33 · Background Services

Flutter Practice Notes

Contents

33 · Background Services

Introduction

This module covers running work when your app **isn't in the foreground**: the **background execution model** (why the OS restricts it, background isolates, platform differences), `workmanager` for deferrable/periodic tasks with constraints, **Android foreground services** for user-visible ongoing work (location tracking, playback) with a persistent notification, **iOS background execution** (`BGTaskScheduler` , background fetch/processing, silent push — all strictly metered), and a capstone tying them behind a service. It builds on isolates (Module 02), notifications (Module 32), and native setup (27/28).

Why this module exists

Sync, uploads, geofencing, and periodic refresh need to run without the UI open — but mobile OSes **aggressively restrict background work to protect battery**. The rules differ sharply by platform (Android WorkManager/foreground services vs iOS's tightly-metered `BGTaskScheduler`), tasks run in a **separate isolate** with no UI/state access, and OEM battery optimizations make timing unreliable. Knowing what's *possible*, *guaranteed*, and *forbidden* is essential to avoid building features the OS will silently kill.

Module map

#	File	Topic	Level
1	01_background_execution_model.md	Why background is restricted, background isolates, platform model	🔴
2	02_workmanager_and_periodic_tasks.md	`workmanager` : one-off/periodic tasks, constraints, retries	🔴
3	03_foreground_services_android.md	Android foreground services, persistent notification, live tracking	🔴
4	04_ios_background_execution.md	`BGTaskScheduler` , background fetch/processing, silent push, limits	🔴
5	05_background_integration.md	Capstone: background sync behind a service, cross-platform strategy	🔴

> **Cross-references:** Isolates: 02 · isolates. Notifications (required for foreground services): Module 32. iOS capabilities/background modes: 28 · ios_integration. Android permissions/manifest: 27 · permissions. Offline-first sync (the main use case): Module 19. Location: 29 · geolocation.

Prerequisites

02 Advanced Dart (isolates), 32 Notifications, 27/28 (native config, iOS background modes), 19 Offline First (sync).

What you'll be able to do after this module

- Explain the OS background-execution model and its isolate/battery constraints.
- Schedule deferrable and periodic background tasks with constraints via `workmanager`.
- Run an Android foreground service with a persistent notification for ongoing work.
- Use iOS `BGTaskScheduler`/background fetch/silent push within Apple's limits.
- Design a cross-platform background-sync strategy behind a service, degrading gracefully.

Capstone

Background sync slice: Periodic background sync (WorkManager on Android / `BGTaskScheduler` on iOS) that flushes an offline outbox (Module 19) when connected + charging, an Android foreground service for a live-tracking session, and silent-push-triggered refresh — all behind a `BackgroundService` with graceful degradation to foreground sync.

Summary

Background work is OS-restricted and platform-divergent: Android uses WorkManager (deferrable/periodic) + foreground services (ongoing/visible); iOS uses tightly-metered `BGTaskScheduler`/fetch/silent-push. Tasks run in a separate isolate, timing isn't guaranteed, and continuous work needs justification. Design for "best-effort, catch up in foreground," behind a cross-platform service.

The Background Execution Model (Why It's Restricted)

> Mobile OSes treat background execution as a **scarce, revocable privilege**, not a right: to protect battery/data/privacy they suspend apps aggressively, meter background time, and kill misbehavers. Background work runs in a **separate Dart isolate** with its own memory (no UI, no app singletons/state), timing is **best-effort** (not guaranteed), and the platforms diverge sharply — so the correct mental model is *"schedule deferrable work, keep it short, and catch up in the foreground when it inevitably doesn't run on time."*

Introduction

Before any plugin, you must understand *why* background work is hard and what's even possible. This file covers the OS rationale (battery/doze/app standby), the background-isolate model (and its no-shared-state constraint), the spectrum from deferrable to continuous work, and the Android/iOS divergence — the foundation for every later file.

Why this concept exists

Early mobile apps drained batteries with unrestricted background activity. OSes responded with **Doze/App Standby (Android)** and **strict background budgets (iOS)**: apps are suspended when not in use, background execution is batched/metered, and continuous work requires explicit justification (foreground service / background mode). This model exists to make phones last a day.

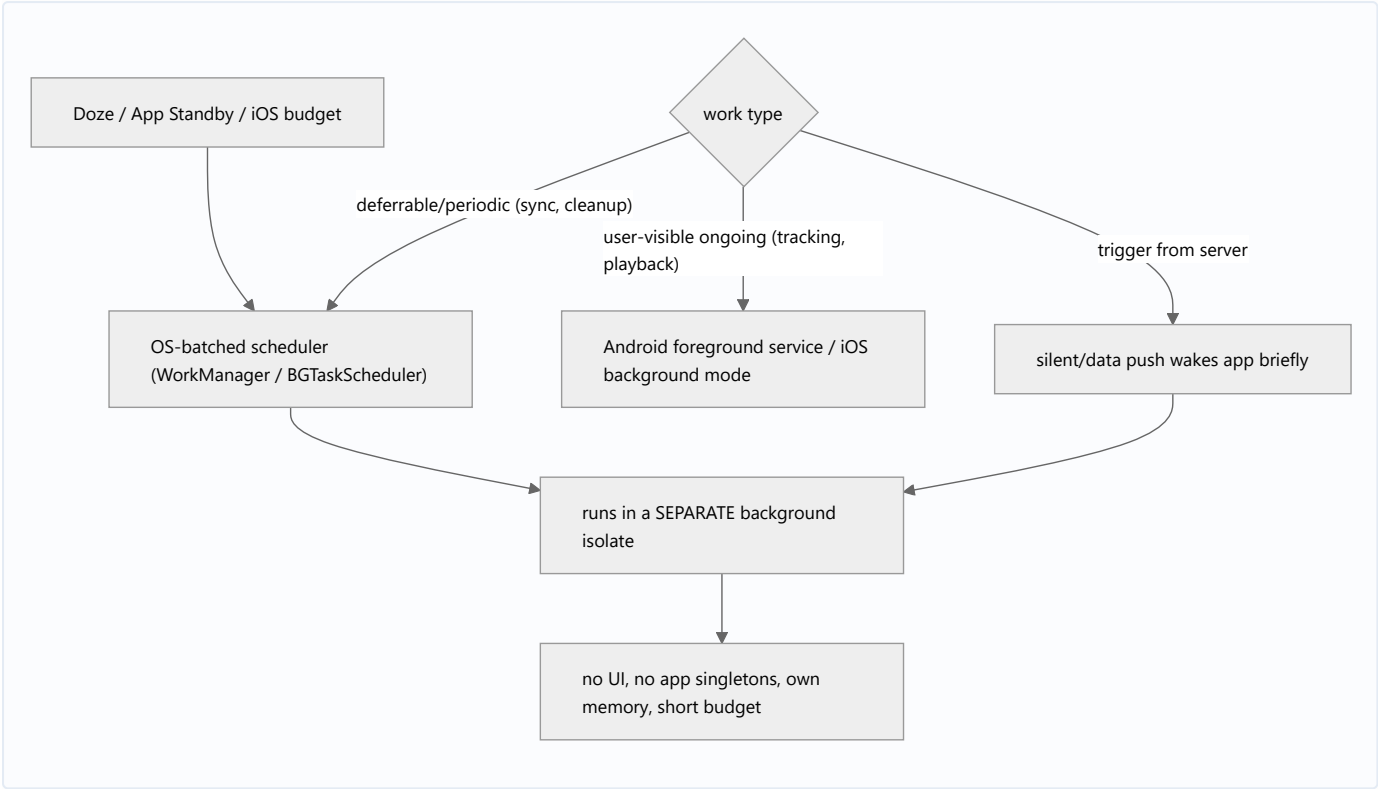
Real-world analogy

Background execution is like **after-hours building access**: you don't get free rein. Most tasks go on a **"do it whenever convenient" list** the facilities team batches overnight (deferrable/WorkManager/BGTask). Truly ongoing presence (a night guard on patrol) requires a **visible, justified badge** (foreground service / background mode) — and even then security can revoke it if you abuse it (OEM battery killers).

Problem Statement

Decide whether "sync every hour," "track a run continuously," and "process on push" are even feasible in the background, understand why a naive `Timer` / `Future` won't run when the app is killed, and design work to survive the OS's restrictions. You'll map work types to what the OS allows.

Internal Working



- **The OS suspends your app** when backgrounded; in-app `Timer` / `Future` / `Stream` **stop running** once suspended/killed. You cannot "just keep a loop going."
- **Background isolate**: OS-scheduled background callbacks run in a **fresh isolate** (separate entry point), with **its own memory** — **no access** to your UI, providers, or in-memory singletons (02 · isolates). It must re-init what it needs (Firebase, DB, plugins) and communicate via **persistent storage**, not shared variables.
- **Work spectrum**:
  - **Deferrable/periodic** (sync, cache cleanup, upload) → **WorkManager (Android)** / `BGTaskScheduler` **(iOS)**: the OS decides *when* (batched, respecting Doze/budget/constraints). **Not exact.**
  - **Continuous/user-visible** (navigation, run tracking, media, calls) → **Android foreground service** (persistent notification) / **iOS background mode** (location/audio/VoIP). Justified, ongoing, but scrutinized.
  - **Server-triggered** → **silent/data push** wakes the app briefly to do a little work (32 · fcm_push).
- **Timing is best-effort**: Doze, App Standby, low battery, and **OEM battery killers** (Xiaomi/Huawei/Samsung, etc.) delay or drop background work. **Never assume it ran on time** — design to reconcile/catch up when the app next opens (Module 19).
- **iOS is stricter than Android**: iOS gives short, opportunistic windows the system learns from usage; there's no general "run every 15 min guaranteed." Android is more permissive but still Doze-gated.

Memory Representation

The background isolate has a **separate heap**; nothing from the UI isolate is visible. Durable state must live in storage (DB/prefs/files) that both isolates read/write (Module 15).

Compiler Behavior

Background entry points must be **top-level/static** and annotated `@pragma('vm:entry-point')` so AOT retains them (32 · fcm_push).

Runtime Behavior

Scheduled callbacks fire when the OS decides (subject to constraints/Doze/budget); continuous work runs only while the foreground service/background mode is active. Killed apps run **nothing** except OS-scheduled callbacks and push-triggered wakes.

Flutter Engine Behavior

The OS spins up a **background** `FlutterEngine` **/isolate** to run the Dart callback, then tears it down. It's not your running app — it's a fresh, minimal engine (10 · embedder_and_startup).

Dart VM Behavior

Two (or more) isolates with **no shared memory**; communicate via storage or platform mechanisms, not references. Keep the background isolate's work short (budgets/timeouts).

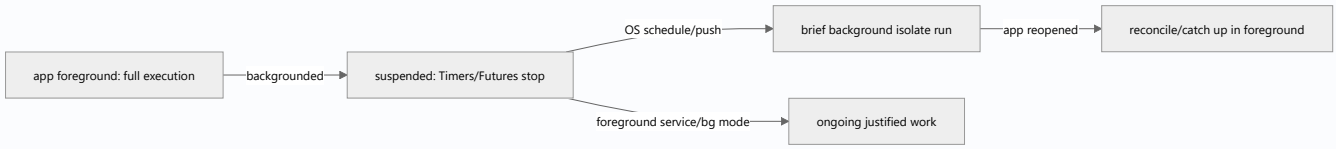
Examples

```
// ANTI-PATTERN: this does NOT keep running in the background.
// Once the app is suspended/killed, the Timer stops. Do not rely on it.
Timer.periodic(const Duration(minutes: 15), (_) => sync()); // ❌ only runs while alive


// CORRECT model: hand deferrable work to the OS scheduler (see workmanager file),
// which runs it in a background isolate that re-inits its own dependencies.
@pragma('vm:entry-point')            // top-level, retained by AOT
Future<void> backgroundTask() async {
  // Fresh isolate: no UI, no app singletons. Re-init what you need:
  // await Firebase.initializeApp(); final db = await openDb();
  // Do SHORT work; persist results to storage (not memory).
  // Assume you may be killed early — make it resumable/idempotent.
}
```

```
Feasibility cheat-sheet:
  "sync every hour"      -> deferrable/periodic; OS decides timing (NOT exact). ✅ (best-effort)
  "track a run live"     -> foreground service (Android) / location bg mode (iOS). ✅ (justified)
  "process on server event" -> silent/data push wake. ✅ (brief, best-effort)
  "run my loop forever in bg" -> ❌ not allowed; OS suspends/kills it.
```

Diagrams



Common Mistakes

Mistake	Why it fails	Fix
Relying on `Timer` / `Future` in background	Stops when suspended/killed	Use OS scheduler / foreground service
Accessing app state in background isolate	Separate memory	Re-init deps; use storage to share
Assuming exact timing	Doze/budget/OEM killers	Best-effort; reconcile on open
No `@pragma('vm:entry-point')`	Entry point stripped (AOT)	Top-level/static + annotation
Long background work	Killed at budget/timeout	Keep short, resumable, idempotent
Same design both platforms	iOS far stricter	Platform-specific strategy

Best Practices

- Model background work as **best-effort and deferrable**; **reconcile/catch up in the foreground** — never assume it ran on time.
- Use the **OS scheduler** (WorkManager/ `BGTaskScheduler`) for periodic work, a **foreground service/background mode** only for justified continuous work, and **silent push** for server triggers.
- Treat the background isolate as **isolated**: re-init dependencies, share via **storage**, keep work **short/idempotent/resumable** (top-level `@pragma('vm:entry-point')` entry).
- Plan **per-platform** (iOS stricter); account for **OEM battery killers**; degrade gracefully.

Performance

The whole model exists for battery. Keep background work minimal (short bursts, batched, constraint-gated: charging/wifi). Continuous work (foreground service/location) is the biggest drain — justify and bound it.

Advantages / Disadvantages

- **+** (Within limits) sync/uploads/tracking without the UI open, battery-respecting via OS scheduling, server-triggered wakes.
- **−** No guaranteed timing, isolate constraints (no shared state), strict iOS budgets, OEM unreliability, per-platform complexity.

Interview Questions

1. 🟢 **Why does a `Timer.periodic` not keep running in the background?** — The OS suspends/kills the app; in-app timers/futures stop — background work must be OS-scheduled or a foreground service.
2. 🟢 **Where does background work run, and what can't it access?** — In a separate background isolate with its own memory — no UI, no app singletons/state; it re-inits deps and shares via storage.
3. 🟡 **What are the three kinds of background work and their mechanisms?** — Deferrable/periodic (WorkManager/ `BGTaskScheduler`), continuous/visible (foreground service/background mode), server-triggered (silent push).
4. 🟡 **Why is timing not guaranteed?** — Doze/App Standby/iOS budgets and OEM battery killers batch, delay, or drop background execution.
5. 🟡 **How do iOS and Android differ?** — iOS is far stricter (short, learned, opportunistic windows; no guaranteed periodic); Android (WorkManager) is more permissive but Doze-gated.
6. 🔴 **How should you design a background sync given all this?** — Best-effort scheduling + short idempotent work + reconcile/catch-up in foreground; constraints (charging/wifi); per-platform.
7. 🔴 **Why must background entry points be top-level `@pragma('vm:entry-point')` ?** — They're invoked by a fresh engine/isolate; the annotation keeps them from being tree-shaken in AOT builds.

Senior Engineer Tips

- Design for "it might not run" from the start: idempotent, resumable work + a foreground catch-up sync is the only reliable pattern.

- Never share memory with the background isolate — persist a work queue/outbox and have both isolates operate on storage.
- Test on real OEM devices (not just Pixel/Simulator); aggressive battery managers are where "works on my phone" background features die.

Architect Perspective

The background model is a constraint framework, not an API: the OS owns *when*, you own *what* (short, idempotent, storage-backed). Architecting around best-effort scheduling + foreground reconciliation, with continuous work quarantined behind justified foreground services/background modes, produces features that survive Doze, budgets, and OEM killers. Every later file is a platform-specific realization of this model, and offline-first is its primary consumer (Module 19, 02_workmanager_and_periodic_tasks.md, 04_ios_background_execution.md).

Summary

- Background execution is a metered, revocable privilege; suspended apps stop timers/futures — only OS-scheduled callbacks, foreground services/background modes, and push wakes run.
- Work runs in a separate isolate (no shared state; re-init + storage); timing is best-effort (Doze/budgets/OEM).
- Design short, idempotent, resumable work + foreground reconciliation; plan per-platform (iOS stricter).

Revision Notes

- Suspended/killed → in-app `Timer` / `Future` / `Stream` stop; only OS scheduler, foreground service/bg mode, silent push run.
- Background isolate: separate heap, no UI/singletons; re-init deps; share via storage; top-level `@pragma('vm:entry-point')`; short/idempotent.
- Deferrable→WorkManager/`BGTaskScheduler`; continuous→foreground service/bg mode; server→silent push. Best-effort timing; reconcile in foreground; iOS stricter; OEM killers.

Practice Questions

1. Why won't an in-app loop keep syncing after the app is killed?
2. What can and can't the background isolate access?
3. Which work types map to which OS mechanisms?

Coding Questions

1. Write a compliant top-level background entry point that re-inits deps and persists to storage.
2. Classify a list of features by the correct background mechanism.
3. Sketch an idempotent, resumable background sync.

Mini Project

Background feasibility design (Flutter): For an app needing hourly sync, live run-tracking, and push-triggered refresh, document which OS mechanism each uses, why in-app timers won't work, and how the background isolate shares state via storage. Implement a compliant top-level background entry point stub that re-inits dependencies and writes an idempotent result to storage. Acceptance: correct mechanism per feature; isolate/no-shared-state constraint addressed; best-effort + foreground reconciliation described; compliant entry point stub; per-platform differences noted.

WorkManager & Periodic Tasks (`workmanager`)

> The `workmanager` plugin schedules **deferrable, guaranteed-eventually** background work — one-off (`registerOneOffTask`) or periodic (`registerPeriodicTask`, min ~15 min) — that runs in a **background isolate** via a top-level `callbackDispatcher`, gated by **constraints** (network, charging, battery-not-low) with **backoff retries**. On Android it wraps native WorkManager (survives reboots/app-kill); on iOS it maps onto `BGTaskScheduler` (much more limited). The OS decides *when* — you get "it will run eventually when conditions are met," not exact timing.

Introduction

`workmanager` is the go-to for periodic/deferrable background jobs like sync, uploads, and cleanup. This file covers the callback dispatcher, one-off vs periodic tasks, constraints, retry/backoff, and the platform reality (Android robust, iOS limited) — the practical realization of the "deferrable" branch of the background model.

Why this concept exists

Apps need reliable *eventual* execution of maintenance work without draining battery. Native WorkManager (Android) provides constraint-based, persistent, retryable scheduling; `workmanager` exposes it to Flutter (and bridges iOS `BGTaskScheduler`). It replaces fragile timers with OS-managed jobs that respect Doze and survive restarts.

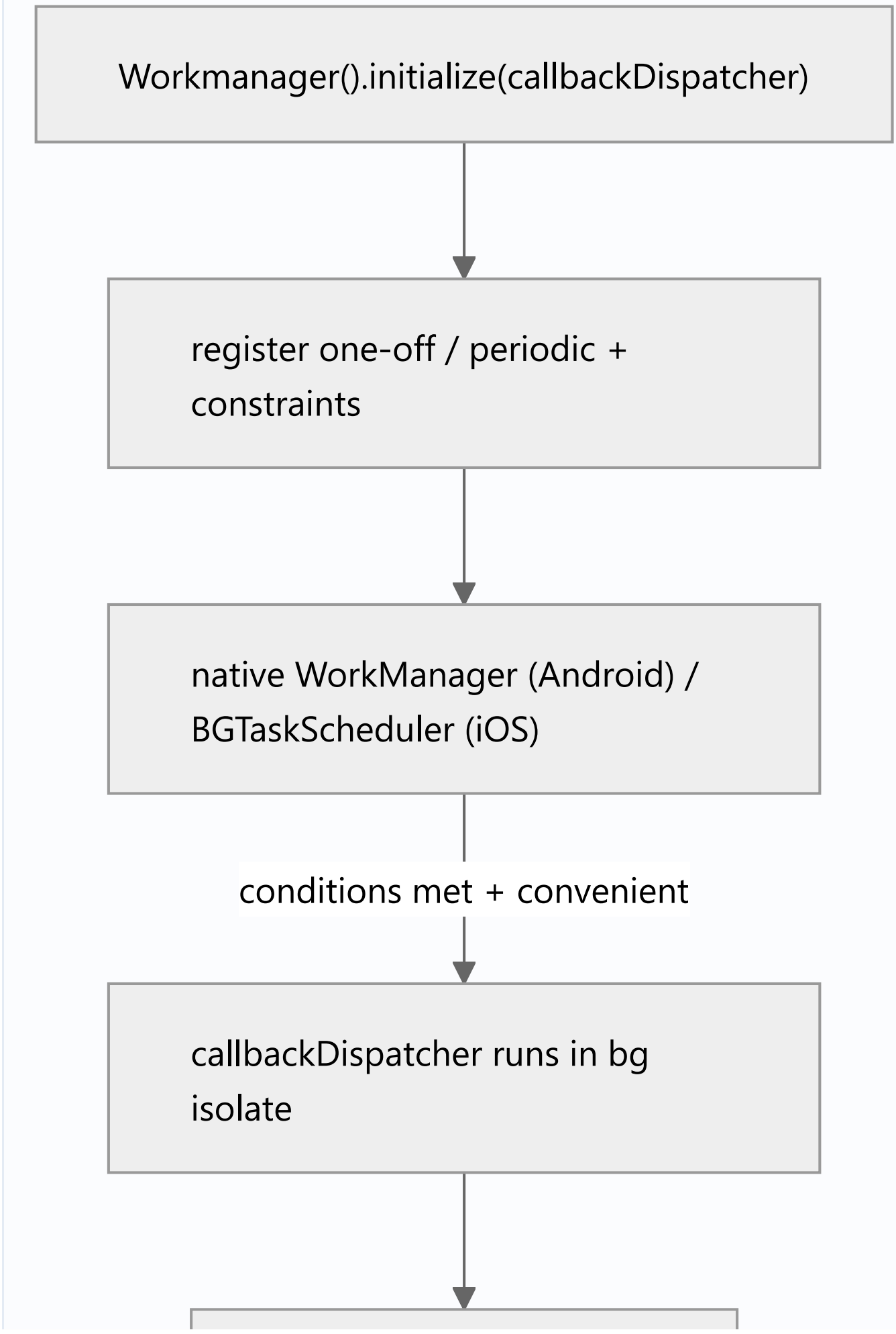
Real-world analogy

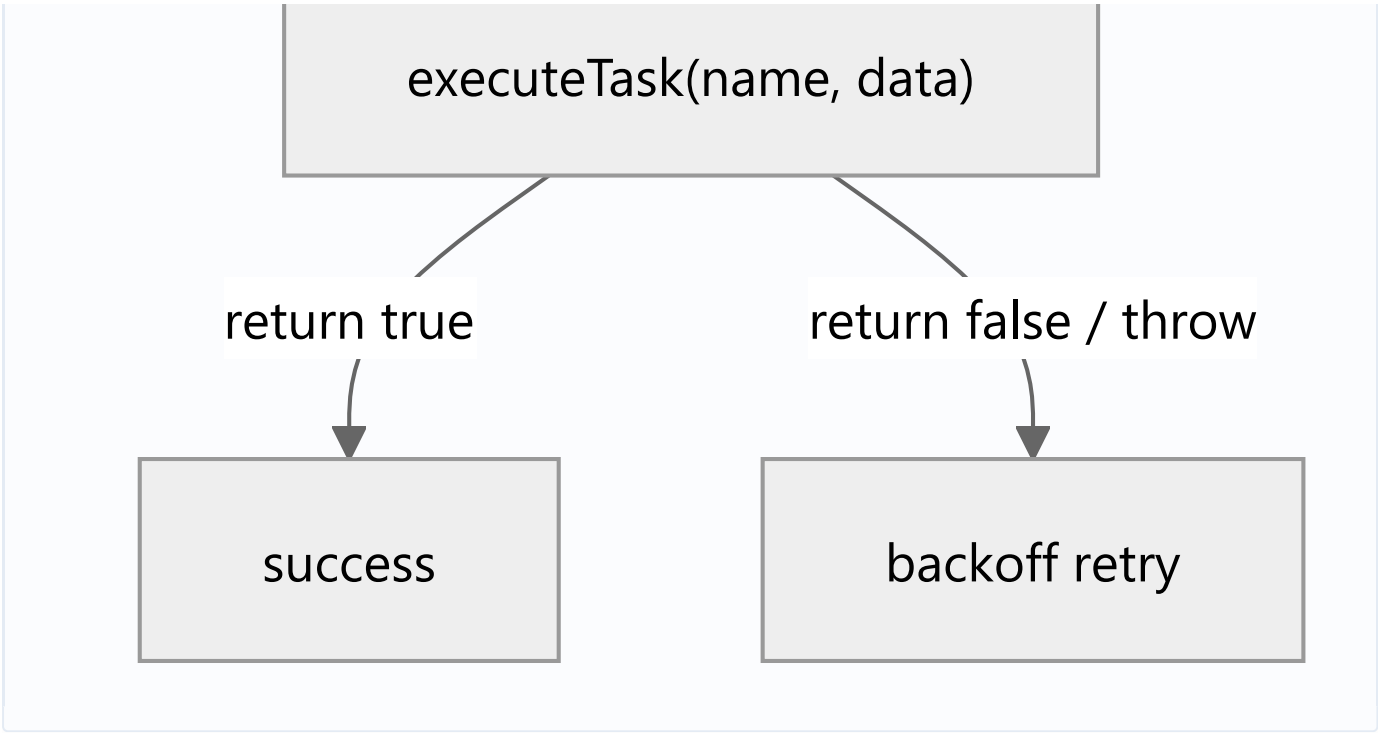
WorkManager is a **dependable errand service with conditions**: you drop off a task ("upload these photos") with rules ("only on wifi, only when charging"). You don't pick the exact time — the service runs it **when your conditions are met and it's convenient**, retries if it fails, and remembers the task even if you (the app) leave or the phone reboots.

Problem Statement

Sync an offline outbox roughly every 15 minutes **only when connected and charging**, retry with backoff on failure, and also run a one-off upload after the user acts — all surviving app-kill on Android. You'll set up the dispatcher, register periodic + one-off tasks with constraints, and handle results.

Internal Working


Workmanager().initialize(callbackDispatcher)
register one-off / periodic + constraints
native WorkManager (Android) / BGTaskScheduler (iOS)
conditions met + convenient
callbackDispatcher runs in bg isolate



- `callbackDispatcher` (top-level `@pragma('vm:entry-point')`): the single background entry point. `Workmanager().executeTask((taskName, inputData) async {...})` dispatches by task name; runs in a **fresh isolate** — re-init Firebase/DB/plugins, no app state.
- **Initialize** once at startup: `Workmanager().initialize(callbackDispatcher)`.
- **One-off**: `registerOneOffTask('upload-1', 'upload', inputData: {...}, constraints: ...)` — runs once when constraints allow.
- **Periodic**: `registerPeriodicTask('sync', 'sync', frequency: Duration(minutes: 15), constraints: ...)` — **minimum ~15 min**; the OS batches and may run **less often** (never guaranteed exact). One periodic task per unique name.
- **Constraints**: `Constraints(networkType: NetworkType.connected, requiresCharging: true, requiresBatteryNotLow: true, requiresStorageNotLow: ...)` — the OS waits until satisfied.
- **Return value = result**: return `true` for success; `false` (or throw) signals failure → **backoff retry** (`BackoffPolicy`). Make tasks **idempotent** (they can re-run).
- **Input/output**: pass small `inputData` (primitives); communicate results via **storage** (no shared memory).
- **Uniqueness/replace**: `existingWorkPolicy` (keep/replace/append) prevents duplicate scheduling.
- **Platform reality**: **Android** — robust, persists across reboot/kill, true constraints/retries. **iOS** — maps to `BGTaskScheduler`, needs registered task identifiers + background modes, and runs **rarely/opportunistically** (Apple decides); don't expect Android-like periodicity (04_ios_background_execution.md).

Memory Representation

Task metadata/queue is persisted by native WorkManager (survives restarts). The isolate has its own heap — share via storage. `inputData` is a small primitive map.

Compiler Behavior

`callbackDispatcher` must be top-level `@pragma('vm:entry-point')` (AOT retention).

Runtime Behavior

Tasks fire when constraints are met and the OS schedules them (batched, Doze-aware) — periodic is a *floor* frequency, not exact. Failures back off and retry. Killed/rebooted apps still run persisted tasks (Android).

Flutter Engine Behavior

Native WorkManager spins up a background `FlutterEngine` to run the dispatcher, then tears it down (01_background_execution_model.md).

Dart VM Behavior

Fresh isolate per run; no shared memory; keep work short (WorkManager expects tasks to finish within a limited window).

Examples

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


// Single background entry point — runs in a fresh isolate
@pragma('vm:entry-point')
void callbackDispatcher() {
  Workmanager().executeTask((taskName, inputData) async {
    // Re-init deps here (no app singletons): Firebase/DB/etc.
    switch (taskName) {
      case 'sync':
        final ok = await _syncOutbox();     // idempotent!
        return ok;                          // false -> backoff retry
      case 'upload':
        return _upload(inputData?['path'] as String?);
    }
```

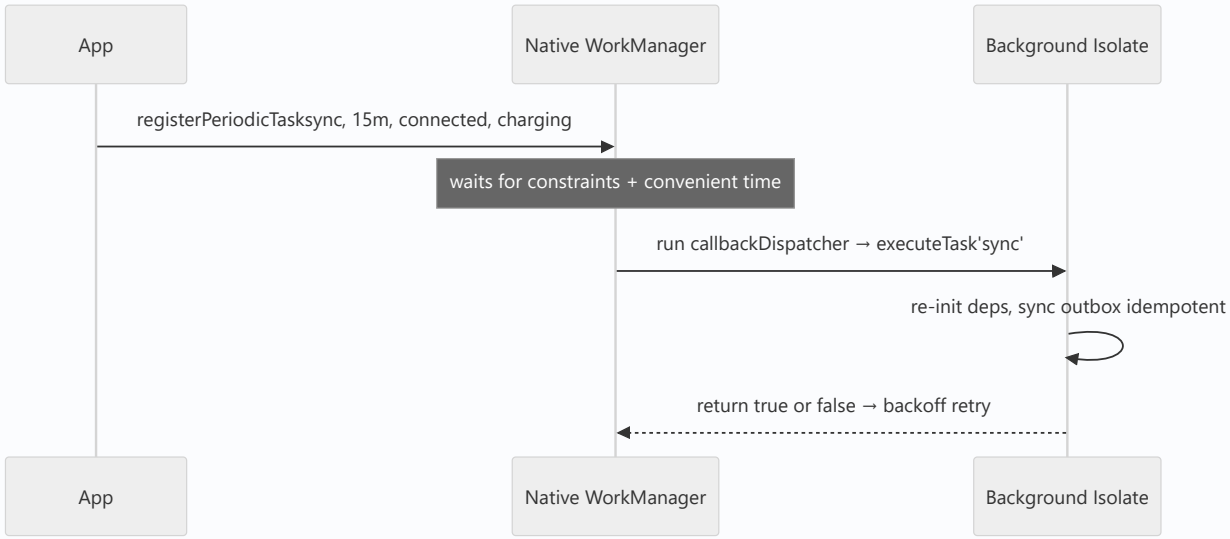
```
    return true;
  });
}

Future<void> initBackground() async {
  await Workmanager().initialize(callbackDispatcher);

  // Periodic sync: >=15 min, only connected + charging (best-effort timing)
  await Workmanager().registerPeriodicTask(
    'sync', 'sync',
    frequency: const Duration(minutes: 15),
    constraints: Constraints(
      networkType: NetworkType.connected,
      requiresCharging: true,
    ),
    existingWorkPolicy: ExistingWorkPolicy.keep,
    backoffPolicy: BackoffPolicy.exponential,
  );
}

// One-off upload triggered by user action
Future<void> scheduleUpload(String path) => Workmanager().registerOneOffTask(
      'upload-$path', 'upload',
      inputData: {'path': path},
      constraints: Constraints(networkType: NetworkType.connected),
    );
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Expecting exact periodic timing	OS batches; ~15min floor, best-effort	Design best-effort; reconcile in foreground
Non-idempotent tasks	Retries/re-runs duplicate effects	Make tasks idempotent
Accessing app state in dispatcher	Separate isolate	Re-init deps; share via storage
`callbackDispatcher` not top-level/ `vm:entry-point`	Not invoked	Top-level + annotation
Ignoring return value	No retry on failure	Return true/false correctly
Expecting iOS to behave like Android	iOS `BGTaskScheduler` runs rarely	iOS-specific expectations (ios file)
Long-running tasks	Killed at window limit	Keep short, chunk/resume

Best Practices

- One top-level `callbackDispatcher` (`@pragma('vm:entry-point')`) dispatching by task name; **re-init deps**, share via **storage**, keep tasks **short + idempotent**.
- Use **constraints** (connected/charging/battery-not-low) so work runs cheaply; return **true/false** for success/retry with **backoff**.
- Treat periodic as a **best-effort floor** (~15 min min), not exact — **reconcile in the foreground**; use `existingWorkPolicy` to avoid duplicates.
- Expect **iOS to run rarely** (`BGTaskScheduler` semantics — 04_ios_background_execution.md); degrade to foreground sync; wrap behind a service.

Performance

Constraints (charging/wifi) keep background work off battery/data budgets. Short, batched, idempotent tasks are OS-friendly; long/frequent tasks get throttled or killed. This is battery-optimal *if* you respect the constraint model.

Advantages / Disadvantages

- + Reliable eventual execution, constraint-based, retry/backoff, persists across reboot/kill (Android), battery-respecting.
- − No exact timing (~15 min floor), iOS severely limited, isolate constraints, must be idempotent, short-task requirement.

Interview Questions

1. 🟢 **What is `workmanager` for?** — Scheduling deferrable one-off/periodic background work (sync/upload/cleanup) that runs eventually under constraints, surviving app-kill (Android).
2. 🟢 **What's the minimum periodic frequency and is timing guaranteed?** — ~15 minutes minimum, and no — the OS batches/defers; treat it as a best-effort floor.
3. 🟡 **Why must tasks be idempotent?** — They can be retried (on failure/backoff) or re-run; effects must be safe to repeat.
4. 🟡 **How does the dispatcher access app state?** — It can't — it runs in a fresh isolate; re-init dependencies and share via storage.
5. 🟡 **How are constraints and retries expressed?** — `Constraints(networkType/requiresCharging/...)` and the task's return value (`true` success / `false` retry) with a `BackoffPolicy`.
6. 🔴 **How does `workmanager` differ across platforms?** — Android wraps native WorkManager (robust, persistent, true constraints); iOS maps to `BGTaskScheduler` and runs rarely/opportunistically — don't expect periodicity.
7. 🔴 **Why keep tasks short?** — The OS grants a limited execution window; long tasks get killed — chunk and make resumable.

Senior Engineer Tips

- Make every task idempotent and resumable, and always pair background sync with a foreground catch-up — background alone is never reliable enough.
- Gate with constraints (charging/wifi) both to respect battery and to increase the odds the OS actually runs it.
- Set iOS expectations explicitly with stakeholders: `workmanager` periodic ≈ "sometimes" on iOS; if you need iOS reliability, lean on silent push + foreground sync.

Architect Perspective

`workmanager` operationalizes the deferrable-work branch of the background model: constraint-gated, retryable, idempotent tasks the OS runs when convenient. Architecturally it's a scheduling adapter behind your `BackgroundService`, feeding the same idempotent sync logic the foreground uses — so background is a best-effort accelerator, not a dependency. Its platform asymmetry (robust Android, limited iOS) makes a cross-platform strategy (+ foreground reconciliation) essential (01_background_execution_model.md, 05_background_integration.md, Module 19).

Summary

- `workmanager`: top-level `callbackDispatcher` (isolate) dispatching one-off/periodic tasks under constraints with backoff retries; Android-robust, iOS-limited.
- Periodic ≥ ~15 min, best-effort (not exact); tasks must be short + idempotent; share via storage; return true/false.
- Reconcile in foreground; set iOS expectations; wrap behind a service.

Revision Notes

- `Workmanager().initialize(callbackDispatcher)`; `registerOneOffTask` / `registerPeriodicTask(frequency ≥15min)`; dispatch via `executeTask((name,data){})`.
- `callbackDispatcher` top-level `@pragma('vm:entry-point')`; fresh isolate → re-init deps, storage-shared, short + idempotent.
- `Constraints(networkType/requiresCharging/...)`; return true/false + `BackoffPolicy`; `existingWorkPolicy`; Android native WM (persistent) vs iOS `BGTaskScheduler` (rare).

Practice Questions

1. Why is periodic timing not exact, and what's the minimum?
2. How does the dispatcher get its dependencies?
3. What does returning `false` from a task do?

Coding Questions

1. Write a `callbackDispatcher` dispatching `sync` + `upload` tasks (idempotent).
2. Register a periodic sync with connected+charging constraints and backoff.
3. Register a one-off upload with input data on a user action.

Mini Project

Constraint-based background sync (Flutter): Implement a `callbackDispatcher` with an idempotent outbox `sync` task and a one-off `upload` task, register periodic sync (≥15 min, connected+charging, exponential backoff) and one-off uploads, re-initializing deps in the isolate and sharing via storage. Add a foreground catch-up sync. Acceptance: dispatcher top-level/ `vm:entry-point` ; tasks idempotent + short; constraints + backoff applied; survives app-kill on Android; foreground reconciliation present; iOS limitations documented.

Android Foreground Services (Ongoing, User-Visible Work)

> When Android work must run **continuously and visibly** — live location tracking, media playback, an active call, an ongoing upload — you use a **foreground service**: it shows a **mandatory persistent notification** (so the user knows it's running), is far less likely to be killed than background work, and (on Android 10+/14) requires a **declared** `foregroundServiceType` + matching runtime permissions. In Flutter this is done via a plugin (e.g. `flutter_foreground_task`). It's the *only* sanctioned way to keep running while backgrounded — but it's scrutinized and battery-costly, so use it only for genuinely ongoing, user-aware tasks.

Introduction

Foreground services are Android's answer to "I need to keep working while the user isn't in the app, and they should know." Unlike WorkManager (deferrable), a foreground service runs *now and continuously*. This file covers when they're justified, the persistent-notification requirement, service types/permissions, and running Dart work in one via a plugin.

Why this concept exists

Android kills background work to save battery, but some tasks legitimately must persist (a run being tracked, a song playing). A foreground service is the OS's contract: *you may keep running, but you must show a notification so the user is aware and can stop you.* Android 8+ enforces this; 10+/14 add type declarations to prevent abuse.

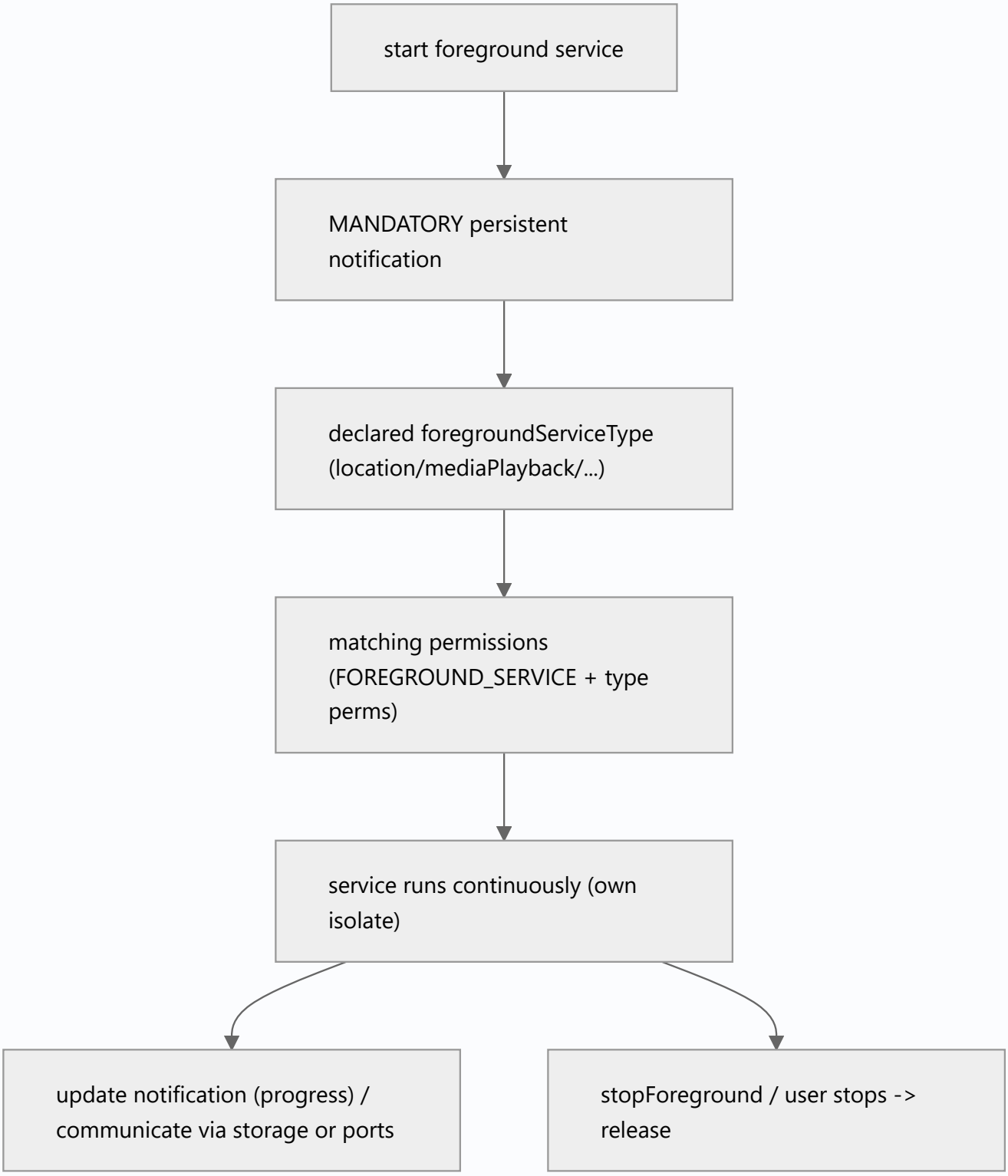
Real-world analogy

A foreground service is a **visible, badged on-site contractor**: unlike an anonymous after-hours worker (background task) who gets kicked out, the contractor wears a **hi-vis badge the whole time** (persistent notification) so everyone knows they're there and why — and because they're accountable and justified, security lets them keep working. But you don't hire one to change a lightbulb (deferrable work) — only for jobs that genuinely need continuous on-site presence.

Problem Statement

Track a user's run continuously (location every few seconds) while they switch apps or lock the screen, showing a "Tracking run — 2.4 km" notification they can stop, without the OS killing it. You'll run a foreground service hosting the location stream.

Internal Working



- **When justified**: continuous, user-visible/expected work — **location tracking, media/audio playback, active phone/VoIP call, ongoing large upload/download, fitness tracking**. Not for deferrable maintenance (use WorkManager) or hidden work.
- **Mandatory notification**: a foreground service **must** post an ongoing notification (Android 8+) with a channel (32 · local_notifications); update it for progress; the user can stop the

service from it.

- **Service type (Android 10+, stricter in 14)**: declare `android:foregroundServiceType` (e.g., `location`, `mediaPlayback`, `dataSync`, `camera`, `microphone`) in the manifest and start with that type; **Android 14** requires the type + justification + the corresponding runtime permission (e.g., `ACCESS_FINE_LOCATION` for `location`).
- **Permissions/manifest**: `FOREGROUND_SERVICE` (+ type-specific like `FOREGROUND_SERVICE_LOCATION`) and the underlying data permission (location/mic/etc.) (27 · permissions).
- **Flutter execution**: a plugin (`flutter_foreground_task` or similar) runs a Dart callback in the service's **background isolate**; communicate with the UI via **ports/storage** (no shared memory). You can host a location stream (29 · geolocation) there.
- **Lifecycle**: start on user action, keep the notification updated, **stop promptly** when done (`stopForeground`) — running longer than needed drains battery and annoys users.
- **iOS**: no direct equivalent — iOS uses **background modes** (location/audio) instead (04_ios_background_execution.md); design cross-platform accordingly.

Memory Representation

The service runs in its own isolate/heap; share state with the UI via **send ports / storage**. The notification is OS-owned.

Compiler Behavior

Manifest declares the service + `foregroundServiceType`; the Dart callback entry point is top-level `@pragma('vm:entry-point')`.

Runtime Behavior

The service runs continuously while active (much harder for the OS to kill than plain background work), consuming battery (esp. location/GPS). Missing type/permission (Android 14) throws at start.

Flutter Engine Behavior

The plugin hosts a background `FlutterEngine`/isolate for the service callback, separate from the UI engine.

Dart VM Behavior

Separate isolate; communicate via ports/storage; keep the loop efficient (throttle location, batch writes).

Examples

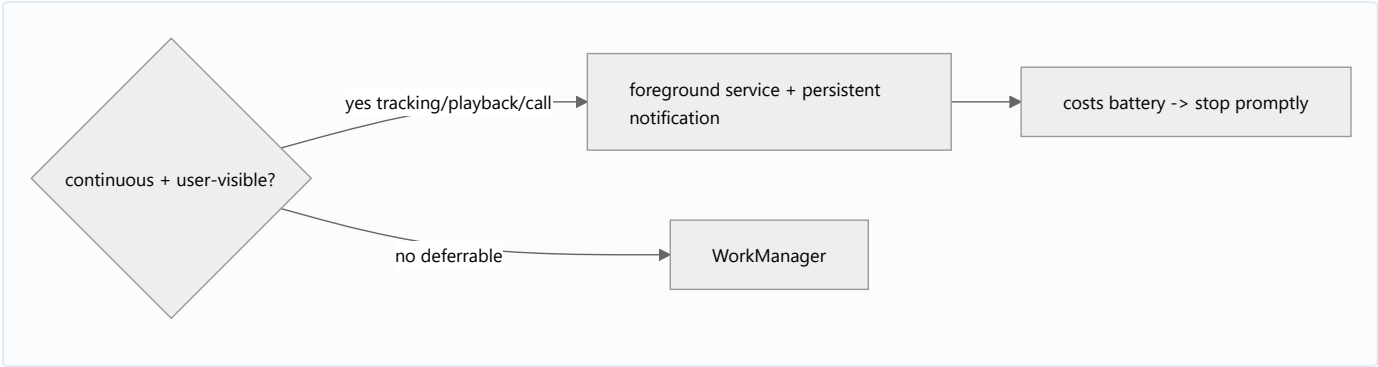
```
<!-- AndroidManifest.xml -->
<uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<application>
  <service
     android:name="...ForegroundService"
     android:foregroundServiceType="location"    <!-- required (Android 10+/14) -->
     android:exported="false"/>
</application>
```

```
// Using a foreground-task plugin (illustrative): run location tracking in the service
@pragma('vm:entry-point')
void startCallback() {
  FlutterForegroundTask.setTaskHandler(RunTrackerHandler()); // runs in service isolate
}

class RunTrackerHandler extends TaskHandler {
  StreamSubscription? _loc;
  @override
  Future<void> onStart(DateTime ts, TaskStarter starter) async {
    _loc = geolocator.track().listen((p) {
      _appendToStore(p);                        // persist (share via storage)
      FlutterForegroundTask.updateService(      // update the persistent notification
        notificationTitle: 'Tracking run',
        notificationText: '${_distanceKm()} km',
      );
    });
  }
  @override
  Future<void> onDestroy(DateTime ts) async => _loc?.cancel(); // release on stop
}

// Start on user action (after requesting location permission):
// await FlutterForegroundTask.startService(notificationTitle: 'Tracking run', callback: startCallback);
// Stop when done: await FlutterForegroundTask.stopService();
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Foreground service for deferrable work	Battery/UX abuse, rejection	Use WorkManager for deferrable
Missing persistent notification	Illegal (Android 8+); crash	Post an ongoing notification + channel
Missing `foregroundServiceType` /perms (14)	Start throws	Declare type + runtime permissions
Not stopping when done	Battery drain, annoyance	`stopService` promptly
Accessing UI state in the service	Separate isolate	Ports/storage to communicate
Assuming iOS parity	No FG service on iOS	Use iOS background modes

Best Practices

- Use a foreground service **only for genuinely continuous, user-visible work** (location/media/call/upload); use **WorkManager** for deferrable.
- Always post + update the **mandatory notification** (channel), let the user **stop** it, and **stop the service promptly** when finished.
- Declare the correct `foregroundServiceType` + **runtime permissions** (Android 14); communicate with the UI via **ports/storage**; throttle heavy streams (location).
- Design **cross-platform**: pair with iOS **background modes**; wrap behind a service so the app requests "track a run," not "start a service" (05_background_integration.md).

Performance

Continuous execution (esp. GPS/location) is the heaviest background battery cost. Throttle sampling (distance filter — 29 · geolocation), batch persistence, and stop the service the instant the task ends.

Advantages / Disadvantages

- + Reliable continuous execution (resists kill), user-visible/accountable, supports location/media/uploads while backgrounded.
- − Mandatory notification, heavy battery cost, Android 14 type/permission strictness, isolate comms, no iOS equivalent (background modes instead).

Interview Questions

1. 🟢 **When do you use a foreground service vs WorkManager?** — Foreground service for continuous, user-visible work (tracking/playback/call); WorkManager for deferrable/periodic maintenance.
2. 🟢 **What's mandatory for a foreground service?** — An ongoing persistent notification (Android 8+) informing the user it's running.
3. 🟡 **What changed for foreground services in Android 10/14?** — A declared `foregroundServiceType` (10+) and, in 14, type + justification + the matching runtime permission are required.
4. 🟡 **How does the service communicate with the UI?** — Via send ports / shared storage — it runs in a separate isolate with no shared memory.
5. 🟡 **Why stop the service promptly?** — Continuous execution (esp. location) drains battery and annoys users; keep it alive only while the task is active.
6. 🔴 **What's the iOS equivalent?** — There isn't a foreground service; iOS uses background modes (location/audio/VoIP) with their own constraints.
7. 🔴 **Why is a foreground service harder for the OS to kill?** — It's a declared, notified, accountable ongoing task — the OS treats it as user-important, unlike anonymous background work.

Senior Engineer Tips

- Reserve foreground services for the short list Android actually blesses (location/media/call/upload); anything else risks rejection and battery complaints.
- Get the Android 14 type + permission matrix right up front — it's a hard start-time failure otherwise.
- Throttle the work inside (distance-filtered location, batched writes) and stop the instant the session ends; a leaked tracking service is a battery horror story.

Architect Perspective

Foreground services are the "justified continuous work" branch of the background model — powerful but accountable (notification) and costly (battery). Behind a `BackgroundService`, the app expresses intent ("track this run") while the platform layer picks a foreground service on Android and a background mode on iOS, keeping the work throttled, stoppable, and permission-correct. This quarantines the OS-specific complexity and battery risk behind one seam (01_background_execution_model.md, 04_ios_background_execution.md, 05_background_integration.md).

Summary

- Foreground services run continuous, user-visible work with a mandatory persistent notification, resisting OS kill.
- Android 10+/14 require `foregroundServiceType` + matching runtime permissions; communicate via ports/storage; stop promptly (battery).
- Use only when justified (location/media/call/upload); no iOS equivalent (background modes); wrap behind a cross-platform service.

Revision Notes

- Continuous + visible → foreground service; mandatory ongoing notification (channel); user can stop.
- Manifest: `FOREGROUND_SERVICE` (+ `FOREGROUND_SERVICE_LOCATION` etc.) + `foregroundServiceType` (Android 10+/14) + data permission.
- Runs in own isolate (ports/storage to UI); throttle (location distance filter), batch writes, `stopService` promptly; iOS → background modes.

Practice Questions

1. When is a foreground service the right tool vs WorkManager?
2. What must every foreground service display and why?
3. What does Android 14 require to start a typed foreground service?

Coding Questions

1. Declare a `location` foreground service (manifest type + permissions).
2. Run a throttled location stream in the service and update its notification.
3. Start/stop the service on user actions and release resources.

Mini Project

Live run tracker (Flutter + Android): Build a foreground-service-based run tracker: start on "Start run," track distance-filtered location in the service isolate, update a persistent "Tracking — X km" notification, persist points to storage, and stop on "Stop run" (release the stream). Declare the `location` service type + permissions (Android 14). Acceptance: tracks continuously while backgrounded/locked; persistent notification updates + stoppable; correct type/permissions; throttled + battery-conscious; stops promptly; behind a service (iOS uses background modes).

iOS Background Execution (`BGTaskScheduler`, Fetch, Silent Push)

> iOS grants background time **grudgingly and opportunistically** — there is **no guaranteed "run every N minutes."** You get: `BGAppRefreshTask` (short, ~30s, opportunistic refresh the system schedules based on usage), `BGProcessingTask` (longer maintenance, typically overnight while charging), **silent/content-available push** (server-triggered brief wake), and continuous **background modes** (location/audio/VoIP) only if genuinely used. All require **registered task identifiers + Background Modes capability**, run in a background isolate, and must **finish fast or be killed** — so the reliable pattern is *server-triggered + best-effort + foreground catch-up*.

Introduction

iOS background execution is the strictest part of this module. This file covers the `BGTaskScheduler` task types, silent push, continuous background modes, the setup (identifiers/capabilities), and — crucially — how little you can rely on any of it, shaping a design that leans on push + foreground reconciliation.

Why this concept exists

Apple prioritizes battery and privacy over developer convenience: it learns app usage and grants background windows opportunistically, rather than honoring fixed schedules. `BGTaskScheduler` (iOS 13+) replaced older background-fetch APIs with a model where the **system decides timing** and enforces tight budgets. Continuous modes exist only for legitimately ongoing needs (navigation, audio).

Real-world analogy

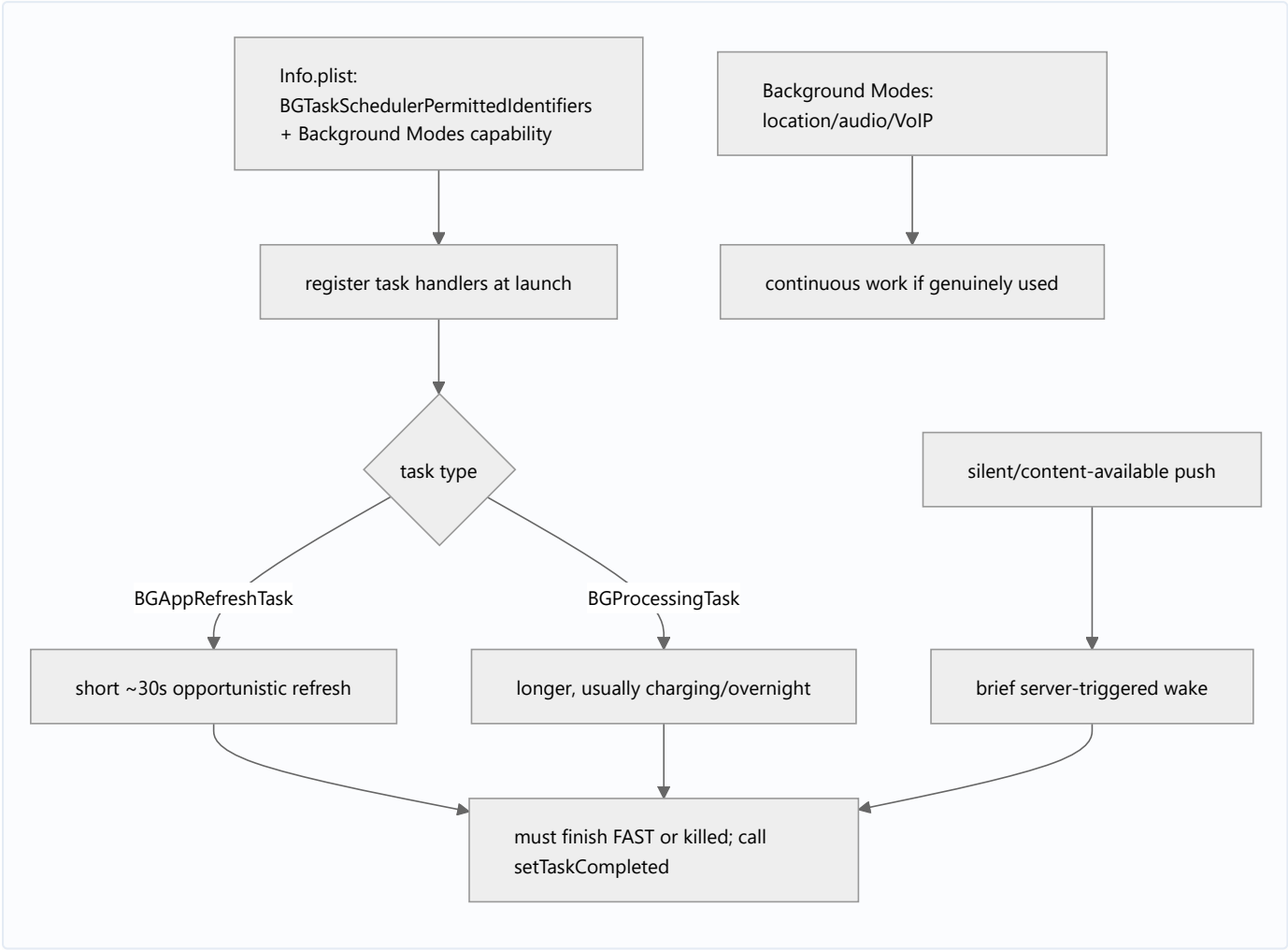
iOS background time is a **standby-only gig with no fixed shifts**: you can't schedule yourself. The manager (iOS) **calls you in briefly when it happens to be convenient** (`BGAppRefreshTask`), gives you **longer overnight tasks while plugged in** (`BGProcessingTask`), or a **client can page you for a quick job** (silent push). A **continuous shift** (background mode) is only for roles that truly require presence (a live navigator) — and you must clock out fast or you're sent home (killed).

Problem Statement

Refresh content periodically on iOS, run heavier cleanup overnight while charging, trigger a sync from the server, and support continuous location during navigation — all within Apple's limits, with foreground catch-up because none of it is guaranteed. You'll register BG tasks, enable

capabilities, and handle silent push.

Internal Working



- **Setup**: add each task id to `BGTaskSchedulerPermittedIdentifiers` in `Info.plist`, enable the **Background Modes** capability (Background fetch / Background processing / Remote notifications / Location / Audio as needed — 28 · ios_integration), and **register handlers at launch** (before the app finishes launching).
- `BGAppRefreshTask`: short (~30s), for keeping content fresh; the system schedules it **opportunistically** based on how often the user opens the app — **not** a fixed interval. You submit a request; iOS decides if/when.
- `BGProcessingTask`: longer-running maintenance (DB compaction, large sync); typically runs when **charging + idle** (often overnight); can request network/charging requirements.
- **Silent push** (`content-available: 1`): the server wakes the app briefly to fetch/sync (32 · fcm_push); still budget-limited and **throttled** by iOS (over-use → delivery reduced). The most reliable "run something now-ish" trigger, but still best-effort.
- **Continuous background modes**: **location** (turn-by-turn), **audio** (playback), **VoIP** — allow ongoing execution **only if the app genuinely provides that feature**; claiming unused modes → **App Store rejection**.

- **Budgets + completion**: every task **must call** `setTaskCompleted` quickly (finish within the granted window) and **schedule the next one** itself; overrunning → the task is killed and future budget shrinks.
- **In Flutter**: use plugins (`workmanager` maps periodic → `BGTaskScheduler` ; `background_fetch` ; `firebase_messaging` for silent push) — runs in a background isolate (re-init deps, share via storage).
- **Reality**: none of this is guaranteed or periodic. **Design for "maybe, eventually" + foreground catch-up.**

Memory Representation

Background isolate with its own heap; persist to storage. Task requests/identifiers are managed by iOS.

Compiler Behavior

Task identifiers must be declared in `Info.plist` ; handlers registered at launch; Dart entry points top-level `@pragma('vm:entry-point')` .

Runtime Behavior

Tasks run when iOS chooses (opportunistic/overnight), within tight time budgets; must complete fast. Silent push wakes briefly but is throttled. Overuse or overrun shrinks future opportunities.

Flutter Engine Behavior

iOS spins up a background `FlutterEngine` /isolate for the task/push handler, then suspends it (01_background_execution_model.md).

Dart VM Behavior

Separate isolate; no shared memory; keep work minimal and idempotent; complete before the budget expires.

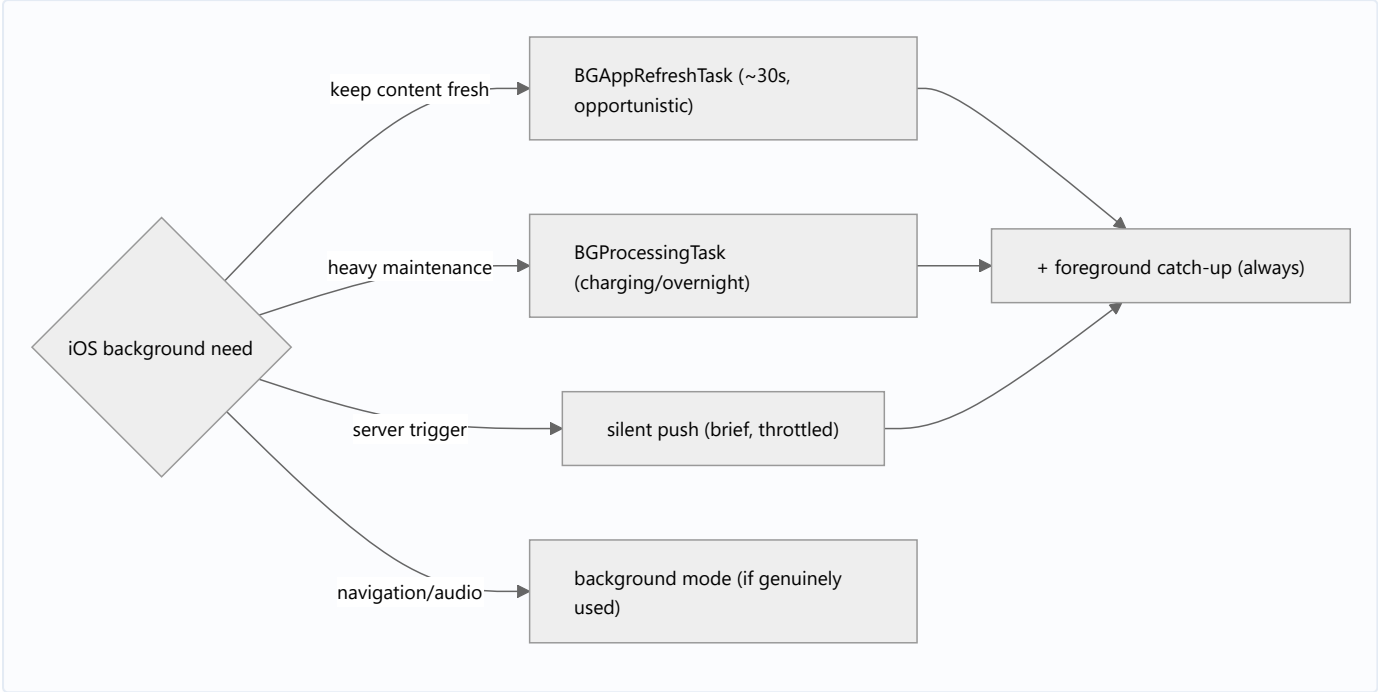
Examples

```
<!-- ios/Runner/Info.plist -->
<key>BGTaskSchedulerPermittedIdentifiers</key>
<array>
  <string>com.example.app.refresh</string>
  <string>com.example.app.processing</string>
</array>
<key>UIBackgroundModes</key>
<array>
  <string>fetch</string>
  <string>processing</string>
  <string>remote-notification</string>   <!-- silent push -->
  <!-- <string>location</string> only if you provide navigation/tracking -->
</array>
```

```
// AppDelegate: register + handle a BGAppRefreshTask (schedule the next one each run)
import BackgroundTasks
func registerBGTasks() {
  BGTaskScheduler.shared.register(forTaskWithIdentifier: "com.example.app.refresh", using: nil) { task in
    scheduleNextRefresh()                       // always reschedule
    // do SHORT work (or bridge to Flutter), then:
    task.setTaskCompleted(success: true)        // MUST complete quickly
  }
}
func scheduleNextRefresh() {
  let req = BGAppRefreshTaskRequest(identifier: "com.example.app.refresh")
  req.earliestBeginDate = Date(timeIntervalSinceNow: 15 * 60)  // hint only; iOS decides
  try? BGTaskScheduler.shared.submit(req)
}
```

```
// Flutter side: silent push is the most reliable "sync now-ish" trigger (still best-effort)
@pragma('vm:entry-point')
Future<void> onSilentPush(RemoteMessage m) async {
  // re-init deps; do a SHORT idempotent sync; persist to storage.
}
// Pair with a foreground catch-up sync — never rely on background alone on iOS.
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Expecting fixed periodic runs	iOS is opportunistic, not scheduled	Best-effort + foreground catch-up
Not calling `setTaskCompleted`	Killed; future budget shrinks	Complete quickly; reschedule
Long tasks	Overrun budget → killed	Keep short; chunk/resume
Claiming unused background modes	App Store rejection	Only modes you genuinely use
Missing permitted identifiers/capability	Task won't run/register fails	Declare ids + enable capability
Over-using silent push	iOS throttles delivery	Use sparingly; reconcile in foreground

Best Practices

- Treat iOS background as **opportunistic/best-effort**: `BGAppRefreshTask` (short refresh), `BGProcessingTask` (charging/overnight maintenance), **silent push** (server trigger) — and **always add foreground catch-up**.
- Register handlers **at launch**, declare **permitted identifiers + Background Modes**, `setTaskCompleted` **fast**, and **reschedule** the next task each run.
- Use **continuous background modes only for genuine features** (navigation/audio) — never claim unused ones (rejection).
- Keep work **short, idempotent, storage-backed**; prefer **silent push** as the most reliable trigger but use it **sparingly** (throttling).

Performance

iOS's whole design optimizes battery: short opportunistic windows, charging-gated heavy tasks. Respect budgets (complete fast) and use silent push sparingly; overuse/overrun reduces your future background allotment.

Advantages / Disadvantages

- + Battery-safe refresh/maintenance, server-triggered wakes, continuous modes for real features — within Apple's contract.
- − No guaranteed timing, tight budgets, throttled silent push, rejection risk for unused modes, isolate constraints, requires foreground fallback.

Interview Questions

1. 🟢 **Is there a guaranteed periodic background task on iOS?** — No; iOS schedules `BGAppRefreshTask` opportunistically based on usage — treat all background as best-effort.
2. 🟢 **What are the iOS background task types?** — `BGAppRefreshTask` (short refresh), `BGProcessingTask` (longer, charging/overnight), plus silent push and continuous background modes.
3. 🟡 **What's the most reliable way to trigger background work on iOS?** — Silent/content-available push (server-triggered brief wake) — but it's throttled, so use sparingly + reconcile in foreground.
4. 🟡 **What must every BG task do, and what's the risk of overrunning?** — Call `setTaskCompleted` within its budget and reschedule; overrunning gets it killed and shrinks future budget.
5. 🟡 **When can you use a continuous background mode?** — Only when the app genuinely provides that feature (navigation/audio/VoIP); claiming unused modes causes rejection.
6. 🔴 **How do you set up `BGTaskScheduler` in an iOS app?** — Declare identifiers in `BGTaskSchedulerPermittedIdentifiers`, enable Background Modes, register handlers at launch, submit requests, complete fast, reschedule.
7. 🔴 **Why is a foreground catch-up essential on iOS?** — Background execution is unguaranteed/throttled; the app must reconcile state when opened so missed runs don't cause staleness.

Senior Engineer Tips

- Assume iOS background "runs sometimes" and build the real reliability into silent push + foreground reconciliation; `BGAppRefreshTask` is a bonus, not a foundation.

- Register tasks at launch and always reschedule inside the handler — a task that doesn't resubmit itself runs once and never again.
- Only declare background modes you can defend in review; an unused `location` / `audio` mode is a classic rejection.

Architect Perspective

iOS background execution is the constraint that forces the whole module's design philosophy: because nothing is guaranteed, background becomes a best-effort accelerator over a foreground-reconciled source of truth, triggered primarily by silent push. Behind a `BackgroundService`, the app requests "sync eventually" and the iOS layer maps it to `BGTaskScheduler`/push within budget — while Android gets WorkManager/foreground services. The cross-platform seam absorbs iOS's strictness without leaking it into features (01_background_execution_model.md, 02_workmanager_and_periodic_tasks.md, 05_background_integration.md).

Summary

- iOS background is opportunistic/best-effort: `BGAppRefreshTask` (short), `BGProcessingTask` (charging/overnight), silent push (trigger), background modes (genuine features only).
- Declare identifiers + Background Modes, register at launch, `setTaskCompleted` fast + reschedule, keep work short/idempotent.
- Nothing is guaranteed — always add foreground catch-up; silent push is the most reliable trigger (use sparingly).

Revision Notes

- `BGTaskScheduler` (iOS 13+): `BGAppRefreshTask` (~30s opportunistic), `BGProcessingTask` (charging/overnight); + silent push (`content-available`), background modes (location/audio/VoIP — genuine only).
- Setup: `BGTaskSchedulerPermittedIdentifiers` + Background Modes capability; register at launch; `setTaskCompleted` + reschedule; short/idempotent/storage-backed isolate.
- No guaranteed periodicity; silent push best trigger (throttled); always foreground catch-up; unused modes → rejection.

Practice Questions

1. Why can't you guarantee a periodic iOS background task?
2. What must a BG task handler do before its window ends?
3. What's the most reliable iOS background trigger and its caveat?

Coding Questions

1. Declare permitted identifiers + Background Modes and register a `BGAppRefreshTask` that reschedules itself.
2. Handle a silent push with a short idempotent sync in Flutter.
3. Add a foreground catch-up sync that reconciles missed background runs.

Mini Project

iOS best-effort sync (Flutter + iOS): Set up `BGTaskScheduler` (permitted identifiers + Background Modes), register a self-rescheduling `BGAppRefreshTask` and a charging-gated `BGProcessingTask`, handle a silent-push-triggered sync, and add a foreground catch-up that reconciles state. Keep tasks short/idempotent. Acceptance: identifiers/capabilities declared; tasks register + reschedule + complete fast; silent push triggers sync; foreground catch-up reconciles; no unused background modes; documented as best-effort; runs on device.

Background Integration (Capstone: One Service, Cross-Platform Strategy)

> The maintainable design: expose a single `BackgroundService` whose API is stated in **app intent** ("sync eventually", "track this run"), and let a platform layer map each intent to the right mechanism — **WorkManager / foreground service (Android)**, `BGTaskScheduler` **/ background mode / silent push (iOS)** — all running the **same idempotent work** the foreground uses. Because background is best-effort everywhere, the architecture's backbone is **foreground reconciliation + idempotent, storage-backed tasks**, with background as an *accelerator*, never a dependency.

Introduction

This module capstone unifies the execution model, WorkManager, foreground services, and iOS background into one strategy. The recurring truth — *background is best-effort and platform-divergent* — means the app must never depend on it. This file shows a `BackgroundService` that expresses intent, maps per-platform, shares idempotent logic with the foreground, and reconciles on open.

Why this concept exists

Scattering `workmanager`, foreground-service, and `BGTaskScheduler` calls across features couples them to platform quirks and makes them untestable and unreliable. One service that speaks intent and centralizes the best-effort + reconcile discipline (like a repository for data) isolates the mess and guarantees correctness via the foreground path — consistent with clean architecture (Module 40) and offline-first (Module 19).

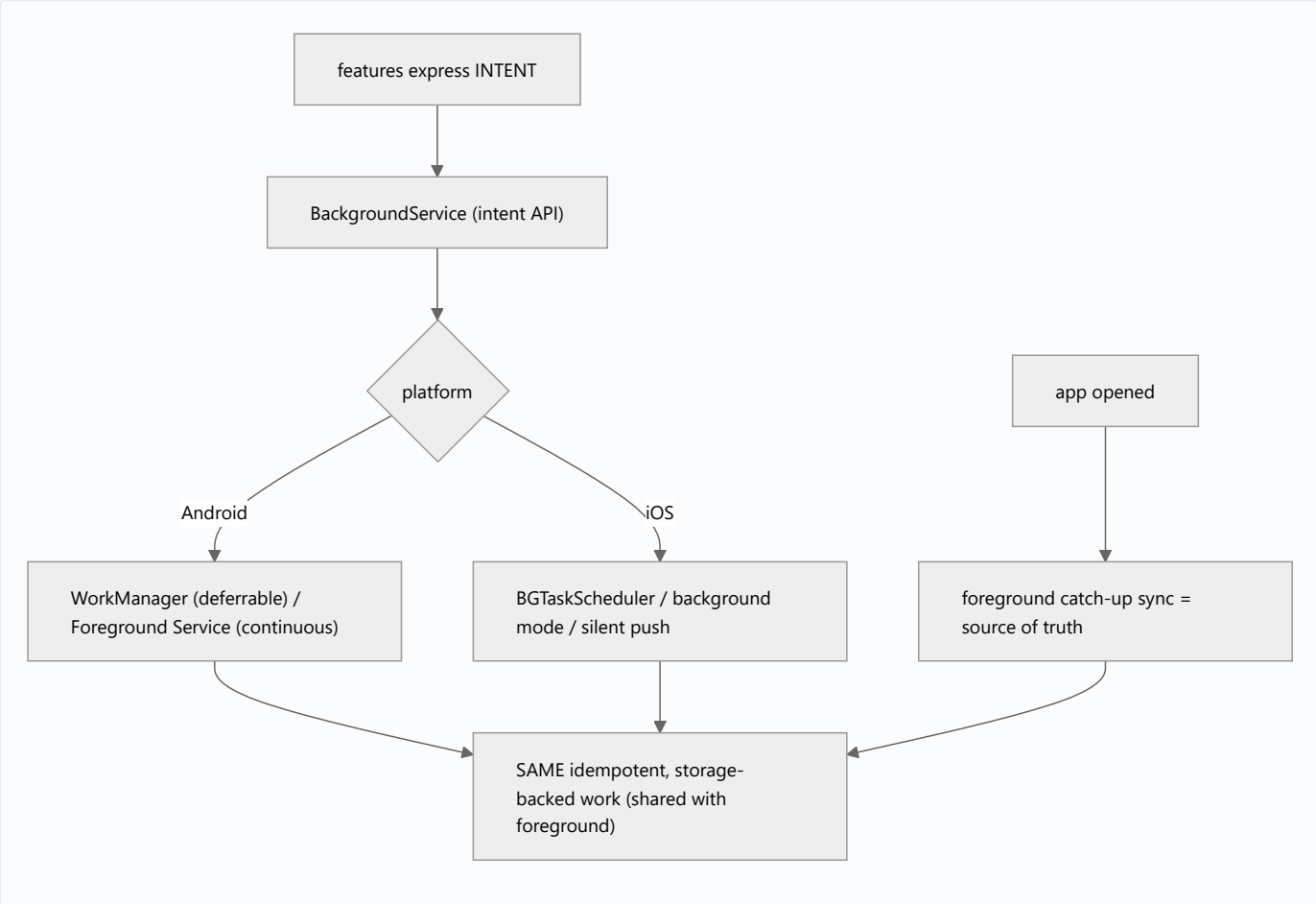
Real-world analogy

`BackgroundService` is a **dispatcher**: teams (features) radio in *what* they need done ("sync the outbox", "track this run"), and the dispatcher picks the right crew for the city they're in (Android vs iOS mechanism). Critically, the **head office reconciles the books when everyone's back** (foreground) — so even if a field crew got delayed by traffic (Doze/budget), nothing is ultimately lost.

Problem Statement

Deliver: periodic outbox sync (WorkManager/ `BGTaskScheduler`) gated on connected+charging, a live run-tracking session (Android foreground service / iOS location background mode), silent-push-triggered sync, and a foreground catch-up that reconciles anything missed — all behind one intent-based `BackgroundService` with graceful degradation. You'll compose every mechanism in this module.

Internal Working



- **Intent-based API**: `scheduleSync()` , `startRunTracking()/stop()` , `syncOnPush()` , `reconcileNow()` . Features don't know the mechanism.
- **Platform mapping** (inside the service): deferrable → **WorkManager (Android)** / `BGTaskScheduler` **(iOS)**; continuous → **foreground service (Android)** / **location background mode (iOS)**; server trigger → **silent push (both)**.
- **Shared idempotent work**: the actual task (e.g., `SyncOutbox`) is **one function** invoked by the foreground, WorkManager, BGTask, and push handler alike — idempotent, resumable, storage-backed. No duplicated logic per mechanism.
- **Foreground reconciliation = source of truth**: on app open (and connectivity return — 29 · connectivity), run the same sync; this guarantees correctness regardless of whether background ran. Background merely makes it *timelier*.

- **Isolate discipline**: every background entry re-inits deps and shares via **storage/ports** (no app state) — the service abstracts this.
- **Graceful degradation**: if background is denied/killed/throttled (OEM/iOS), the app still works via foreground sync; surface nothing broken to the user.
- **Testability**: the service depends on platform-mechanism interfaces (mockable) + the shared task; unit-test that intents map correctly and the shared task is idempotent.

Memory Representation

The service holds scheduling state; the durable work queue/outbox + sync cursors live in storage shared across isolates (Module 15). Background isolates have separate heaps.

Compiler Behavior

Background entry points are top-level `@pragma('vm:entry-point')`; the service otherwise compiles against interfaces (mockable).

Runtime Behavior

The service schedules per platform; background runs opportunistically/continuously as allowed; the shared idempotent task runs in whichever context fires; foreground reconciliation converges state on open. Missed/duplicated runs are safe (idempotent).

Flutter Engine Behavior

Multiple background engines/isolates (WorkManager, foreground service, BGTask, push) plus the UI isolate — the service centralizes their entry points; all funnel to the same task.

Dart VM Behavior

Several isolates, no shared memory; coordination via storage. Keep background work short + idempotent so any isolate can run it safely.

Examples

```
// Intent-based API — features never touch platform mechanisms
abstract class BackgroundService {
  Future<void> scheduleSync();            // deferrable/periodic
  Future<void> startRunTracking();        // continuous
  Future<void> stopRunTracking();
  Future<void> reconcileNow();            // foreground catch-up (source of truth)
```

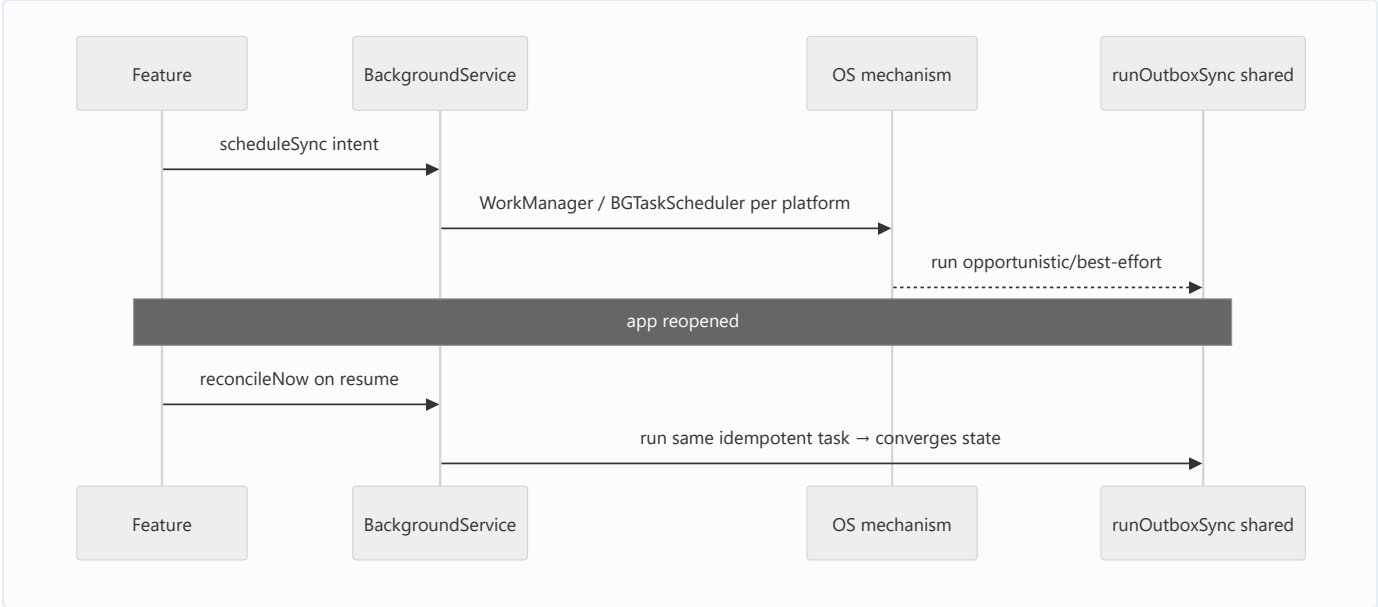
```
}

// ONE idempotent task, shared by every trigger (fg, WorkManager, BGTask, push)
@pragma('vm:entry-point')
Future<bool> runOutboxSync() async {
  // re-init deps (isolate-safe); read outbox from storage; push each op idempotently;
  // mark synced; safe to re-run / resume. Returns success for retry semantics.
  return true;
}

// Android implementation maps intent -> mechanism
class AndroidBackgroundService implements BackgroundService {
  @override
  Future<void> scheduleSync() => workmanager.registerPeriodicTask(
        'sync', 'sync', frequency: const Duration(minutes: 15),
        constraints: Constraints(networkType: NetworkType.connected, requiresCharging: true));
  @override
  Future<void> startRunTracking() => foregroundService.start(callback: startCallback);
  @override
  Future<void> stopRunTracking() => foregroundService.stop();
  @override
  Future<void> reconcileNow() => runOutboxSync(); // same task, foreground
}
// iOS impl maps to BGTaskScheduler / background mode / silent push, same runOutboxSync().

// App bootstrap + on-resume reconcile (correctness guarantee)
// WidgetsBinding: on AppLifecycleState.resumed -> backgroundService.reconcileNow();
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Depending on background running	Best-effort/throttled/OEM-killed	Foreground reconciliation as source of truth
Duplicated logic per mechanism	Drift/bugs	One shared idempotent task
Platform calls in features	Coupling, untestable	Intent API + platform mapping in service
Non-idempotent shared task	Duplicates on re-run/retry	Make idempotent + resumable
No graceful degradation	Feature "breaks" without background	Foreground fallback, silent to user
Same expectations both platforms	iOS far stricter	Per-platform mapping + iOS best-effort

Best Practices

- Expose an **intent-based** `BackgroundService`; map to platform mechanisms **inside** it (WorkManager/foreground service; `BGTaskScheduler`/background mode/silent push).
- Run **one idempotent, storage-backed task** across all triggers; make **foreground reconciliation the source of truth** (background = accelerator).
- **Degrade gracefully** (denied/killed/throttled → foreground sync); keep background work short/resumable; re-init deps in isolates.
- Depend on **interfaces** for testability; verify **intent→mechanism mapping** and **task idempotency**; set **iOS best-effort expectations**.

Performance

Constraint-gated background (charging/wifi) + short idempotent tasks minimize battery; foreground reconciliation adds negligible cost on open. The architecture avoids wasted/duplicated work and OEM-killer fragility by never depending on background timing.

Advantages / Disadvantages

- **+** Correct regardless of background reliability (foreground truth), testable, cross-platform, no duplicated logic, graceful degradation.
- **−** Upfront structure/boilerplate, per-platform mapping, must design idempotency + reconciliation, multi-isolate coordination.

Interview Questions

1. 🟢 **Why should background never be a dependency?** — It's best-effort and platform-divergent (Doze/budgets/OEM killers); correctness must come from foreground reconciliation, with background as an accelerator.
2. 🟢 **What does an intent-based `BackgroundService` API look like?** — Methods like `scheduleSync` / `startRunTracking` / `reconcileNow` — features express *what*, the service picks the platform mechanism.
3. 🟡 **Why share one idempotent task across triggers?** — To avoid logic drift and make foreground, WorkManager, BGTask, and push all converge to the same correct state safely.
4. 🟡 **How do intents map per platform?** — Deferrable → WorkManager/`BGTaskScheduler`; continuous → foreground service/background mode; server → silent push.
5. 🟡 **How do you guarantee correctness despite unreliable background?** — Run the same idempotent sync on app resume/connectivity return — foreground is the source of truth.
6. 🔴 **How is this tested without devices?** — Interfaces for platform mechanisms + a pure idempotent task; assert intent→mechanism mapping and re-run safety with fakes.
7. 🔴 **How does this integrate with offline-first?** — The shared task flushes the outbox; connectivity/foreground trigger reconciliation; background just flushes sooner (Module 19).

Senior Engineer Tips

- Build the shared idempotent task + foreground reconciliation first; add background triggers last as accelerators — this ordering makes the feature correct before it's optimized.
- Keep every platform call behind the service; features asking for a "foreground service" or "BGTask" directly is the smell that guarantees untestable, brittle code.
- Set explicit expectations: on iOS (and OEM-crippled Android), background "sometimes" runs — the demo of reliability is the resume-reconcile path, and that's what you test.

Architect Perspective

Background integration is the synthesis of the module's core lesson: because the OS owns *when* and gives no guarantees, the app owns *what* (one idempotent, storage-backed task) and *correctness* (foreground reconciliation), while a thin platform layer maps intents to WorkManager/foreground-service/`BGTaskScheduler`/push. This yields features that are reliable, testable, cross-platform, and battery-respecting — background accelerates but never gates — fitting cleanly into offline-first and clean-architecture boundaries (Module 19, Module 40).

Summary

- One intent-based `BackgroundService` maps to per-platform mechanisms; features express *what*, not *how*.
- A single idempotent, storage-backed task runs across all triggers; **foreground reconciliation is the source of truth**, background an accelerator.
- Degrade gracefully, keep tasks short/resumable, test via interfaces + idempotency; expect iOS best-effort.

Revision Notes

- Intent API (`scheduleSync` / `startRunTracking` / `reconcileNow`); map inside service: Android WorkManager/foreground service, iOS `BGTaskScheduler`/background mode/silent push.
- Shared idempotent `runOutboxSync` for all triggers; foreground reconcile (on resume/connectivity) = source of truth; background = accelerator.
- Graceful degradation; short/resumable/storage-backed isolate work; interfaces + idempotency for tests; iOS best-effort.

Practice Questions

1. Why is foreground reconciliation the source of truth?
2. How does one intent map to different platform mechanisms?
3. Why must the shared task be idempotent?

Coding Questions

1. Define an intent-based `BackgroundService` + Android/iOS implementations.
2. Implement one idempotent `runOutboxSync` used by all triggers.
3. Wire foreground reconciliation on app resume + connectivity return.

Mini Project

Cross-platform background sync (capstone — Flutter): Build a `BackgroundService` with an intent API (`scheduleSync` , `startRunTracking` / `stop` , `syncOnPush` , `reconcileNow`) mapping to WorkManager/foreground service (Android) and `BGTaskScheduler` /background mode/silent push (iOS), all invoking one idempotent, storage-backed `runOutboxSync` . Add foreground reconciliation on resume/connectivity as the source of truth, with graceful degradation. Provide interfaces + tests. Acceptance: features use intents only; one shared idempotent task across triggers; foreground reconciliation guarantees correctness; degrades when background is denied/killed; per-platform mapping correct; unit-tested (mapping + idempotency); documented iOS best-effort.