29 · Device Features

Flutter Practice Notes

Contents

29 · Device Features

Introduction

This module covers accessing **native device capabilities** from Flutter through plugins: the **camera & gallery** (capture/pick images/video), **geolocation** (GPS position, streams, geocoding), **device sensors** (accelerometer/gyroscope/magnetometer + battery/connectivity/device info), and **connectivity** (network reachability). It shows how to wrap each behind a repository, handle permissions (Module 27 / Module 28), and stream results reactively.

Why this module exists

Most real apps read the physical device: scan/attach photos, locate the user, react to motion, adapt to being offline. Flutter reaches these through platform channels (Module 26) — usually via maintained plugins — but you must handle permissions, lifecycle, streams, and graceful degradation correctly to be reliable and battery-friendly.

Module map

#	File	Topic	Level
1	01_camera_and_gallery.md	Capture photos/video, pick from gallery (`camera` / `image_picker`)	🔵
2	02_geolocation.md	GPS position, location streams, accuracy, geocoding (`geolocator`)	🔴
3	03_sensors_and_device_info.md	Accelerometer/gyroscope, battery, device info (`sensors_plus` family)	🔵
4	04_connectivity.md	Network reachability + streams, offline handling (`connectivity_plus`)	🔵
5	05_device_integration.md	Capstone: features behind repositories, permissions, streams	🔴

> **Cross-references:** Permissions: 27 · permissions_and_manifest, 28 · infoplist_and_permissions. Platform channels (how plugins work): Module 26. Maps: Module 30. Streams: 02 · streams. Offline-first: Module 19. File handling: Module 34.

Prerequisites

26 Platform Channels, 27 Native Android, 28 Native iOS (permissions), 02 Advanced Dart (streams/async).

What you'll be able to do after this module

- Capture photos/video and pick media, handling permissions + files.

- Read the device location once and as a stream, with accuracy/geocoding.
- Read sensors, battery, connectivity, and device info reactively.
- Detect and react to offline/online transitions.
- Wrap every device feature behind a repository with graceful degradation.

Capstone

Device feature slice: A screen that attaches a photo (camera/gallery), tags it with the current location (geolocator + geocoding), shows an online/offline banner (connectivity stream), and reads device/battery info — every feature behind a repository, permissions handled, degrading gracefully when denied/unavailable.

Summary

Device features = maintained plugins over platform channels for camera/gallery, geolocation, sensors/device info, and connectivity. Handle permissions and lifecycle, prefer streams for continuous data, and wrap each behind a repository so the app stays testable, battery-friendly, and resilient offline.

Camera & Gallery (Capture and Pick Media)

> Capture photos/video with the `camera` plugin (live preview via a `CameraController`) or, for the common "just get an image" case, pick/capture with `image_picker` (launches the system UI) — both require permissions (Android/iOS), return files you must manage, and belong behind a repository.

Introduction

Two levels of camera access: `image_picker` (simplest — hand off to the OS camera/gallery UI, get a file back) and `camera` (full control — live preview, resolution, flash, custom capture UI). This file covers when to use each, the controller lifecycle, permissions, and file handling.

Why this concept exists

Apps constantly attach images (avatars, receipts, KYC, posts). Flutter can't access the camera sensor directly, so plugins bridge to the native camera/gallery via platform channels (Module 26). `image_picker` covers 90% of cases cheaply; `camera` handles custom capture experiences (scanners, filters).

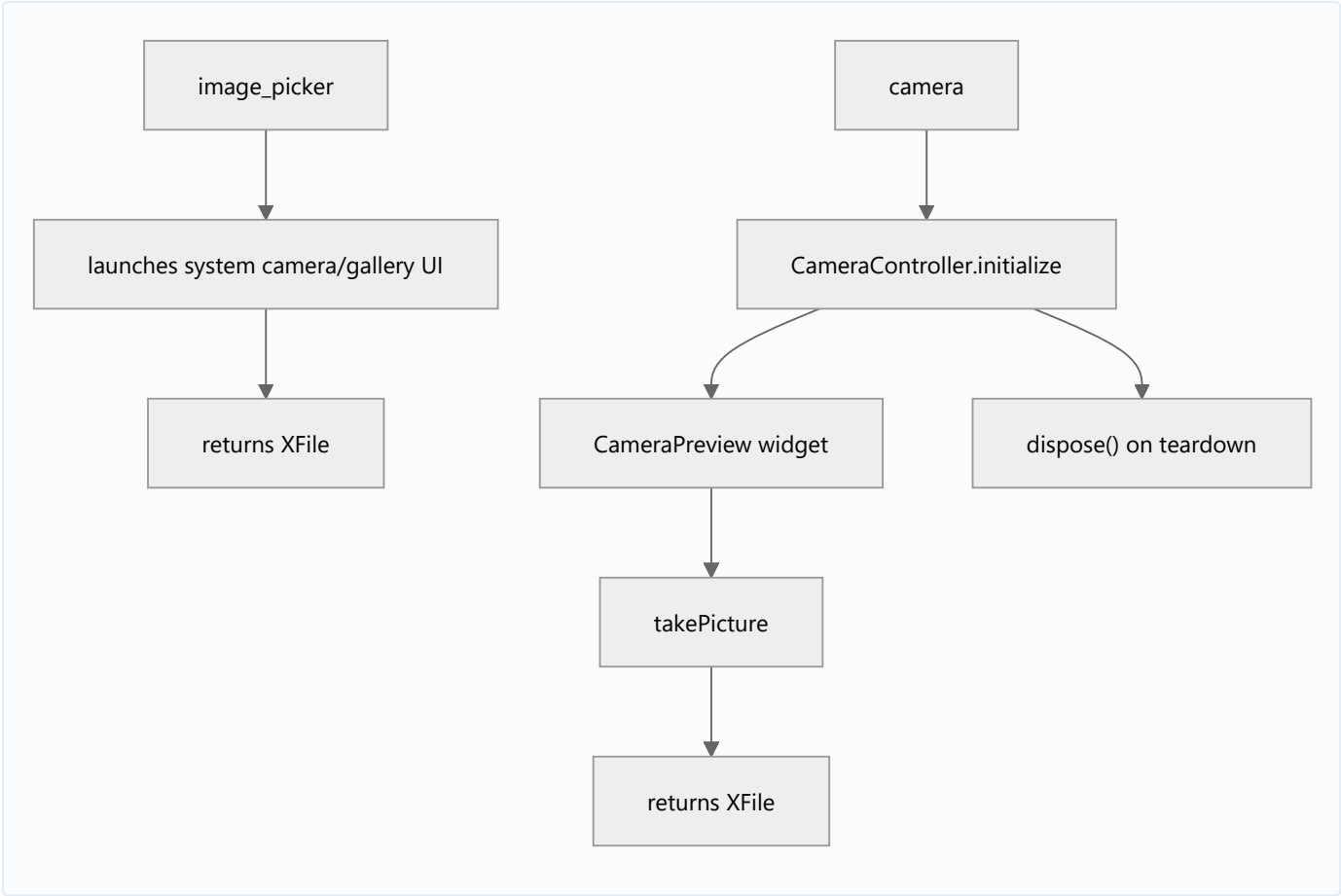
Real-world analogy

`image_picker` is **ordering a photo from a photo booth** — you press a button, the booth handles everything, you get a print. `camera` is **operating your own studio camera** — you control the lens, preview, and shutter, and you're responsible for setting up and tearing down the equipment (the `CameraController`).

Problem Statement

Let the user attach a profile photo — either pick from the gallery or take a new one — and (for a document scanner) show a live preview with a custom capture button. You'll use `image_picker` for the simple case and `camera` for the custom one, behind a repository, with permissions handled.

Internal Working



- **`image_picker`**: `pickImage(source: ImageSource.camera | .gallery)` / `pickVideo` returns an `XFile` (or null if cancelled). No preview/controller — the OS handles the UI. Best default for attach-a-photo.
- **`camera`**: `availableCameras()` → construct a `CameraController(camera, ResolutionPreset)` → `await controller.initialize()` → show `CameraPreview(controller)` → `controller.takePicture()` / `startVideoRecording()`. You **must** `dispose()` the controller and handle app lifecycle (release on pause, reinit on resume — 08 · lifecycle).
- **Permissions**: camera needs runtime permission + usage strings; gallery/photos too on iOS. Handle denied/restricted → Settings (28 · infoplist_and_permissions).
- **Files**: returned `XFile` points to a temp/cache path — copy to app storage if you need to keep it (Module 34); compress large images before upload.
- **Repository**: expose `Future<File?> pickAvatar()` / `captureDocument()`; the UI never touches the plugin directly (testable, swappable).

Memory Representation

Camera preview holds native buffers + a texture; full-resolution images are large in memory — compress/resize. The `CameraController` owns native resources until disposed.

Compiler Behavior

Not applicable; plugin bridges to native at runtime.

Runtime Behavior

`image_picker` suspends your app while the OS UI runs, then resumes with the file. `camera` streams preview frames continuously (battery/heat) — release when not visible.

Flutter Engine Behavior

`CameraPreview` uses a texture/platform view composited into the Flutter surface (28 · platform_views_ios); preview is engine-composited each frame.

Dart VM Behavior

Not applicable; capture/encoding happen natively, results marshalled over channels.

Examples

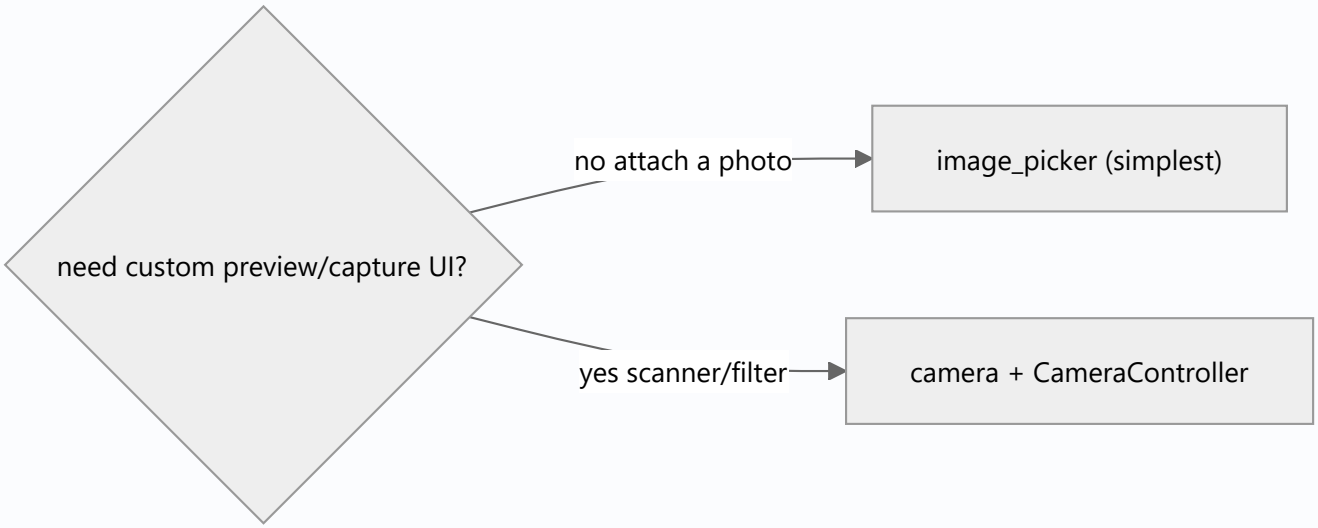
```
import 'package:image_picker/image_picker.dart';
import 'dart:io';

// Simple, most common case: pick or capture a single image
class MediaRepository {
  final _picker = ImagePicker();

  Future<File?> pickImage({required bool fromCamera}) async {
    final XFile? x = await _picker.pickImage(
      source: fromCamera ? ImageSource.camera : ImageSource.gallery,
      maxWidth: 1600,          // downscale to keep memory/upload small
      imageQuality: 85,        // compress
    );
    return x == null ? null : File(x.path); // null = user cancelled
  }
}
```

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

// Full control: live preview + custom capture (e.g., a document scanner)
```

```
class ScannerScreen extends StatefulWidget {
  const ScannerScreen({super.key});
  @override
  State<ScannerScreen> createState() => _ScannerScreenState();
}

class _ScannerScreenState extends State<ScannerScreen> {
  CameraController? _controller;

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

  Future<void> _init() async {
    final cameras = await availableCameras();
    final c = CameraController(cameras.first, ResolutionPreset.high, enableAudio: false);
    await c.initialize();
    if (mounted) setState(() => _controller = c);
  }

  @override
  void dispose() {
    _controller?.dispose();           // MUST release native resources
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    final c = _controller;
    if (c == null || !c.value.isInitialized) {
      return const Center(child: CircularProgressIndicator());
    }
    return Stack(children: [
      CameraPreview(c),
      Align(
        alignment: Alignment.bottomCenter,
        child: FloatingActionButton(
          onPressed: () async {
```

```
            final XFile shot = await c.takePicture();
            // hand shot.path to the repository / next screen
          },
          child: const Icon(Icons.camera),
        ),
      ),
    ]);
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Not disposing `CameraController`	Native leak, camera stays busy	`dispose()` in State; release on pause
Ignoring app lifecycle	Preview breaks on resume	Reinit on resume, release on pause
Not handling cancellation	Null crash	`XFile?` can be null (user cancelled)
Keeping full-res images in memory	OOM / jank	`maxWidth` / `imageQuality`, resize/compress
Using `camera` for a simple pick	Needless complexity	Use `image_picker`
Assuming the temp file persists	File lost/cleared	Copy to app storage if keeping (Module 34)
Skipping permissions/usage strings	Crash/denied	Handle permissions + usage strings

Best Practices

- Default to `image_picker`; use `camera` only for custom preview/capture; always `dispose()` the controller and handle **lifecycle** (release on pause).
- **Compress/resize** (`maxWidth`, `imageQuality`) before keeping/uploading; copy temp files to app storage if persisting (Module 34).
- Handle **permissions** + denied/restricted (Settings fallback) and **cancellation** (nullable result); degrade gracefully.
- Wrap in a **repository** (`pickAvatar` / `captureDocument`) so UI/tests don't touch the plugin.

Performance

Live preview streams frames (battery/heat) — release when off-screen. Full-res images are memory-heavy; downscale. `image_picker` is cheap (OS handles capture).

Advantages / Disadvantages

- **+** Rich media capture/selection with little code; `camera` gives full control; cross-platform.
- **−** Permissions/lifecycle/disposal responsibility, native resource + memory cost, temp-file management, custom capture complexity.

Interview Questions

1. 🟢 **When use `image_picker` vs `camera`?** — `image_picker` for the common attach-a-photo (OS handles UI); `camera` when you need a live preview/custom capture (scanner/filter).
2. 🟢 **What does `pickImage` return and what if the user cancels?** — An `XFile?`; it's `null` on cancellation, so handle that.
3. 🟡 **Why must you dispose the `CameraController`?** — It holds native camera resources/texture; not disposing leaks and keeps the camera busy.
4. 🟡 **How do you handle app lifecycle with the camera?** — Release the controller on pause and reinitialize on resume (08 · lifecycle).
5. 🟡 **How do you avoid OOM with large images?** — Downscale/compress (`maxWidth`, `imageQuality`) and don't hold full-res in memory.
6. 🔴 **Where do captured files live and how do you persist them?** — In temp/cache; copy to app storage (Module 34) if you need to keep them.
7. 🔴 **How does `CameraPreview` render in Flutter?** — Via a texture/platform view composited into the Flutter surface by the engine.

Senior Engineer Tips

- Reach for `image_picker` first — most "camera" tickets are really "attach an image."
- Treat the `CameraController` like a native resource: dispose it, and wire `WidgetsBindingObserver` to release/reinit across lifecycle.
- Compress at capture time; shipping full-res images silently bloats uploads and memory.

Architect Perspective

Camera/gallery is a device-capability + permission + file-lifecycle concern. Wrapping it behind a repository (returning domain files/paths), compressing early, and handling permissions/lifecycle keeps the feature testable, battery-friendly, and swappable — integrating with file handling, uploads, and offline queues (Module 34, Module 19).

Summary

- `image_picker` = simple pick/capture (OS UI, `XFile?`); `camera` = full control (controller + preview + `takePicture`, must dispose).
- Handle permissions, lifecycle, cancellation; compress/resize; persist temp files if needed.
- Wrap behind a repository; degrade gracefully.

Revision Notes

- `image_picker`: `pickImage(source, maxWidth, imageQuality)` → `XFile?` (null = cancelled). Default choice.
- `camera`: `availableCameras()` → `CameraController(...).initialize()` → `CameraPreview` → `takePicture()`; **dispose** + lifecycle.
- Permissions + usage strings; temp files → copy to persist; compress before upload; repository wrapper.

Practice Questions

1. When would you choose `camera` over `image_picker`?
2. What resources does a `CameraController` hold and how do you release them?
3. How do you keep captured images from bloating memory/uploads?

Coding Questions

1. Implement `MediaRepository.pickImage(fromCamera)` returning a compressed `File?`.
2. Build a camera preview screen with capture that disposes correctly.

3. Handle lifecycle: release the controller on pause, reinit on resume.

Mini Project

Photo attach feature (Flutter): Build a `MediaRepository` that picks from gallery or captures via `image_picker` (compressed), plus an optional custom-capture screen using `camera` (preview + capture, proper disposal + lifecycle). Handle permissions/cancellation and copy kept files to app storage. Acceptance: pick + capture both work; images compressed; controller disposed + lifecycle-safe; cancellation handled; feature behind a repository; runs on device.

Geolocation (Position, Streams, Accuracy, Geocoding)

> Read the device location with `geolocator`: a one-shot `getCurrentPosition()` or a continuous `getPositionStream()`, tuned by an **accuracy/distance filter** (higher accuracy = more battery); check **service enabled + permission** (incl. iOS while-in-use vs always), and convert between coordinates and addresses with **geocoding** — all behind a repository.

Introduction

Location powers maps, delivery, geofencing, and "near me" features. `geolocator` provides current position, a position stream, distance calculations, and settings checks; `geocoding` converts lat/lng ↔ address. This file covers accuracy vs battery, the permission/service flow, streams, and wrapping it cleanly.

Why this concept exists

GPS is a shared, battery-expensive sensor gated by permissions and a user-toggleable service. Flutter reaches it via a plugin over platform channels (Module 26). You must request the right permission tier, respect battery (accuracy/interval), and handle "service off"/"denied" states.

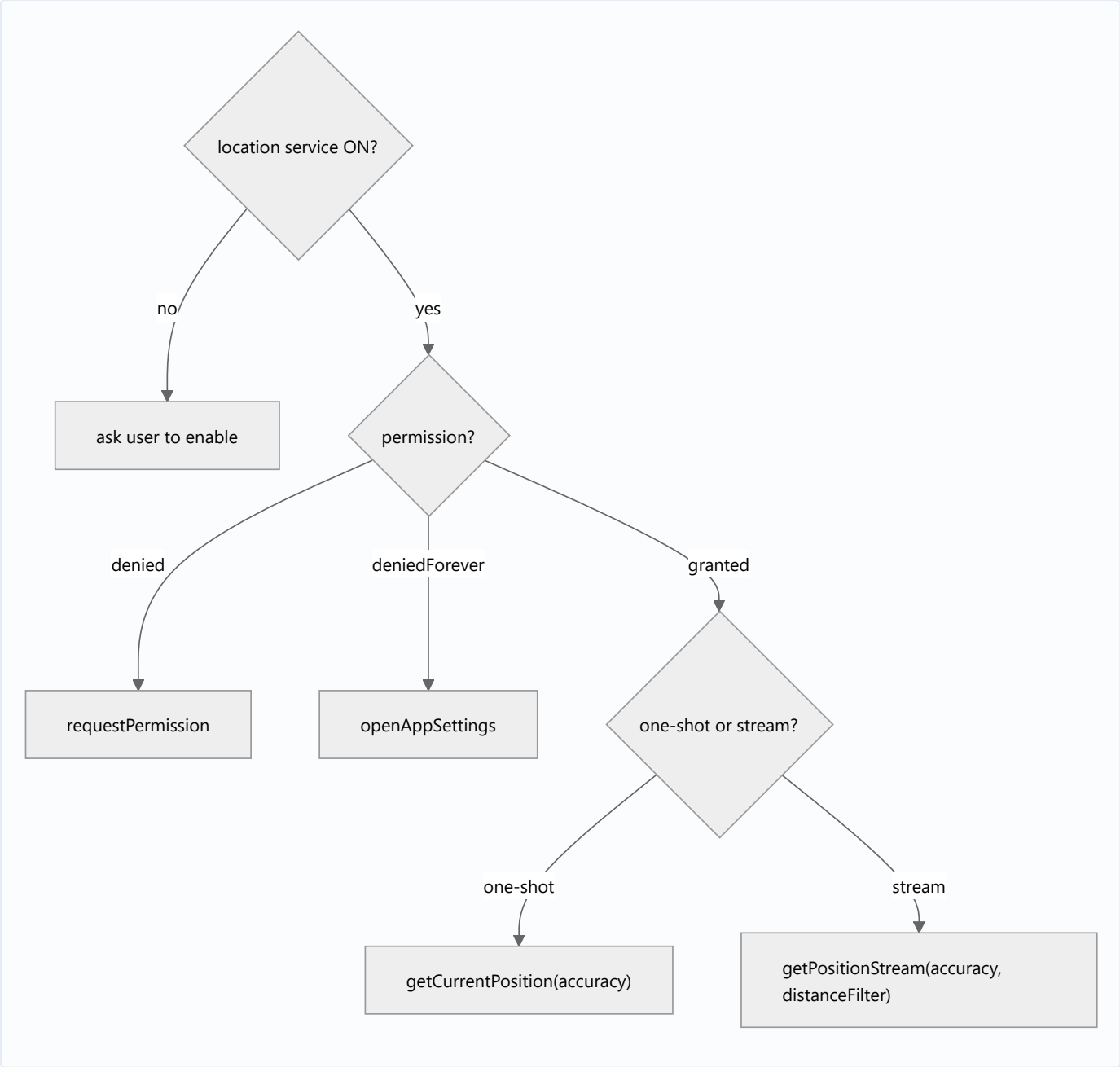
Real-world analogy

Asking for location is like **asking a navigator for your position**: first they must have the **map turned on** (location service enabled) and **permission to tell you** (granted), then you choose **how precise** (rough neighborhood = cheap/fast, exact GPS = slow/battery). A **stream** is the navigator calling out your position as you move.

Problem Statement

Show the user's current location on a "near me" screen, keep it updated as they move (for live tracking), display a human-readable address, and handle service-off/denied gracefully. You'll use `getCurrentPosition`, `getPositionStream`, and `geocoding`, behind a repository.

Internal Working



- **Preflight**: `isLocationServiceEnabled()` (user can toggle GPS off) → `checkPermission()` / `requestPermission()`. iOS distinguishes **whileInUse** vs **always** (background needs always + justification — 28 · ios_integration); Android has coarse vs fine.
- **One-shot**: `getCurrentPosition(desiredAccuracy: LocationAccuracy.high)` returns a `Position` (lat/lng, accuracy, speed, heading, timestamp).
- **Stream**: `getPositionStream(locationSettings: LocationSettings(accuracy, distanceFilter: 10))` emits on movement ≥ `distanceFilter` meters — the battery-friendly way to track. Cancel the subscription when done.
- **Accuracy vs battery**: higher accuracy + smaller distance filter = more GPS use = more drain/heat. Choose the **lowest accuracy that works**.

- **Geocoding** (`geocoding` package): `placemarkFromCoordinates(lat, lng)` → address; `locationFromAddress("...")` → coordinates. Network-dependent, rate-limited — cache.
- **Distance/bearing**: `Geolocator.distanceBetween(...)` for "how far."
- **Repository**: expose `Future<Position> current()` and `Stream<Position> track()`; centralize permission/service checks.

Memory Representation

A `Position` is a small value object. The stream subscription holds a native listener — cancel it to stop GPS.

Compiler Behavior

Not applicable; native bridge at runtime.

Runtime Behavior

`getCurrentPosition` may take seconds for a fresh fix (especially cold/high-accuracy). The stream keeps GPS active while listened — a live drain until cancelled.

Flutter Engine Behavior

Location updates arrive over a platform channel/ `EventChannel` (26 · event_channel) and surface as a Dart stream.

Dart VM Behavior

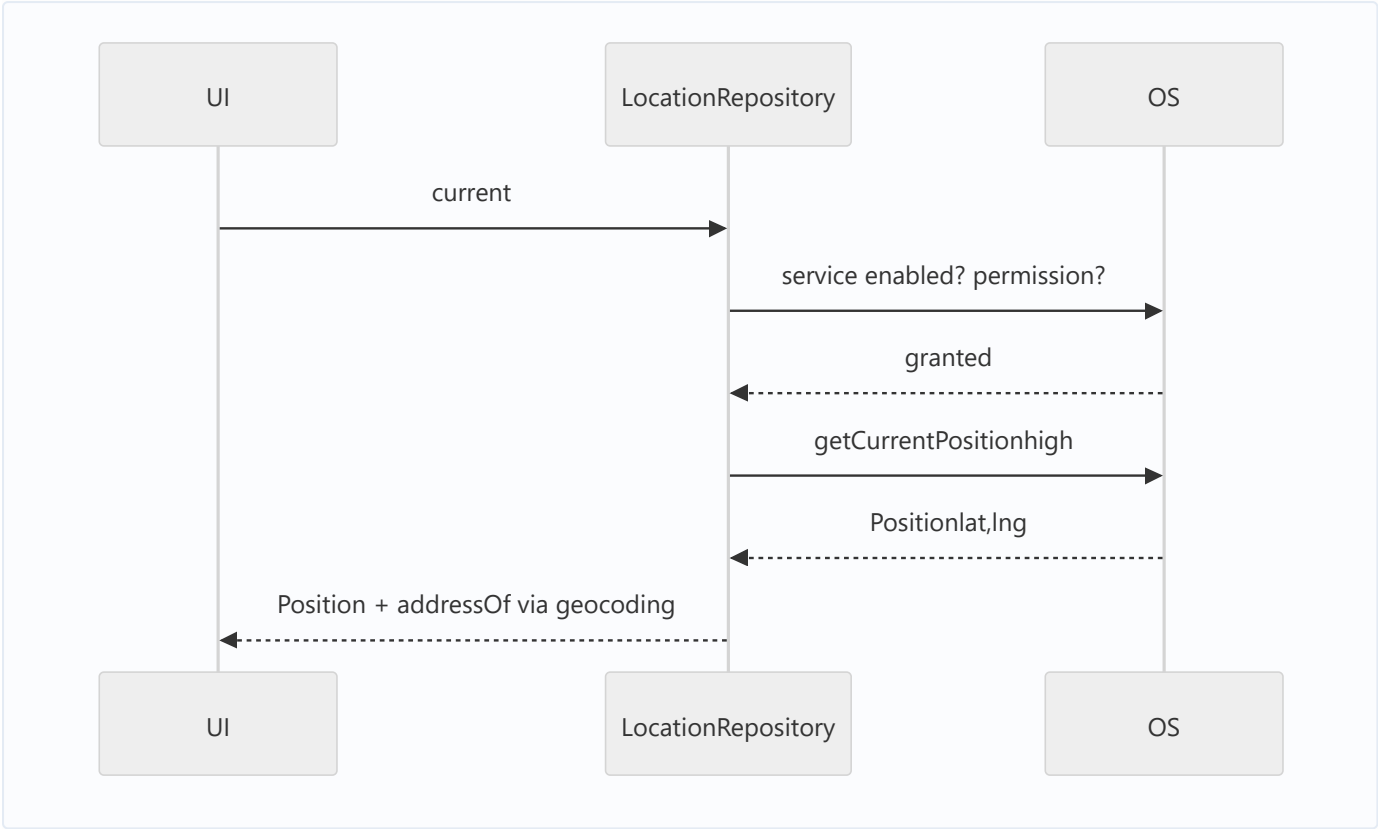
Not applicable; positions are marshalled from native.

Examples

```
import 'package:geolocator/geolocator.dart';
import 'package:geocoding/geocoding.dart';


class LocationRepository {
  // Preflight: service + permission (throws a domain error the UI can show)
  Future<void> _ensureReady() async {
    if (!await Geolocator.isLocationServiceEnabled()) {
      throw const LocationException('Location service is off');
    }
    var perm = await Geolocator.checkPermission();
```

```
    if (perm == LocationPermission.denied) perm = await Geolocator.requestPermission();
    if (perm == LocationPermission.denied) throw const LocationException('Permission denied');
    if (perm == LocationPermission.deniedForever) {
      await Geolocator.openAppSettings();
      throw const LocationException('Permission permanently denied');
    }
  }

  Future<Position> current() async {
    await _ensureReady();
    return Geolocator.getCurrentPosition(desiredAccuracy: LocationAccuracy.high);
  }

  // Battery-friendly tracking: only emit after moving >= 10 m
  Stream<Position> track() async* {
    await _ensureReady();
    yield* Geolocator.getPositionStream(
      locationSettings: const LocationSettings(
        accuracy: LocationAccuracy.high,
        distanceFilter: 10,
      ),
    );
  }

  Future<String> addressOf(Position p) async {
    final marks = await placemarkFromCoordinates(p.latitude, p.longitude);
    final m = marks.first;
    return '${m.street}, ${m.locality}, ${m.country}';
  }
}

class LocationException implements Exception {
  final String message;
  const LocationException(this.message);
  @override
  String toString() => message;
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Not checking service enabled	Hangs/errors when GPS off	`isLocationServiceEnabled()` first
Not handling `deniedForever`	User stuck	Route to `openAppSettings()`
Always max accuracy	Battery drain/heat	Lowest accuracy + distance filter that works
Not cancelling the stream	GPS runs forever	Cancel subscription on dispose
Requesting `always` when `whileInUse` suffices	Rejection/low acceptance	Request the minimum tier needed
Geocoding on every frame	Rate limits/latency	Cache, throttle

Best Practices

- **Preflight** service + permission (handle denied/`deniedForever` → Settings); request the **minimum tier** (`whileInUse` unless background truly needed).
- Choose the **lowest accuracy + largest distance filter** that works; **cancel** the position stream when done (battery).
- Use `getPositionStream` (not polling) for tracking; **cache/throttle geocoding** (network, rate-limited).

- Wrap in a **repository** exposing `current()` / `track()` / `addressOf()`; surface service/permission errors as domain errors (Module 38).

Performance

GPS is one of the biggest battery drains; accuracy and update frequency dominate cost. Use distance filters, stop streams promptly, and prefer coarse accuracy when precision isn't needed.

Advantages / Disadvantages

- **+** Precise location, movement streams, distance/geocoding; cross-platform.
- **−** Battery/heat cost, permission/service complexity (tiers, background), fix latency, geocoding is network/rate-limited.

Interview Questions

1. 🟢 **One-shot vs stream location?** — `getCurrentPosition` for a single fix; `getPositionStream` (with a distance filter) for continuous tracking that emits on movement.
2. 🟢 **What two preconditions must you check before reading location?** — The location **service is enabled** and **permission is granted** (request if denied; Settings if deniedForever).
3. 🟡 **How do you make location battery-friendly?** — Lower `desiredAccuracy`, use a `distanceFilter`, cancel streams when done, avoid `always` unless needed.
4. 🟡 **What's the difference between whileInUse and always?** — Foreground-only vs background location; `always` needs extra justification/keys and is scrutinized in review.
5. 🟡 **What is geocoding?** — Converting coordinates ↔ addresses (`placemarkFromCoordinates` / `locationFromAddress`); network-based and rate-limited, so cache.
6. 🔴 **Why cancel the position stream?** — An active stream keeps GPS running, draining battery — cancel on dispose.
7. 🔴 **How do you compute distance between two points?** — `Geolocator.distanceBetween(lat1, lng1, lat2, lng2)` (meters).

Senior Engineer Tips

- Always preflight service + permission and model the failure states as domain errors — half of location bugs are "GPS off" or "deniedForever" unhandled.
- Default to the coarsest accuracy the feature tolerates and a generous distance filter; escalate only when needed.
- Never leave a position stream running past the screen — tie its subscription to widget/bloc disposal.

Architect Perspective

Geolocation is a battery/permission/privacy-sensitive capability. A `LocationRepository` that centralizes preflight, exposes one-shot + stream APIs, enforces minimum accuracy/tier, and returns domain errors keeps location testable and power-efficient — feeding maps, geofencing, and offline tagging (Module 30, Module 19).

Summary

- `geolocator`: preflight service + permission → `getCurrentPosition` (one-shot) or `getPositionStream` (tracking, distance filter).
- Accuracy + frequency drive battery; request the minimum permission tier; cancel streams; cache geocoding.
- Wrap behind a repository with domain errors.

Revision Notes

- Preflight: `isLocationServiceEnabled()` + `checkPermission()` / `requestPermission()`; handle `deniedForever` → `openAppSettings()`.
- One-shot `getCurrentPosition(desiredAccuracy)`; stream `getPositionStream(LocationSettings(accuracy, distanceFilter))` — cancel it.
- iOS whileInUse vs always (background = justification); Android coarse vs fine; lowest accuracy that works.
- `geocoding`: coords ↔ address (cache); `distanceBetween` for distance; repository wrapper.

Practice Questions

1. What must be true before you can read the device location?
2. How do you track movement without draining the battery?
3. What is the whileInUse vs always distinction on iOS?

Coding Questions

1. Implement `LocationRepository.current()` with full preflight + domain errors.
2. Implement `track()` as a `distanceFilter` stream and consume it in a widget (cancel on dispose).
3. Add `addressOf(Position)` using geocoding with caching.

Mini Project

"Near me" + live tracking (Flutter): Build a `LocationRepository` with `current()`, `track()` (distance-filtered stream), and `addressOf()` (cached geocoding), full service/permission preflight, and domain errors. UI shows current address, a live-updating position while tracking (subscription cancelled on dispose), and graceful handling of service-off/denied. Acceptance: preflight + all permission states handled; one-shot + stream work; stream cancelled; geocoding cached; battery-conscious accuracy/filter; behind a repository; runs on device.

Sensors, Battery & Device Info

> Read motion sensors with `sensors_plus` (accelerometer/gyroscope/magnetometer as **streams**), battery level/state with `battery_plus`, and static hardware/OS details with `device_info_plus` (model, OS version, unique-ish id) — subscribe only while needed (sensors are high-frequency and battery-hungry) and expose everything behind a repository.

Introduction

Beyond camera/location, apps read the device's **motion sensors** (steps, shake-to-undo, AR/tilt UIs), **battery** (defer heavy work on low battery), and **device info** (adapt to model/OS, analytics, feature gating). This file covers the streaming sensors, the battery API, device-info reads, and the battery/lifecycle discipline they demand.

Why this concept exists

These are native hardware/OS signals with no Dart equivalent; plugins bridge them via platform channels (Module 26). Sensors emit at high frequency (tens–hundreds Hz) so must be subscribed sparingly; battery/device info let apps adapt behavior and diagnose issues across the fragmented device landscape.

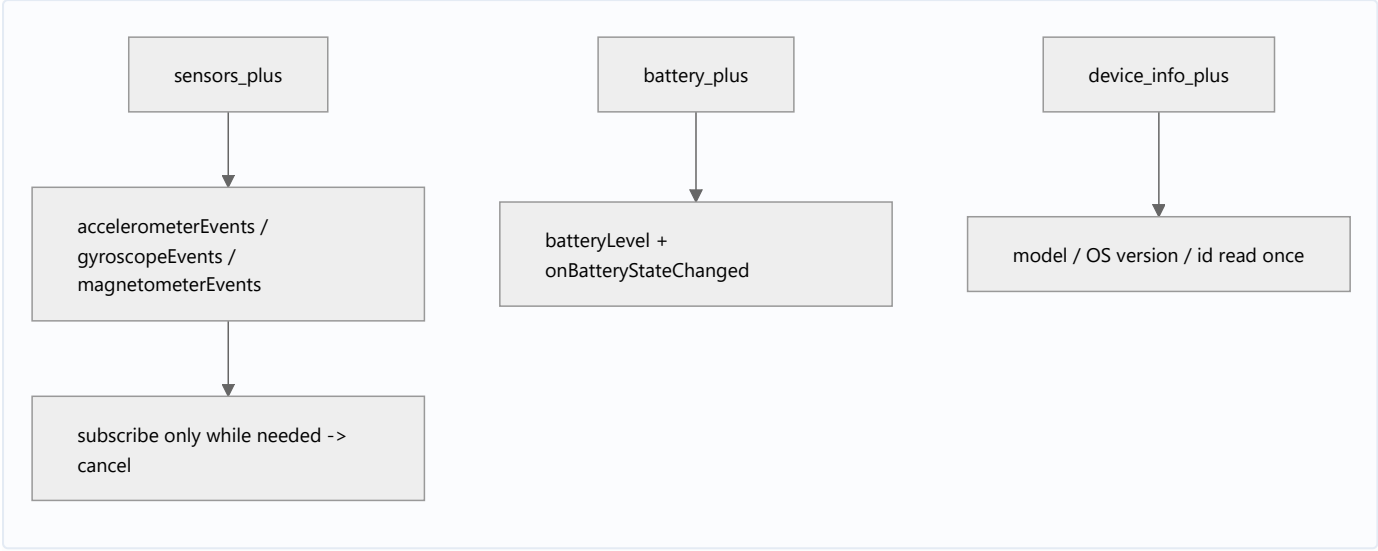
Real-world analogy

Sensors are a **firehose of readings** — you only open the valve while you're actually drinking (subscribed), because leaving it on floods you (battery). Battery info is the **fuel gauge** (throttle heavy work when low). Device info is the **vehicle's spec sheet** (model/year/OS) you read once to adapt.

Problem Statement

Add shake-to-undo (accelerometer), defer a heavy sync when the battery is low/unplugged (`battery_plus`), and log device model + OS version for analytics/crash context (`device_info_plus`). You'll subscribe to a sensor stream (and cancel it), read battery state, and read device info — behind repositories.

Internal Working



- `sensors_plus`: exposes `accelerometerEventStream()`, `gyroscopeEventStream()`, `magnetometerEventStream()`, `userAccelerometerEventStream()` (gravity removed). High-frequency — subscribe **only while the feature is active**, cancel on dispose, and consider throttling/debouncing.
- `battery_plus`: `battery.batteryLevel` (0–100) and `battery.onBatteryStateChanged` (charging/discharging/full) — use to defer heavy work (sync/backup) when low/unplugged.
- `device_info_plus`: `DeviceInfoPlugin().androidInfo` / `.iosInfo` → model, manufacturer, OS version, identifiers. **Static** — read once and cache. Note: device ids aren't stable/unique across reinstalls and are privacy-sensitive; don't use as a security identity.
- **Discipline**: sensor streams are the main battery risk here — treat them like GPS (subscribe/cancel tightly). Battery/device info are cheap.
- **Repository**: `MotionRepository.shakes()`, `PowerRepository.state()`, `DeviceRepository.info()`.

Memory Representation

Sensor events are tiny value objects but arrive rapidly (GC pressure if you allocate per event). A subscription holds a native listener until cancelled.

Compiler Behavior

Not applicable; native bridges at runtime.

Runtime Behavior

An active sensor stream keeps the sensor powered and delivers events continuously — a live battery cost. Battery-state changes emit on transitions; device info is a one-time native read.

Flutter Engine Behavior

Sensor/battery events arrive via `EventChannel`s (26 · event_channel) as Dart streams; device info via a `MethodChannel`.

Dart VM Behavior

Not applicable; values marshalled from native. High-frequency streams can create allocation churn — throttle.

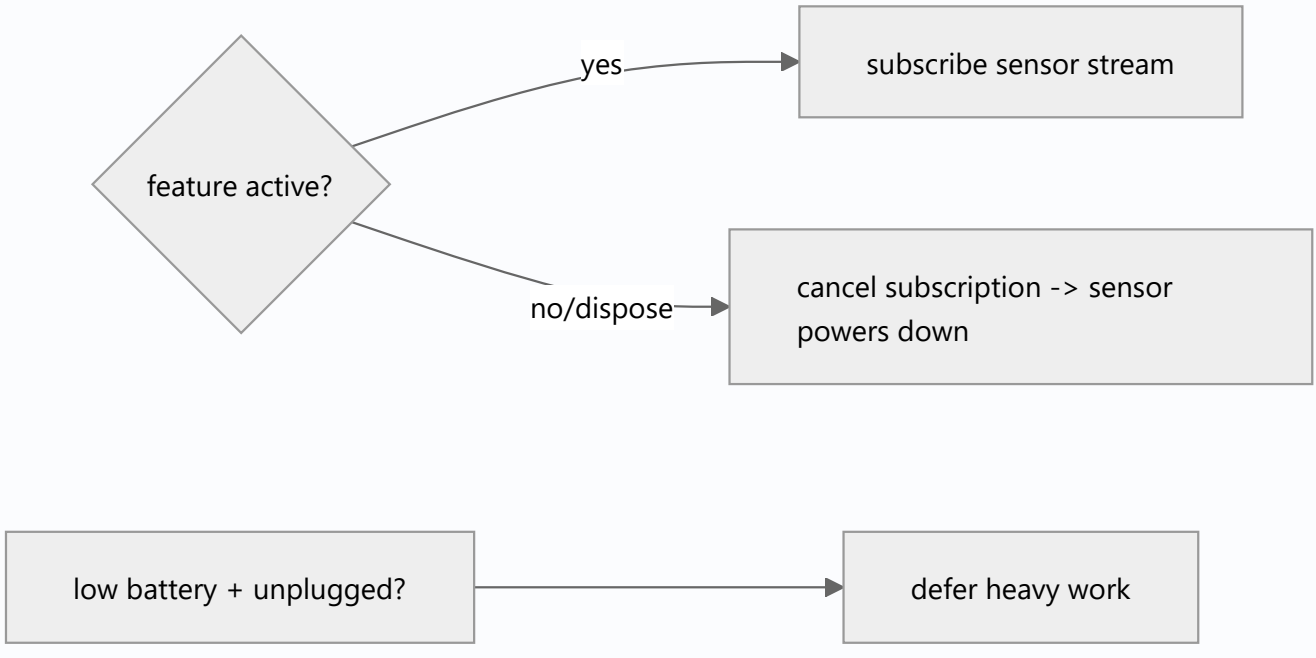
Examples

```
import 'package:sensors_plus/sensors_plus.dart';
import 'dart:math';

// Shake detection from the accelerometer stream (subscribe only while active)
class MotionRepository {
  Stream<void> shakes({double threshold = 20}) {
    return accelerometerEventStream()
        .where((e) => sqrt(e.x * e.x + e.y * e.y + e.z * e.z) > threshold)
        .map((_) {});
  }
}
// Usage: final sub = motion.shakes().listen((_) => undo());  ... later: sub.cancel();
```

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

class PowerRepository {
  final _battery = Battery();
  Future<bool> shouldDeferHeavyWork() async {
    final level = await _battery.batteryLevel;
    final state = await _battery.batteryState;
    return level < 20 && state != BatteryState.charging; // defer sync/backup
  }
  Stream<BatteryState> stateChanges() => _battery.onBatteryStateChanged;
}
```

```
import 'package:device_info_plus/device_info_plus.dart';
import 'dart:io';

class DeviceRepository {
  Future<String> summary() async {
    final plugin = DeviceInfoPlugin();
    if (Platform.isAndroid) {
      final a = await plugin.androidInfo;
      return '${a.manufacturer} ${a.model} · Android ${a.version.release}';
    } else {
      final i = await plugin.iosInfo;
      return '${i.name} · iOS ${i.systemVersion}';
    }
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Leaving a sensor stream subscribed	Constant battery drain	Subscribe only while active; cancel on dispose
Heavy work per sensor event	Jank/GC churn	Throttle/debounce; keep handlers tiny
Reading device info repeatedly	Wasteful	Read once, cache
Using device id as security identity	Not stable/unique; privacy	Use proper auth; treat id as best-effort
Ignoring battery state for heavy jobs	Drains user's battery	Defer sync/backup on low/unplugged
Not handling platform differences	Field mismatch	Branch Android/iOS info

Best Practices

- Treat sensor streams like GPS: **subscribe only while the feature is active, cancel on dispose**, and **throttle** high-frequency events.
- Use `userAccelerometerEventStream` when you want motion without gravity; keep event handlers cheap.
- **Read device info once and cache**; branch Android/iOS; don't rely on device ids for identity/security (privacy + instability).
- Use **battery state** to defer heavy/background work (sync, backup) when low/unplugged; wrap each in a **repository**.

Performance

Sensor streams are the dominant cost — high frequency + continuous power. Cancel promptly and throttle. Battery/device info reads are negligible.

Advantages / Disadvantages

- **+** Rich device awareness (motion, power, hardware) for adaptive/interactive features and diagnostics; cross-platform.
- **−** Sensor streams drain battery + create allocation churn; device ids unstable/privacy-sensitive; platform-specific fields.

Interview Questions

1. 🟢 **How are motion sensors exposed in Flutter?** — As high-frequency streams (`accelerometerEventStream`, `gyroscopeEventStream`, etc.) via `sensors_plus`.
2. 🟢 **Why must you cancel sensor subscriptions?** — An active stream keeps the sensor powered and delivers events continuously — a battery drain; cancel on dispose.

3. 🟡 **accelerometer vs userAccelerometer?** — The former includes gravity; `userAccelerometer` removes it (pure user motion).
4. 🟡 **How do you use battery state?** — Read `batteryLevel` / `batteryState` (or listen to changes) to defer heavy work when low/unplugged.
5. 🟡 **Is a device id a good user identifier?** — No — it's not stable across reinstalls and is privacy-sensitive; use real auth.
6. 🔴 **How do you keep high-frequency sensor handling smooth?** — Throttle/debounce, keep handlers tiny, avoid per-event allocations/setState storms.
7. 🔴 **Why read device info once?** — It's static; repeated native reads are wasteful — cache the result.

Senior Engineer Tips

- Tie every sensor subscription to a lifecycle (bloc/state dispose); a forgotten accelerometer listener is a silent battery bug.
- Throttle sensor streams (e.g., sample every N ms) before doing work — raw streams are far faster than your UI needs.
- Cache device info in a singleton and attach it to logs/crash reports for field diagnostics.

Architect Perspective

Sensors/battery/device-info are adaptive-behavior and diagnostics signals. Repositories that expose throttled sensor streams, power-aware scheduling hooks, and cached device info let the app react to motion, respect battery, and adapt across a fragmented device base — feeding background scheduling, analytics, and crash reporting (Module 33, Module 52).

Summary

- `sensors_plus` = high-frequency motion streams (subscribe/cancel tightly, throttle); `battery_plus` = level/state (defer heavy work); `device_info_plus` = static specs (read once, cache).
- Sensor streams are the battery risk; device ids aren't identities.
- Wrap each behind a repository.

Revision Notes

- Sensors: `accelerometerEventStream` / `gyroscope` / `magnetometer` / `userAccelerometer`; subscribe while active, cancel on dispose, throttle.
- Battery: `batteryLevel` (0–100), `batteryState`, `onBatteryStateChanged`; defer heavy work when low/unplugged.

- Device info: `androidInfo` / `iosInfo` (model/OS/id) — static, cache; id ≠ identity (privacy/instability).
- Repositories: `MotionRepository` / `PowerRepository` / `DeviceRepository`.

Practice Questions

1. Why are sensor streams a battery concern and how do you mitigate it?
2. When should the app defer heavy background work based on battery?
3. Why shouldn't you use a device id as a user identifier?

Coding Questions

1. Implement shake detection from the accelerometer stream and cancel it on dispose.
2. Implement `PowerRepository.shouldDeferHeavyWork()`.
3. Build a cached `DeviceRepository.summary()` branching Android/iOS.

Mini Project

Device-aware utilities (Flutter): Build `MotionRepository` (shake-to-undo from a throttled accelerometer stream, cancelled on dispose), `PowerRepository` (defer a simulated heavy sync when battery < 20% and unplugged), and a cached `DeviceRepository` that logs model + OS. Acceptance: shake triggers undo; sensor subscription cancelled; heavy work deferred on low battery; device info read once + cached + logged; each behind a repository; runs on device.

Connectivity (Network Reachability & Offline Handling)

> Detect the network *interface* (wifi/mobile/none) with `connectivity_plus` and, crucially, verify **actual internet reachability** separately — "connected to wifi" ≠ "has internet." React to online/offline transitions via a stream, show an offline banner, pause/resume network work, and feed the offline-first sync engine (Module 19).

Introduction

Apps must behave well offline: queue writes, pause polling, show a banner, retry on reconnect. `connectivity_plus` reports the active interface and emits changes; but a connected interface can still lack real internet (captive portal, no signal). This file covers reachability vs connectivity, the change stream, and wiring it into app behavior.

Why this concept exists

Mobile networks are unreliable — connections drop, switch (wifi↔mobile), and lie (connected but no internet). The OS exposes interface state via platform channels (Module 26); Flutter apps need this to drive offline UX and sync. It's the trigger layer for offline-first (Module 19).

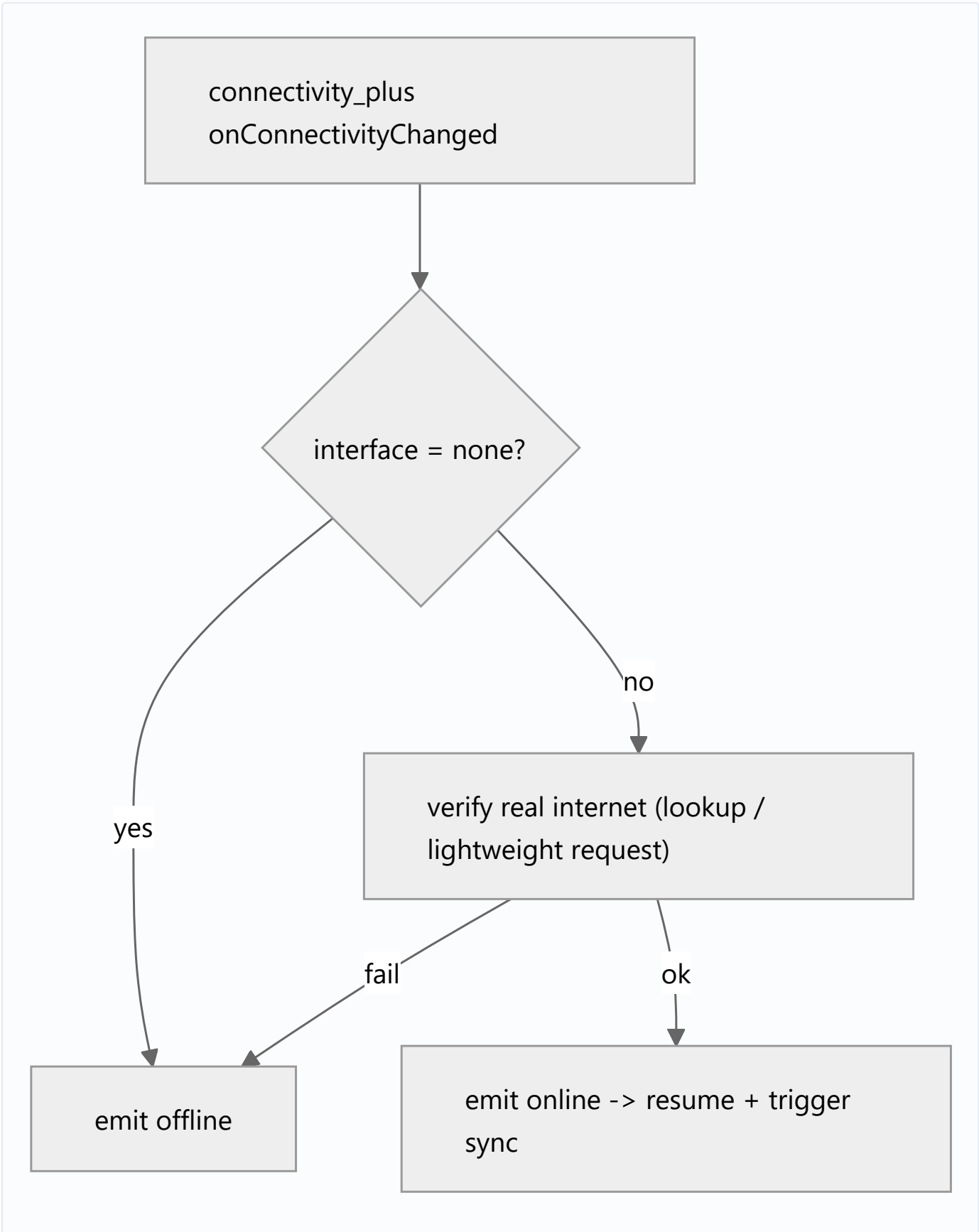
Real-world analogy

`connectivity_plus` tells you **the tap is connected to a pipe** (wifi/mobile/none). But a connected pipe can still be **dry** (captive portal, dead cell). So you also **run the tap** (a real request/lookup) to confirm water actually flows (internet reachable).

Problem Statement

Show an offline banner, pause a live feed and outbound requests when offline, and resume + trigger sync when the connection (with real internet) returns — reliably, without treating "on wifi" as "online." You'll use `connectivity_plus` for changes plus a reachability check, behind a repository exposing a boolean `isOnline` stream.

Internal Working



- `connectivity_plus`: `checkConnectivity()` → current interface(s) (`wifi` / `mobile` / `ethernet` / `vpn` / `none`) and `onConnectivityChanged` stream. **It reports the interface, not internet** — a key gotcha.

- **Reachability**: confirm real internet with a lightweight check (`InternetAddress.lookup('example.com')` or a HEAD request to your API/health endpoint). Combine: online = interface ≠ none **and** reachability ok. (The `internet_connection_checker` package packages this.)
- **Debounce/dedupe**: connectivity can flap; debounce transitions and only emit real changes (distinct) to avoid banner flicker/sync storms.
- **React**: on offline → show banner, pause polling/uploads, queue writes; on online → hide banner, resume, **trigger sync** (Module 19).
- **Don't gate every request on it**: still handle request failures directly (the check can be stale); use connectivity as a *hint/trigger*, not a guarantee.
- **Repository**: expose `Stream<bool> isOnline` + `Future<bool> checkNow()`.

Memory Representation

Trivial; a stream subscription + last-known state. Reachability checks are transient network calls.

Compiler Behavior

Not applicable.

Runtime Behavior

The stream emits on interface changes; reachability adds a quick async check. Connectivity may flap during transitions (handoff wifi↔mobile) — debounce.

Flutter Engine Behavior

Connectivity events arrive via an `EventChannel` (26 · event_channel) as a Dart stream.

Dart VM Behavior

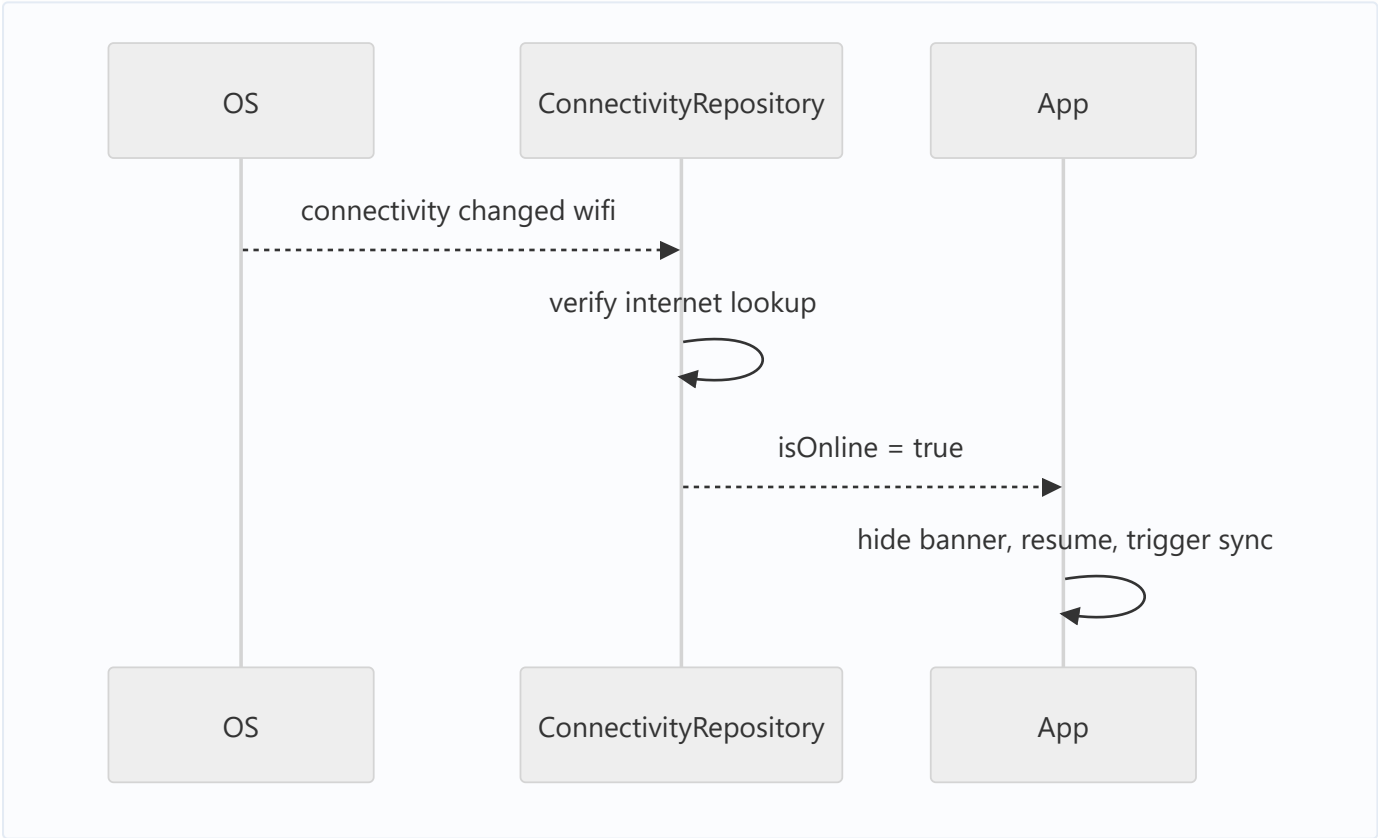
Not applicable.

Examples

```
import 'package:connectivity_plus/connectivity_plus.dart';
import 'dart:io';
import 'dart:async';

class ConnectivityRepository {
  final _connectivity = Connectivity();

  // interface change -> verify real internet -> distinct online/offline stream
  Stream<bool> get isOnline async* {
    yield await checkNow();
    await for (final _ in _connectivity.onConnectivityChanged) {
      yield await checkNow();
    }
  }

  Future<bool> checkNow() async {
    final result = await _connectivity.checkConnectivity();
    if (result.contains(ConnectivityResult.none)) return false;
    return _hasInternet(); // interface up != internet up
  }

  Future<bool> _hasInternet() async {
    try {
      final r = await InternetAddress.lookup('example.com')
          .timeout(const Duration(seconds: 3));
      return r.isNotEmpty && r.first.rawAddress.isNotEmpty;
    } catch (_) {
      return false;
    }
  }
}
// Use .distinct() when consuming to avoid duplicate emissions / banner flicker.
```

```
// Offline banner driven by the stream
StreamBuilder<bool>(
  stream: connectivity.isOnline, // .distinct() applied in repo/consumer
  builder: (context, snap) {
    final online = snap.data ?? true;
    return online
        ? const SizedBox.shrink()
        : Container(
            color: Colors.red, padding: const EdgeInsets.all(8),
            child: const Text('You are offline', textAlign: TextAlign.center),
          );
  },
);
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Treating interface = internet	Captive portal / no signal	Verify reachability separately
No debounce/distinct	Banner flicker, sync storms	Debounce + `.distinct()`
Gating every request on the check	Stale, races	Use as hint/trigger; handle request errors too
Not triggering sync on reconnect	Stale offline data	Fire sync on online transition (Module 19)
Blocking UI on reachability check	Jank	Async + timeout
Assuming online at startup	Wrong initial UI	Emit initial `checkNow()`

Best Practices

- Distinguish **interface (connectivity_plus)** from **real internet (reachability check)**; combine both for `isOnline`.
- **Debounce +** `.distinct()` transitions; emit an **initial** state at startup.
- Use connectivity as a **trigger/hint** (banner, pause/resume, fire sync) — still handle per-request failures directly.
- On reconnect, **resume work + trigger the sync engine**; wrap in a **repository** exposing `Stream<bool> isOnline`.

Performance

Negligible; a stream + occasional short reachability checks (with timeouts). Debouncing avoids redundant work.

Advantages / Disadvantages

- **+** Reactive offline UX, sync triggering, pause/resume network work; cross-platform.
- **−** Interface≠internet gotcha, flapping needs debounce, reachability adds a network call, only a hint (not a guarantee).

Interview Questions

1. 🟢 **What does `connectivity_plus` actually report?** — The active network **interface** (wifi/mobile/none), *not* whether the internet is reachable.
2. 🟢 **How do you know the device truly has internet?** — Do a reachability check (DNS lookup / lightweight request) in addition to the interface state.
3. 🟡 **Why debounce/ `.distinct()` connectivity events?** — Connections flap during handoffs; without it you get banner flicker and repeated sync triggers.

4. 🟡 **Should you gate every network request on connectivity?** — No — use it as a hint/trigger; the state can be stale, so still handle request errors directly.
5. 🟡 **What should happen on reconnect?** — Hide the banner, resume paused work, and trigger the offline sync engine.
6. 🔴 **How does this integrate with offline-first?** — It's the trigger: offline → queue writes (outbox); online → flush/sync (Module 19).
7. 🔴 **How do you avoid blocking the UI on the reachability check?** — Run it async with a timeout; emit the result to a stream.

Senior Engineer Tips

- Never equate "on wifi" with "online" — captive portals and dead cells are the #1 false-positive; always add a reachability probe with a timeout.
- Debounce transitions and expose a single `.distinct()` `Stream<bool>` so the whole app shares one source of truth.
- Treat connectivity as the sync *trigger*, not the sync *gate* — requests must still handle their own failures.

Architect Perspective

Connectivity is the reactive trigger layer for resilient apps: it drives offline UX and kicks the sync engine. A `ConnectivityRepository` combining interface + reachability into one debounced `Stream<bool>` gives the app a single, reliable online signal — the backbone of offline-first behavior (Module 19, Module 16).

Summary

- `connectivity_plus` reports the interface; verify real internet with a reachability check — combine into `isOnline`.
- Debounce + `.distinct()` + initial emission; use as a trigger (banner, pause/resume, sync), not a hard gate.
- Wrap in a repository exposing `Stream<bool> isOnline`; drives offline-first.

Revision Notes

- Interface (`checkConnectivity` / `onConnectivityChanged`) ≠ internet → add reachability (`InternetAddress.lookup` /HEAD, with timeout).
- `isOnline` = interface≠none AND reachable; debounce + `.distinct()` + initial state.
- Trigger, not gate: banner, pause/resume, fire sync on reconnect; still handle request errors.
- Repository: `Stream<bool> isOnline` + `checkNow()`; feeds Module 19.

Practice Questions

1. Why is "connected to wifi" not the same as "online"?
2. Why debounce connectivity transitions?
3. What should the app do the moment connectivity returns?

Coding Questions

1. Implement `ConnectivityRepository.isOnline` combining interface + reachability.
2. Build an offline banner driven by that stream (distinct).
3. Trigger a sync callback on the offline→online transition.

Mini Project

Offline-aware shell (Flutter): Build a `ConnectivityRepository` exposing a debounced, `.distinct()` `Stream<bool> isOnline` (interface + reachability with timeout). Wire an app-wide offline banner, pause a simulated live feed when offline, and trigger a `sync()` callback on reconnect. Acceptance: distinguishes interface vs internet; initial state emitted; no banner flicker; pauses/resumes + triggers sync on reconnect; behind a repository; runs on device.

Device Integration (Capstone: Features Behind Repositories)

> The unifying pattern for every device feature: wrap each native capability (camera, location, sensors, connectivity, battery, device info) behind a **repository** with a small domain API, **centralize permissions**, prefer **streams** for continuous data (cancel on dispose), and **degrade gracefully** when denied/unavailable — so the UI and tests never touch a plugin and the app stays portable, testable, and battery-friendly.

Introduction

This module capstone ties the individual device features together into one architecture. Rather than sprinkling plugin calls across widgets, you expose each capability through a repository interface, inject it (DI — Module 14), and compose them in a feature. This file shows the pattern, a combined "photo + location + connectivity" example, and the cross-cutting concerns (permissions, lifecycle, errors, testing).

Why this concept exists

Device plugins are imperative, permission-gated, lifecycle-sensitive, and platform-specific. Calling them directly from widgets makes code untestable (needs a real device), brittle (permission/lifecycle bugs), and hard to swap. A repository layer isolates these concerns behind a clean, mockable boundary — the same discipline used for networking/storage (Module 16, Module 15).

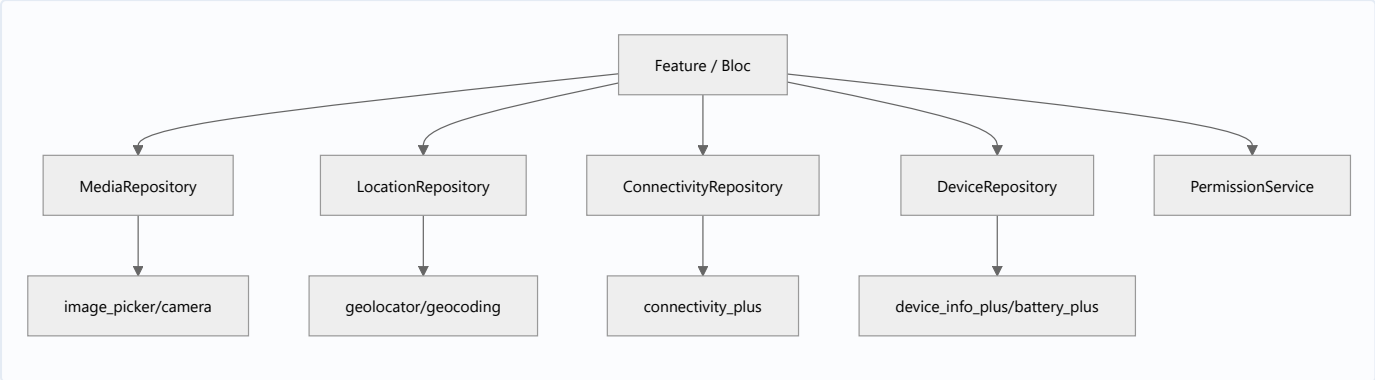
Real-world analogy

The repository is a **universal remote** for the device: the app presses simple buttons ("get photo", "where am I", "am I online") without knowing the make/model of each gadget (plugin) behind it. Swap a gadget (change plugin) or use a dummy one (mock in tests) — the remote's buttons don't change.

Problem Statement

Build an "add place" feature: attach a photo (camera/gallery), tag it with the current location + address, show an online/offline banner, and record device/battery context — all composed from device repositories, permissions handled once, degrading gracefully, and unit-testable without a real device. You'll compose the repositories from this module.

Internal Working



- **Interface per capability**: define abstract repositories (`MediaRepository` , `LocationRepository` , `ConnectivityRepository` , `DeviceRepository`) with domain methods returning domain types/ `Result` / `Either` — not raw plugin types.
- **Centralized permissions**: a `PermissionService` (27/28) all repositories use; request in context, handle denied/restricted → Settings.
- **Streams for continuous data**: location tracking, sensors, connectivity are streams — expose them, and **cancel subscriptions** on dispose (bloc/state).
- **Graceful degradation**: every feature has a no-permission/unavailable path (skip location tag, hide the sensor feature, offline banner). Never crash on denial.
- **DI + testing**: inject repositories (Module 14); in tests provide fakes returning canned positions/files/online-state — no device needed (Module 49).
- **Errors**: convert plugin/permission failures into domain errors (Module 38).

Memory Representation

Repositories are thin; the cost is in underlying resources (camera controller, GPS/sensor subscriptions). The repository layer is where you enforce disposal/cancellation.

Compiler Behavior

Abstract interfaces let you compile against contracts; concrete plugin impls are injected — supports swapping/mocking.

Runtime Behavior

The feature orchestrates async calls + stream subscriptions; repositories manage native resources; degradation paths run when capabilities are denied/unavailable.

Flutter Engine Behavior

Underlying plugins use channels/textures/platform views as covered in their files; the repository layer adds no engine cost.

Dart VM Behavior

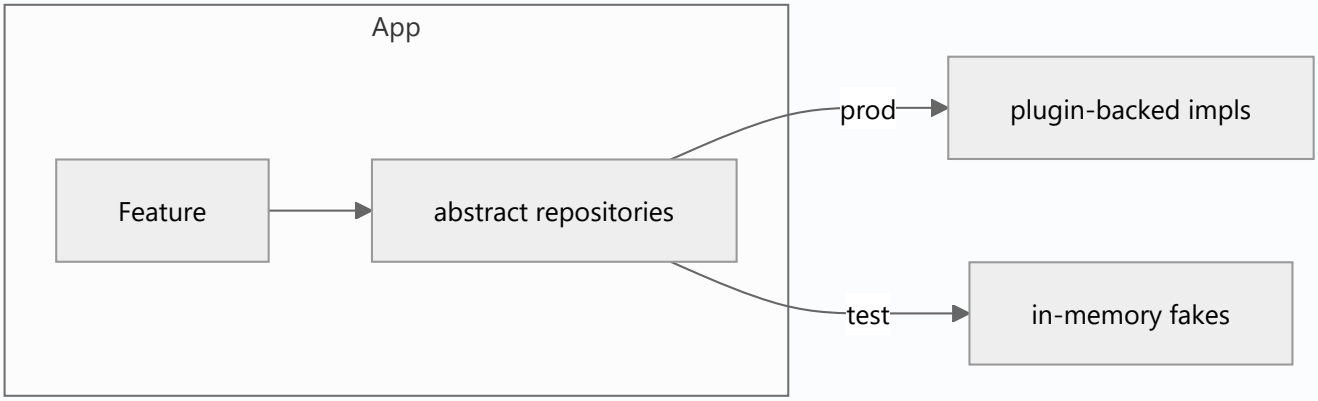
Not applicable beyond the individual features (stream allocation churn — throttle sensors).

Examples

```
// Domain interfaces (implementations wrap the plugins from this module)
abstract class MediaRepository { Future<File?> pickImage({required bool fromCamera}); }
abstract class LocationRepository {
  Future<Position> current();
  Future<String> addressOf(Position p);
}
abstract class ConnectivityRepository { Stream<bool> get isOnline; }

// Feature composing them — no plugin references, fully testable
class AddPlaceController {
  final MediaRepository media;
  final LocationRepository location;
  final PermissionService permissions;
  AddPlaceController(this.media, this.location, this.permissions);

  Future<PlaceDraft> capture() async {
    final photo = await media.pickImage(fromCamera: true); // null if cancelled
    Position? pos;
    String? address;
    try {
      pos = await location.current();           // may throw LocationException
      address = await location.addressOf(pos);
    } on LocationException {
      // graceful: save the place without a location tag
    }
    return PlaceDraft(photo: photo, position: pos, address: address);
  }
}
```

```
// Test with fakes — no real device
class FakeLocationRepository implements LocationRepository {
  @override
  Future<Position> current() async => Position(/* canned lat/lng ... */);
  @override
  Future<String> addressOf(Position p) async => '1 Test St, Testville';
}
// Inject FakeLocationRepository + a stub MediaRepository -> assert PlaceDraft.
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Calling plugins from widgets	Untestable, brittle	Repository layer + DI
Permission logic scattered	Inconsistent/duplicated	One `PermissionService`
No degradation path	Crashes on denial/unavailable	Handle every denied/unavailable case
Leaking stream subscriptions	Battery drain	Cancel on dispose (bloc/state)
Returning raw plugin types	Leaks plugin into domain	Return domain types/ `Result`
Untestable device code	Needs real hardware	Fakes behind interfaces

Best Practices

- One **repository interface per capability** returning **domain types**; inject via **DI**; keep plugins out of the UI.
- Centralize **permissions** in one service (in-context, Settings fallback); convert failures to **domain errors**.

- Prefer **streams** for continuous data and **cancel** them on dispose; enforce native-resource disposal in the repository layer.
- Provide **graceful degradation** for every capability; write **fakes** for tests (no device needed).

Performance

The pattern adds no overhead; it's where you *enforce* battery discipline (cancel streams, release controllers, defer heavy work on low battery). Underlying feature costs (GPS/sensors/preview) dominate — manage them here.

Advantages / Disadvantages

- **+** Testable (fakes), swappable (plugin-agnostic), consistent permissions/lifecycle, graceful degradation, single place to enforce battery discipline.
- **−** More boilerplate (interfaces/impls), indirection, requires DI discipline.

Interview Questions

1. 🟢 **Why wrap device plugins in repositories?** — To keep plugins out of the UI, make features testable with fakes, allow swapping plugins, and centralize permissions/lifecycle/errors.
2. 🟢 **How do you test code that uses the camera/GPS?** — Inject fake repositories returning canned files/positions — no real device required.
3. 🟡 **Where does permission logic live?** — In a single `PermissionService` used by all repositories (in-context requests, Settings fallback), not scattered in widgets.
4. 🟡 **How do you prevent battery drain in this architecture?** — Expose continuous data as streams and cancel subscriptions on dispose; release native resources in the repository; defer heavy work on low battery.
5. 🟡 **What should repositories return?** — Domain types/ `Result` / `Either`, not raw plugin types — so the plugin never leaks into the domain/UI.
6. 🔴 **How do you handle a denied permission or unavailable sensor?** — Every capability has a graceful-degradation path (skip the feature/tag, show a banner) and a domain error — never crash.
7. 🔴 **How does this connect to offline-first?** — The `ConnectivityRepository` triggers sync; media/location feed queued writes (outbox) that flush on reconnect (Module 19).

Senior Engineer Tips

- Design the interface around the *feature's need* ("tag with current place"), not the plugin's API — that's what makes it swappable and testable.

- Put all disposal/cancellation in the repository or bloc `close()`; a forgotten subscription is the classic device-feature battery bug.
- Ship fakes alongside real impls from day one — device features are otherwise impossible to test in CI.

Architect Perspective

Device integration is where clean architecture meets hardware: capabilities behind interfaces, permissions centralized, streams lifecycle-managed, failures degraded — the same boundary discipline as data/networking. This keeps a device-heavy app testable in CI, portable across plugins/platforms, and power-efficient, and it plugs directly into DI, offline-first, background work, and monitoring (Module 14, Module 19, Module 40).

Summary

- Wrap each device capability behind a repository interface returning domain types; inject via DI; keep plugins out of the UI.
- Centralize permissions, prefer streams (cancel on dispose), degrade gracefully, convert failures to domain errors.
- Provide fakes for CI testing; enforce battery discipline in the repository layer.

Revision Notes

- Interface per capability (Media/Location/Connectivity/Device) → domain types; DI-injected; UI never touches plugins.
- One `PermissionService`; streams for continuous data + cancel on dispose; release native resources; graceful degradation everywhere.
- Domain errors (Module 38); fakes for tests (Module 49); connectivity triggers sync (Module 19).

Practice Questions

1. Why keep device plugins out of widgets?
2. How do you unit-test a feature that uses camera + GPS?
3. Where do you enforce stream cancellation and resource disposal?

Coding Questions

1. Define `MediaRepository` / `LocationRepository` / `ConnectivityRepository` interfaces + plugin-backed impls.

2. Build a feature controller composing them with graceful degradation.
3. Write a unit test using fakes (no device).

Mini Project

"Add place" device slice (capstone — Flutter): Compose `MediaRepository`, `LocationRepository`, `ConnectivityRepository`, and `DeviceRepository` (from this module) behind interfaces + DI to build an "add place" feature: attach a photo, tag it with current location + address (degrade if denied), show an online/offline banner, and record device/battery context. Provide fakes and a unit test covering the no-location path. Acceptance: features composed from repositories (no plugin in UI); permissions centralized; streams cancelled on dispose; graceful degradation for every capability; unit test passes without a device; runs on a real device end-to-end.