacted?
3. What do breadcrumbs add to a crash report?

Coding Questions

1. Implement a batched, bounded, offline-queued `RemoteLogOutput` (redacted).
2. Thread a correlation id through a log, a request header, and crash context.
3. Feed breadcrumbs + context to a (stub) crash reporter.

Mini Project

Observability pipeline (Flutter): Extend the logging facade with a `RemoteLogOutput` (async, batched, bounded, offline-queued, redacted), integrate a (stub) crash reporter with breadcrumbs + context (version/session/screen), and thread a correlation id through logs → request headers → crash context for a user action. Add sampling/caps. Acceptance: logs ship async/batched (no UI block) + redacted; correlation id stitches client logs + request + crash; breadcrumbs + context attached; offline queued; volume sampled/capped (errors kept).

Logging Integration (Capstone: One Facade Across Environments)

> The maintainable shape: **one `AppLogger` facade**, injected via DI, that the whole app depends on — behind it, **environment-specific pipelines**: dev = pretty console + verbose; prod = **redact → threshold → sample → structured → async remote + crash breadcrumbs**, with **correlation ids** threading logs/requests/crashes. App code just calls `logger.info(event, fields)`; the facade owns levels, structure, redaction, sampling, environment selection, and remote/observability wiring. Logging becomes consistent, safe, cheap, correlated, and testable — the observability complement to the error strategy (Module 38).

Introduction

This module capstone composes fundamentals, the `logger` setup, production redaction, and remote observability into one coherent logging architecture. Scattered `print` / `Logger()` calls with per-site formatting/redaction are inconsistent and dangerous; one facade centralizes every concern. This file shows the composition and an end-to-end logged-and-correlated flow.

Why this concept exists

Logging spans levels, structure, environment differences, redaction, sampling, performance, remote shipping, and correlation — all cross-cutting. Left ad hoc, you get leaks, noise, cost, and inconsistency. One facade with a layered pipeline (like a repository for data) makes logging a single, governed capability, consistent with clean architecture and DI (Module 40, Module 14).

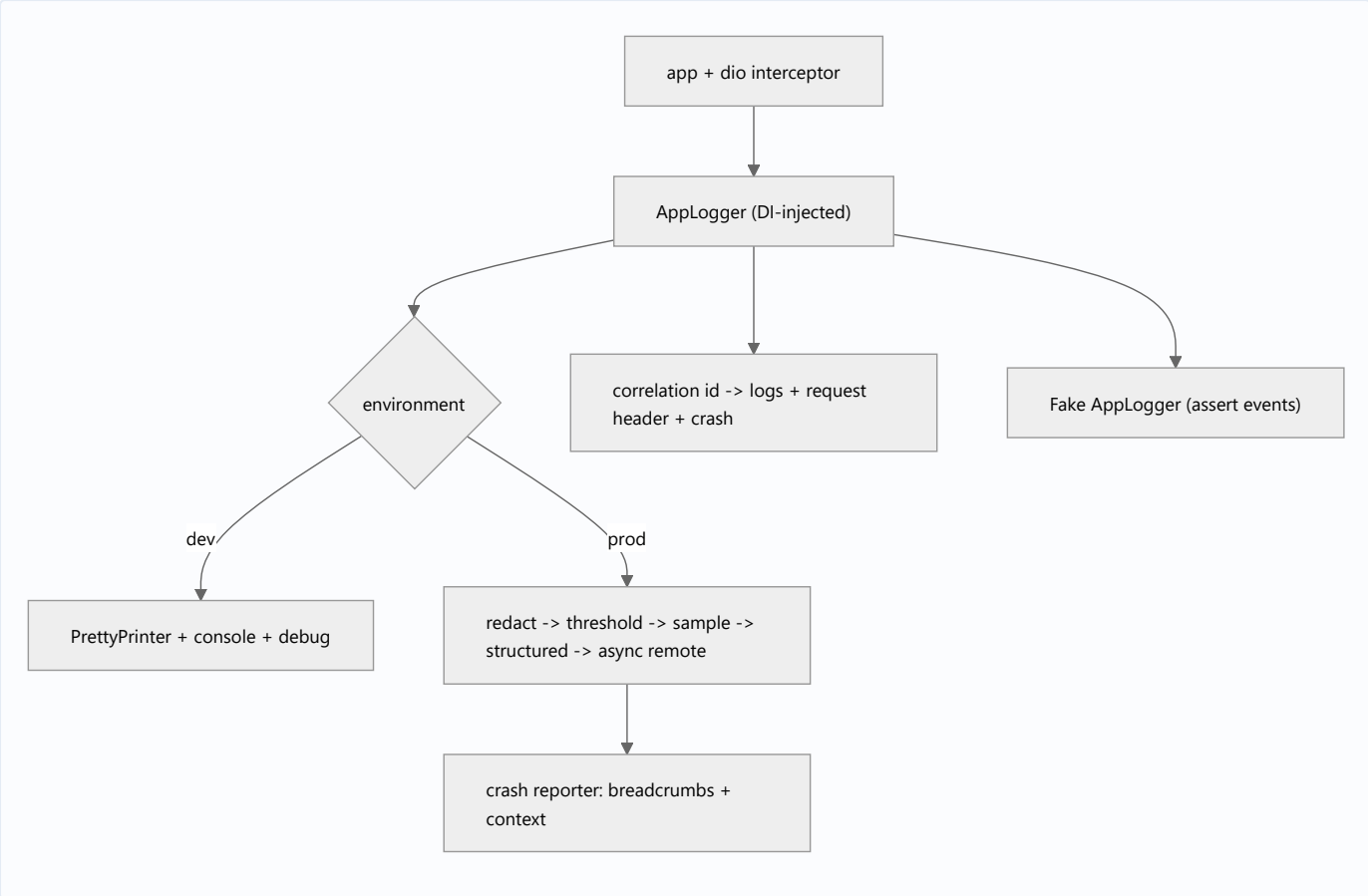
Real-world analogy

The facade is the **newsroom's single editorial desk**: reporters (features) submit stories the same way, and the desk applies house style (structure/levels), **redacts sensitive info** (legal), decides what runs and how prominently (threshold/sampling), and routes it to the right outlet (console in the newsroom, wire service in prod) with a **story slug** (correlation id) so follow-ups line up. No reporter self-publishes raw to the wire.

Problem Statement

Deliver production-grade logging: a DI-injected `AppLogger` used everywhere (with a `dio` request logger), dev pretty console + verbose, prod redacted/sampled/structured/async-remote with crash breadcrumbs and correlation ids stitching client+server+crash, and a fake for tests — all consistent and PII-safe. You'll compose every file in this module.

Internal Working



- **Single facade** (02_logger_package_and_setup.md): `AppLogger` interface (`debug/info/warn/error`, event + fields), the **only** logging dependency in app code; injected via DI as one instance.
- **Environment pipelines**: **dev** = PrettyPrinter/console/debug (verbose, readable); **prod** = the ordered pipeline **redact** → **threshold(info+)** → **sample** → **structured(JSON)** → **async batched remote** + crash breadcrumbs — selected by build mode/flavor.
- **Redaction always in the facade** (03_production_logging_and_redaction.md): allowlist + scrub before anything is emitted/shipped — no call site can leak.
- **Correlation + observability** (04_remote_logging_and_observability.md): a correlation id per action threads through logs, `dio` request headers, and crash context; breadcrumbs + context (version/session/screen) give crashes a narrative.
- **Cross-layer use**: data/domain/presentation log through the facade; a `dio` **interceptor** logs request **metadata** (never bodies/auth) with the correlation id; error handling (Module 38) logs failures (even recovered) via the facade.
- **Performance**: below-threshold dropped cheaply, async/batched remote, guarded construction, sampling — logging never janks or blows up cost.
- **Testability**: inject a **fake** `AppLogger` and assert emitted events/levels/fields (redacted) without console/network.

Memory Representation

One facade instance; a bounded async buffer for remote; correlation ids threaded through calls; breadcrumb ring buffer in the crash tool. Only redacted, allowlisted fields persist.

Compiler Behavior

Dev config/logs tree-shaken in release (build-mode branches); the facade indirection is free.

Runtime Behavior

Each call: redact → threshold → sample → format → (console dev / async remote prod) + breadcrumb; correlation id flows client→server; failures reported to crash tooling. Offline logs queue.

Flutter Engine Behavior

Console output in dev tooling; remote shipping is network I/O; crash breadcrumbs via the crash SDK.

Dart VM Behavior

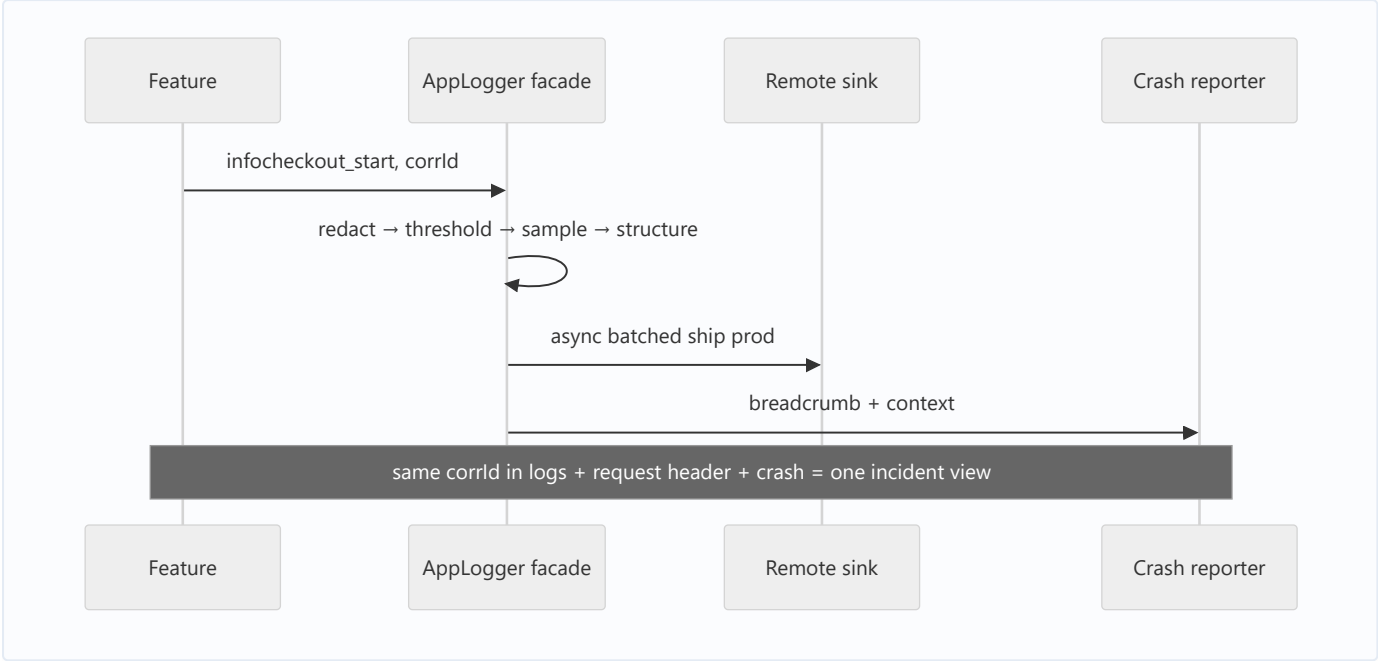
Async batching + guarded construction keep logging off the critical path; dev branches tree-shaken.

Examples

```
// Everything goes through ONE injected facade — dev vs prod pipeline chosen at composition
final AppLogger logger = getIt<AppLogger>();       // dev pretty | prod redact→sample→remote

// Cross-layer + correlation: one action, stitched across client/server/crash
Future<void> checkout(Cart cart) async {
  final corrId = newCorrelationId();
  logger.info('checkout_start', {'correlationId': corrId, 'itemCount': cart.items.length});
  try {
    final res = await api.post('/checkout',
        options: Options(headers: {'X-Correlation-Id': corrId})); // server logs same id
    logger.info('checkout_ok', {'correlationId': corrId, 'statusCode': res.statusCode});
  } catch (e, st) {
    logger.error('checkout_failed', error: e, stack: st, fields: {'correlationId': corrId});
    crashReporter.setKey('correlationId', corrId);  // stitch crash to logs/trace
    rethrow;                                          // error strategy handles UX (Module 38)
  }
}

// dio interceptor: metadata only (never bodies/auth), via the facade
dio.interceptors.add(InterceptorsWrapper(
  onResponse: (r, h) { logger.info('http', {
    'endpoint': r.requestOptions.path, 'statusCode': r.statusCode,
    'correlationId': r.requestOptions.headers['X-Correlation-Id'] }); h.next(r); },
));
```

```
// Testable: inject a fake facade and assert redacted events
test('logs checkout without PII', () async {
  final fake = FakeAppLogger();
  await Checkout(fake, FakeApi()).run(sampleCart);
  expect(fake.events.map((e) => e.name), contains('checkout_start'));
  expect(fake.events.every((e) => !containsPii(e.fields)), isTrue); // redaction holds
});
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`print` / `Logger()` scattered	Inconsistent, unsafe, untestable	One DI-injected `AppLogger`
Per-site redaction/format	Missed leaks, drift	Centralize in the facade
Same dev+prod pipeline	Noisy/leaky or unreadable	Environment-specific pipelines
No correlation id	Can't stitch incidents	Thread id through logs/requests/crash
Sync remote shipping	Jank	Async + batched
Logging bodies/auth/PII	Breach	Metadata only + redaction
Can't fake logging	Untestable	Facade → fake impl

Best Practices

- Route **all** logging through **one DI-injected** `AppLogger`; centralize **levels, structure, redaction, sampling, environment selection, remote/crash wiring** in it.
- **Dev**: pretty/verbose console; **prod**: redact → threshold → sample → structured → async remote + breadcrumbs (chosen by build mode/flavor).
- Thread a **correlation id** through logs → request headers → crash context; log **metadata only** (never bodies/auth/PII); log **failures via the facade** (Module 38).
- Keep logging **cheap/async**; provide a **fake** for tests; treat the facade as the single governed logging capability.

Performance

Centralization lets you apply threshold/sampling/async/guarded-construction uniformly, so logging never janks or overruns cost. Dev-only config is tree-shaken. Correlation is nearly free; the facade indirection is negligible.

Advantages / Disadvantages

- **+** Consistent, safe (redacted), cheap (sampled/async), correlated (observability), environment-aware, testable — one governed capability.
- **−** Upfront facade/pipeline design + DI wiring; discipline to route everything through it; per-environment config to maintain.

Interview Questions

1. 🟢 **Why centralize all logging behind one facade?** — To apply levels, structure, redaction, sampling, environment behavior, and remote/crash wiring consistently and safely from one place — and to make it swappable/testable.
2. 🟢 **How does dev vs prod behavior differ in the facade?** — Dev: pretty/verbose console; prod: redact → threshold → sample → structured → async remote + crash breadcrumbs, selected by build mode/flavor.
3. 🟡 **Why must redaction live in the facade?** — So no call site can leak PII/secrets — every log passes through one redaction step before emission/shipping.
4. 🟡 **How does correlation make logging observability?** — A shared id across client logs, request headers, and crash context stitches an incident together across client and server.
5. 🟡 **How do logging and error handling integrate?** — Failures (even recovered) are logged/reported via the facade with context, so the error strategy and observability share one pipeline.
6. 🔴 **How do you keep centralized logging performant?** — Threshold + sampling + async batching + guarded construction, uniformly applied in the facade; dev config tree-shaken.
7. 🔴 **How is the whole thing testable?** — Inject a fake `AppLogger` and assert emitted (redacted) events/levels/fields without console or network.

Senior Engineer Tips

- Make `AppLogger` the only logging dependency and wire the prod pipeline (redact→threshold→sample→async remote→breadcrumbs) once; every later concern (a new PII field, a remote vendor, sampling change) is then a single edit.
- Thread a correlation id from the first user action and propagate it to backend headers + crash context; it's the highest-leverage debugging investment you can make.

- Provide a fake logger and assert "no PII in emitted events" in tests; it turns redaction from a hope into a guarantee.

Architect Perspective

Logging integration makes observability a first-class, governed capability: one facade owns the pipeline (structure, redaction, sampling, environment, remote, correlation), app code stays trivial, and everything is safe, cheap, correlated, and testable. It mirrors the app's other boundaries (repositories, error strategy) and closes the observability loop with error handling and monitoring — so production incidents are diagnosable without leaking data or degrading performance (Module 38, Module 52, Module 14).

Summary

- One DI-injected `AppLogger` facade owns levels/structure/redaction/sampling/environment/remote/correlation; app code just logs events + fields.
- Dev = pretty/verbose console; prod = redact→threshold→sample→structured→async remote + crash breadcrumbs; correlation ids stitch client+server+crash.
- Metadata only (no bodies/auth/PII), cheap/async, log failures via the facade, fake for tests — one governed, observable, testable capability.

Revision Notes

- Single `AppLogger` via DI; centralize levels/structure/redaction/sampling/env-select/remote/crash; dev pretty/console, prod redact→threshold→sample→structured→async remote + breadcrumbs.
- Correlation id → logs + request header + crash context; metadata only; log failures via facade (Module 38); cheap/async; fake for tests.
- Environment by build mode/flavor (dev tree-shaken); one governed observability capability integrating with monitoring (Module 52).

Practice Questions

1. What does the facade centralize, and why does that matter?
2. How does dev vs prod logging differ in one facade?
3. How do correlation ids turn logs into observability?

Coding Questions

1. Compose the full `AppLogger` (dev + prod pipeline) and register via DI.
2. Add a `dio` interceptor + correlation id threaded to crash context.
3. Test with a fake asserting redacted events (no PII).

Mini Project

Unified logging capability (capstone — Flutter): Build one DI-injected `AppLogger` facade with a dev pipeline (pretty/console/verbose) and a prod pipeline (redact → threshold → sample → structured → async remote + crash breadcrumbs), a `dio` interceptor logging metadata only, correlation ids threaded through logs → request headers → crash context, and a fake for tests. Use it across layers + error handling. Acceptance: app depends only on `AppLogger` (no `print` / `Logger()`); env-specific pipelines; redaction centralized (no PII, metadata-only network); correlation stitches client+server+crash; async/cheap + sampled; failures logged via facade; fake-based tests assert redacted events; runs in dev + release.