32 · Notifications

Flutter Practice Notes

Contents

32 · Notifications

Introduction

This module covers notifications in Flutter: **local notifications** (`flutter_local_notifications` — scheduled/immediate, channels, actions), **push via Firebase Cloud Messaging** (FCM — tokens, the foreground/background/terminated delivery model), **message types & targeting** (notification vs data messages, topics, device tokens), and **handling taps + deep links** (routing a tapped notification to the right screen), tied together in a capstone. It builds on Firebase (Module 18), permissions (27/28), and deep linking (Module 13).

Why this module exists

Notifications drive re-engagement and deliver time-sensitive info, but they're deceptively tricky: platform permission models differ, FCM behaves differently across app states (foreground/background/terminated), **notification** vs **data** messages route differently, and a tapped notification must deep-link reliably even from a cold start. Getting these right is what separates flaky notifications from dependable ones.

Module map

#	File	Topic	Level
1	01_local_notifications.md	`flutter_local_notifications` : immediate/scheduled, channels, actions	🔵
2	02_fcm_push.md	FCM setup, tokens, foreground/background/terminated delivery	🔴
3	03_message_types_and_targeting.md	Notification vs data messages, topics, token/segment targeting	🔴
4	04_handling_and_deeplinks.md	Tap handling, cold-start routing, deep links, foreground display	🔴
5	05_notifications_integration.md	Capstone: FCM + local + deep-link routing behind a service	🔴

> **Cross-references:** Firebase project/setup: Module 18. Permissions: 27 · permissions, 28 · infoplist, 28 · ios_integration (APNs/background modes). Deep linking/routing: Module 13. Background execution: Module 33. Backend send: Module 16.

Prerequisites

18 Firebase (project + FlutterFire), 27/28 (permissions, APNs/entitlements), 13 Routing (go_router for deep links).

What you'll be able to do after this module

- Show immediate and scheduled local notifications with channels and actions.
- Set up FCM, obtain/refresh device tokens, and handle all three app states.
- Choose and send notification vs data messages; target by token/topic.
- Route notification taps (including cold start) to the right screen via deep links.
- Wrap it all behind a notification service that's testable and permission-aware.

Capstone

Notification slice: FCM push received in foreground (shown via local notification), background, and terminated states; data payloads routed through `go_router` to a target screen on tap (including cold start); topic subscription for broadcasts; and scheduled local reminders — all behind a `NotificationService`.

Summary

Notifications = local (`flutter_local_notifications`) + push (FCM), with careful handling of permissions, the foreground/background/terminated model, notification-vs-data messages, and deep-link routing on tap. Wrap it behind a service and let the server own targeting/sending.

Local Notifications (`flutter_local_notifications`)

> Local notifications are scheduled/shown **by the app itself** (no server) via `flutter_local_notifications`: initialize per-platform settings, define **Android channels** (importance/sound — mandatory on modern Android), request permission (Android 13+ / iOS), then `show()` immediately or `zonedSchedule()` for the future — attaching a **payload** you read back when the user taps to route them.

Introduction

Local notifications handle reminders, alarms, timers, and "downloaded/complete" alerts — anything the app can decide on-device without a server. This file covers initialization, channels, permissions, immediate vs scheduled, actions, and the tap payload used for routing (04_handling_and_deeplinks.md).

Why this concept exists

Not every notification needs a server round-trip. Reminders and scheduled alerts are known on-device, so the OS notification system can display them locally. `flutter_local_notifications` wraps the divergent Android/iOS notification APIs (channels, scheduling, timezones) into one Dart API.

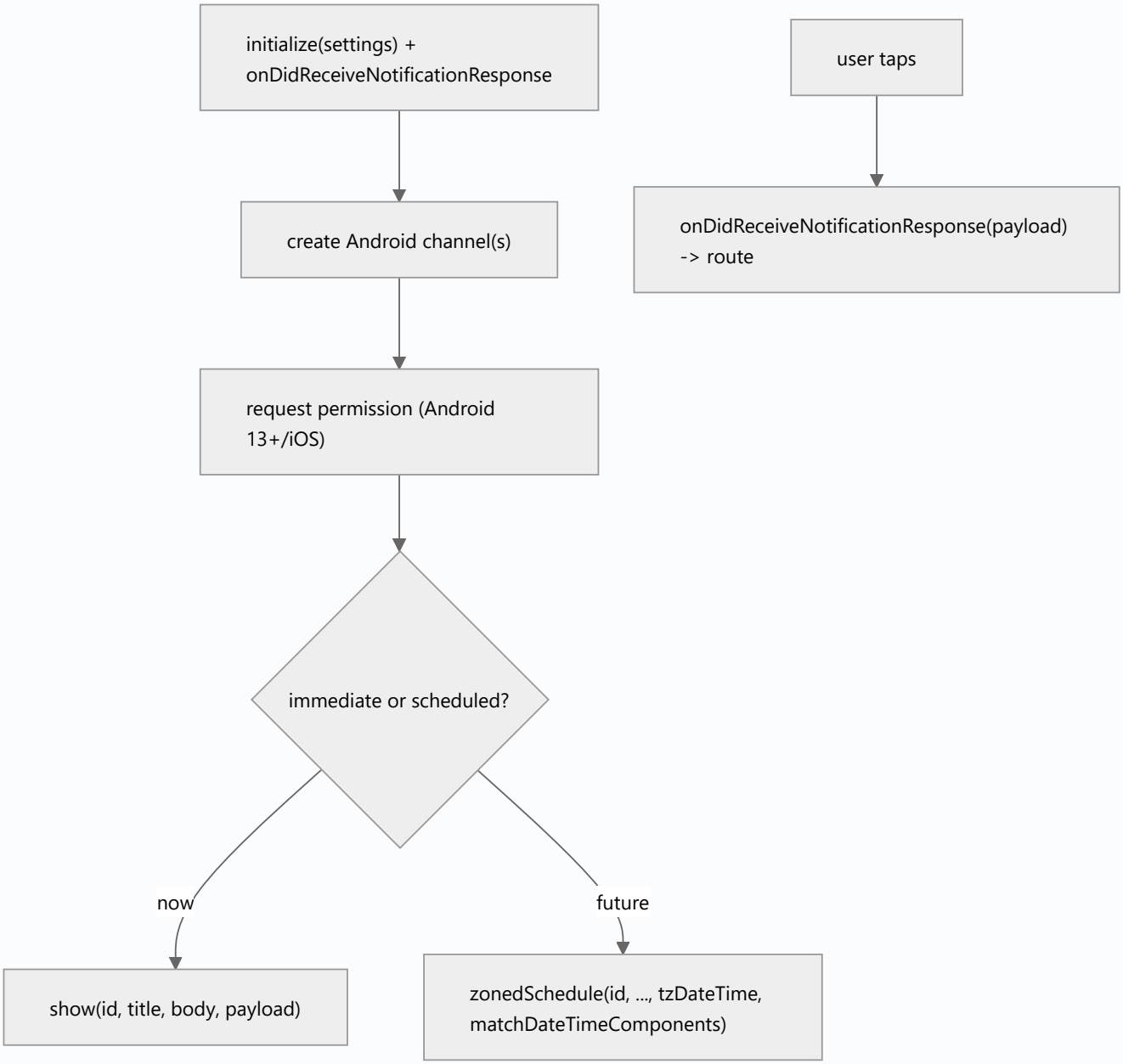
Real-world analogy

A local notification is a **sticky note you set for your future self**: you decide the message and the time, and the OS (your assistant) posts it on the fridge at that moment. No one else is involved — unlike push, where a **courier delivers a note from outside** (02_fcm_push.md).

Problem Statement

Show a "backup complete" alert now and a "drink water" reminder every day at 9am, with an Android channel, permission handling, and a payload that opens the right screen on tap. You'll initialize the plugin, define a channel, and schedule.

Internal Working



- **Initialize**: `AndroidInitializationSettings('@mipmap/ic_launcher')` + `DarwinInitializationSettings(...)`; pass `onDidReceiveNotificationResponse` (tap handler with the payload). Call once at startup.
- **Android channels** (mandatory ≥ Android 8): create `AndroidNotificationChannel(id, name, importance: Importance.high, ...)`; importance/sound are set on the **channel**, not per-notification (can't change after creation). Group related notifications by channel.
- **Permissions**: **Android 13+** needs the `POST_NOTIFICATIONS` runtime permission; **iOS** needs `requestPermissions(alert/badge/sound)`. Handle denial gracefully (27/28).
- **Immediate**: `show(id, title, body, NotificationDetails(...), payload: 'route:/order/42')`. **Unique id** per notification (reusing an id replaces it — useful for progress).

- **Scheduled**: `zonedSchedule(id, title, body, tz.TZDateTime, details, androidScheduleMode: exactAllowWhileIdle, matchDateTimeComponents: DateTimeComponents.time)` for daily/weekly repeats. Requires the `timezone` package (init `tz`); **exact alarms** need a special permission on Android 12+.
- **Actions & styles**: action buttons (`AndroidNotificationAction`), big-text/big-picture/inbox styles, grouping, ongoing/progress notifications.
- **Payload → routing**: the string payload is delivered to the tap handler; parse it to deep-link (04_handling_and_deeplinks.md).

Memory Representation

Notifications live in the OS notification system, not app memory. The plugin keeps init settings + pending scheduled notifications (queryable via `pendingNotificationRequests`).

Compiler Behavior

Not applicable; channels/permissions are runtime + manifest config.

Runtime Behavior

Scheduled notifications fire even if the app is killed (OS-driven) — but OEM battery optimizations can delay/drop them; exact timing isn't guaranteed without exact-alarm permission. Reusing an id updates the existing notification.

Flutter Engine Behavior

Tapping while terminated cold-starts the app; retrieve the launch details (`getNotificationAppLaunchDetails()`) to route on startup (04_handling_and_deeplinks.md).

Dart VM Behavior

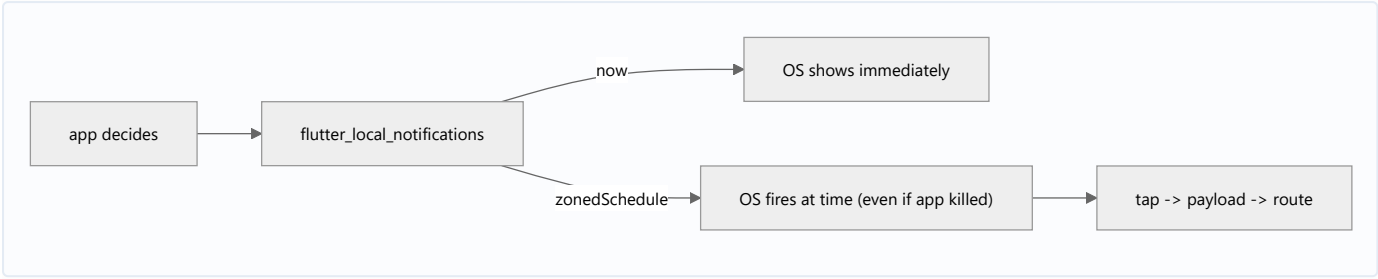
Not applicable.

Examples

```
import 'package:flutter_local_notifications/flutter_local_notifications.dart';
import 'package:timezone/data/latest.dart' as tzdata;
import 'package:timezone/timezone.dart' as tz;

final _plugin = FlutterLocalNotificationsPlugin();
const _channel = AndroidNotificationChannel(
```

```
  'reminders', 'Reminders', importance: Importance.high,
);

Future<void> initLocalNotifications(void Function(String? payload) onTap) async {
  tzdata.initializeTimeZones();
  await _plugin.initialize(
    const InitializationSettings(
      android: AndroidInitializationSettings('@mipmap/ic_launcher'),
      iOS: DarwinInitializationSettings(),
    ),
    onDidReceiveNotificationResponse: (r) => onTap(r.payload), // tap -> route
  );
  await _plugin
      .resolvePlatformSpecificImplementation<AndroidFlutterLocalNotificationsPlugin>()
      ?.createNotificationChannel(_channel);              // mandatory channel
}

Future<void> showNow(int id, String title, String body, String route) =>
    _plugin.show(id, title, body,
        const NotificationDetails(android: AndroidNotificationDetails('reminders', 'Reminders')),
        payload: route);                                  // payload used on tap

Future<void> scheduleDaily9am(int id) => _plugin.zonedSchedule(
      id, 'Drink water', 'Time for a glass 💧',
      _next9am(),
      const NotificationDetails(android: AndroidNotificationDetails('reminders', 'Reminders')),
      androidScheduleMode: AndroidScheduleMode.exactAllowWhileIdle,
      matchDateTimeComponents: DateTimeComponents.time,   // repeat daily
      payload: 'route:/hydration',
    );

tz.TZDateTime _next9am() {
  final now = tz.TZDateTime.now(tz.local);
  var t = tz.TZDateTime(tz.local, now.year, now.month, now.day, 9);
  if (t.isBefore(now)) t = t.add(const Duration(days: 1));
  return t;
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
No Android channel	Notifications silently dropped (≥ O)	Create channel before showing
Changing channel importance later	Ignored after creation	Set at creation; new channel to change
Not requesting Android 13+/iOS permission	No notifications shown	Request + handle denial
Naive `DateTime` scheduling	DST/timezone bugs	Use `timezone` `TZDateTime`
Reusing ids unintentionally	Notifications overwrite	Unique ids (reuse only for updates)
Expecting exact timing always	OEM battery limits/exact-alarm perm	Use exact-alarm mode/perm; tolerate drift
No payload	Can't route on tap	Attach + parse payload

Best Practices

- **Initialize once** at startup with a tap handler; **create Android channels** up front (importance/sound are per-channel and immutable).
- **Request permission** (Android 13+ `POST_NOTIFICATIONS`, iOS) and degrade gracefully; use `timezone` `TZDateTime` for scheduling (DST-safe).
- Use **unique ids** (reuse only to update/progress); attach a **payload** and route on tap (incl. cold start via launch details).
- For exact/repeating alerts use the right **schedule mode**/exact-alarm permission; tolerate OEM delivery drift.

Performance

Negligible; the OS handles display/scheduling. Avoid spamming notifications (annoyance + OS throttling). `pendingNotificationRequests` lets you audit/cancel.

Advantages / Disadvantages

- + No server needed, works offline, scheduled/repeating reminders, channels/actions/styles, fires when app killed.
- − Per-platform channels/permissions, timezone/exact-alarm gotchas, OEM battery-optimization unreliability, only for on-device-known events.

Interview Questions

1. 🟢 **What are local notifications for?** — App-decided, on-device alerts (reminders/alarms/complete) shown/scheduled without a server.
2. 🟢 **Why are Android channels required?** — On Android 8+ every notification needs a channel (which owns importance/sound); without one it's dropped.
3. 🟡 **How do you schedule a reliable daily notification?** — `zonedSchedule` with a `timezone` `TZDateTime` + `matchDateTimeComponents: time`, appropriate schedule mode/exact-alarm permission.
4. 🟡 **Why use `TZDateTime` instead of `DateTime`?** — To avoid DST/timezone bugs; scheduling must be timezone-aware.
5. 🟡 **What permissions do local notifications need?** — Android 13+ `POST_NOTIFICATIONS` runtime permission; iOS alert/badge/sound authorization.
6. 🔴 **How do you route when the user taps a notification?** — Attach a payload; handle it in `onDidReceiveNotificationResponse` (and `getNotificationAppLaunchDetails` for cold start) to deep-link.
7. 🔴 **Why might scheduled notifications not fire on time?** — OEM battery optimizations/doze and lack of exact-alarm permission can delay/drop them.

Senior Engineer Tips

- Define channels deliberately (by importance/purpose) at first run; you can't change a channel's importance later, only recreate it.
- Always schedule with the `timezone` package — naive `DateTime` scheduling is a recurring DST/off-by-hours bug.
- Attach a structured payload (a route/deep link) from day one and handle cold-start launch details — retrofitting tap routing is painful.

Architect Perspective

Local notifications are an on-device scheduling + OS-integration concern. Wrapping init/channels/permissions/scheduling behind a `NotificationService` (with structured payloads feeding deep-link routing) keeps reminders reliable and testable, and unifies cleanly with push

(which also renders via the same local plugin in the foreground — 02_fcm_push.md, 05_notifications_integration.md).

Summary

- `flutter_local_notifications`: init once (+ tap handler), create Android channels, request permission, `show()` / `zonedSchedule()` with `TZDateTime`.
- Unique ids, structured payloads for tap routing (incl. cold start); tolerate OEM delivery drift; exact-alarm perms for precise timing.
- On-device, offline, no server — for app-known events; wrap behind a service.

Revision Notes

- Init `FlutterLocalNotificationsPlugin` + `onDidReceiveNotificationResponse`; create `AndroidNotificationChannel` (importance immutable).
- Permissions: Android 13+ `POST_NOTIFICATIONS`, iOS request; `show(id,...,payload)` (unique id), `zonedSchedule(TZDateTime, matchDateTimeComponents)` with `timezone`.
- Tap → payload → route (+ `getNotificationAppLaunchDetails` cold start); OEM battery limits/exact-alarm affect timing.

Practice Questions

1. Why must you create an Android channel, and what does it own?
2. How do you schedule a DST-safe daily reminder?
3. How do you route the user when they tap a local notification?

Coding Questions

1. Initialize the plugin with a channel + tap handler and show an immediate notification with a payload.
2. Schedule a daily 9am reminder using `TZDateTime`.
3. Handle a cold-start tap via `getNotificationAppLaunchDetails`.

Mini Project

Reminders (Flutter): Build a `LocalNotificationService` that initializes the plugin (+ channel + permission), shows an immediate "task done" notification, schedules a daily 9am reminder (timezone-aware, repeating), and routes taps via payload (including cold start). Acceptance: channel created; permission requested + handled; immediate + daily-repeating notifications fire; tap deep-links via payload (incl. terminated launch); behind a service; runs on device.

Push Notifications with FCM (Foreground / Background / Terminated)

> Firebase Cloud Messaging delivers server-sent push to a device identified by an **FCM registration token** (which rotates — sync it to your backend). The hard part is the **three app-state delivery model**: **foreground** (you receive `onMessage` and must render it yourself, e.g. via a local notification), **background** (the system tray shows it; `onMessageOpenedApp` fires on tap), and **terminated** (a top-level background handler + `getInitialMessage()` on launch) — plus iOS APNs setup.

Introduction

FCM is the standard push service for Flutter (via `firebase_messaging`). This file covers setup (Firebase + APNs), tokens (get/refresh/sync), permission, and — the crux — how message reception differs across foreground, background, and terminated states. Message *content* (notification vs data) is in 03_message_types_and_targeting.md; tap routing in 04_handling_and_deeplinks.md.

Why this concept exists

Servers need to reach devices that aren't running. FCM (backed by APNs on iOS) provides OS-level push delivery. Flutter's `firebase_messaging` exposes it, but because the app may be foreground, backgrounded, or killed, the callbacks and responsibilities differ per state — the single biggest source of "push works sometimes" bugs.

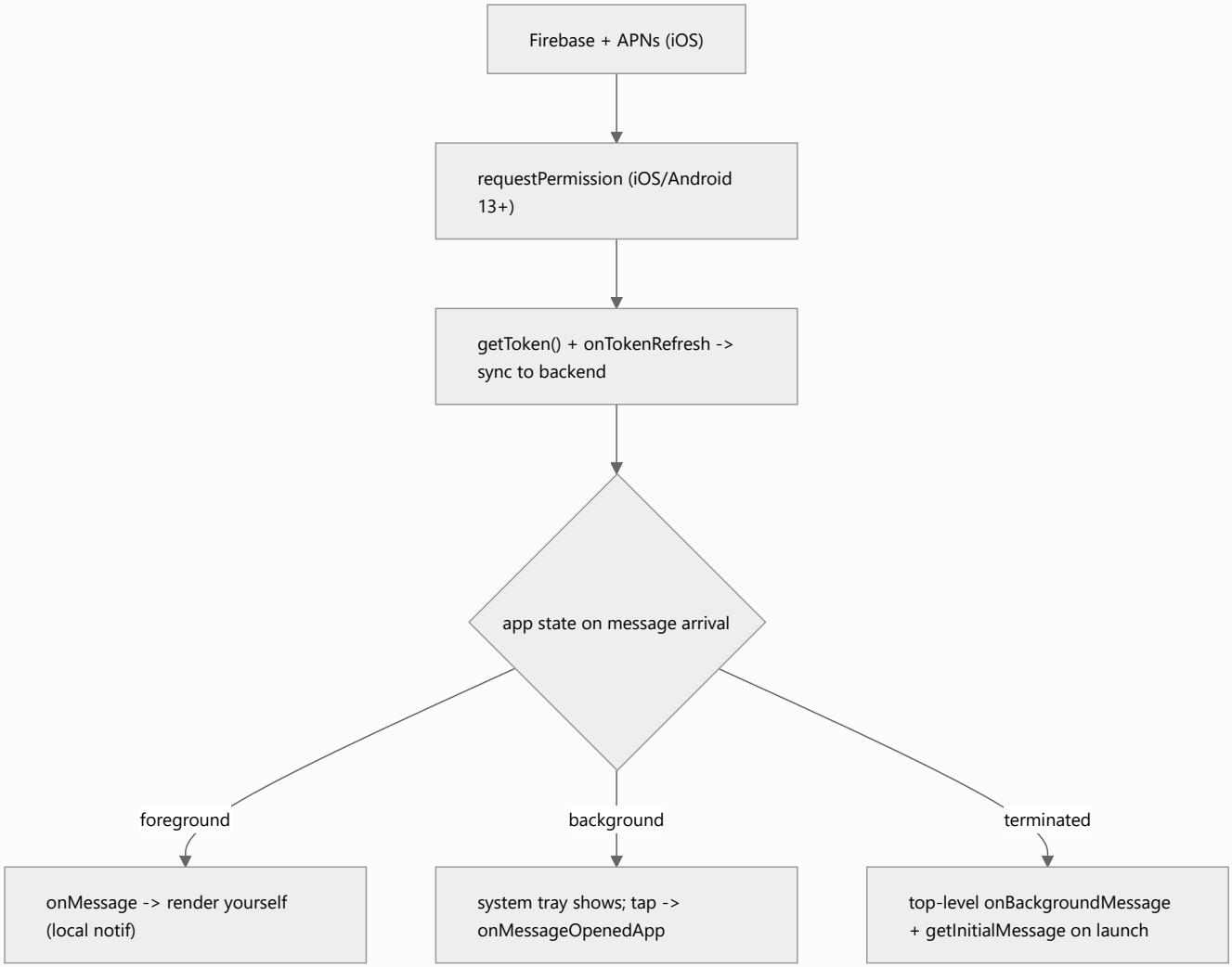
Real-world analogy

FCM is the **postal service**; the **FCM token is your mailing address** (which can change — you must forward it to senders). When you're **home and awake** (foreground) the courier hands you the letter and *you* decide to display it; when you're **out** (background) they drop it in the mailbox (system tray); when the **house is empty** (terminated) it waits in the box and you read it when you get home (on launch).

Problem Statement

Receive push in all three states: show it while the app is open, let the tray show it while backgrounded, and handle it on cold start after a tap — with tokens synced to your backend and iOS APNs configured. You'll wire `firebase_messaging` handlers.

Internal Working



- **Setup**: FlutterFire (Module 18); **iOS** additionally needs **APNs** — Push Notifications capability + an APNs key/cert in Firebase, and background modes (28 · ios_integration). Push doesn't work on iOS Simulator.
- **Permission**: `FirebaseMessaging.instance.requestPermission()` (iOS prompt; Android 13+ `POST_NOTIFICATIONS`). Check `authorizationStatus`.
- **Token**: `getToken()` returns the registration token; `onTokenRefresh` fires when it rotates. **Send/refresh the token to your backend** (tied to the user) so it can target the device; delete on logout.
- **Three states (crucial)**:
  - **Foreground**: `FirebaseMessaging.onMessage` fires. The OS does **not** show a tray notification for you (on Android, and iOS unless configured) — **you render it**, typically via `flutter_local_notifications` (01_local_notifications.md).
  - **Background (app alive, not foreground)**: the system shows the **notification** in the tray automatically (for messages with a `notification` block); tapping fires `onMessageOpenedApp`.

    - **Terminated (killed)**: a **top-level** `onBackgroundMessage` **handler** (a top-level/static function, registered before `runApp`) processes **data** in the background; a tap that launches the app is retrieved via `getInitialMessage()` at startup.

- **Background handler rules**: must be a **top-level or static** function annotated `@pragma('vm:entry-point')`, runs in a **separate isolate** (no UI/most singletons) — keep it minimal (e.g., show a local notification, update storage).
- **iOS foreground presentation**: `setForegroundNotificationPresentationOptions` to let iOS show banners in foreground (else you render).

Memory Representation

The token is a string synced to your backend. Background messages run in an isolate with its own memory — no access to your app's live state.

Compiler Behavior

The background handler must be a top-level/static entry point (`@pragma('vm:entry-point')`) so AOT keeps it.

Runtime Behavior

Foreground → `onMessage`; background → tray + `onMessageOpenedApp` on tap; terminated → background isolate handler + `getInitialMessage()` on launch. Delivery is best-effort (not guaranteed/instant); OEM/doze can delay.

Flutter Engine Behavior

The background message handler runs in a **dedicated background isolate/engine** — separate from the UI isolate (02 · isolates); don't touch UI/providers there.

Dart VM Behavior

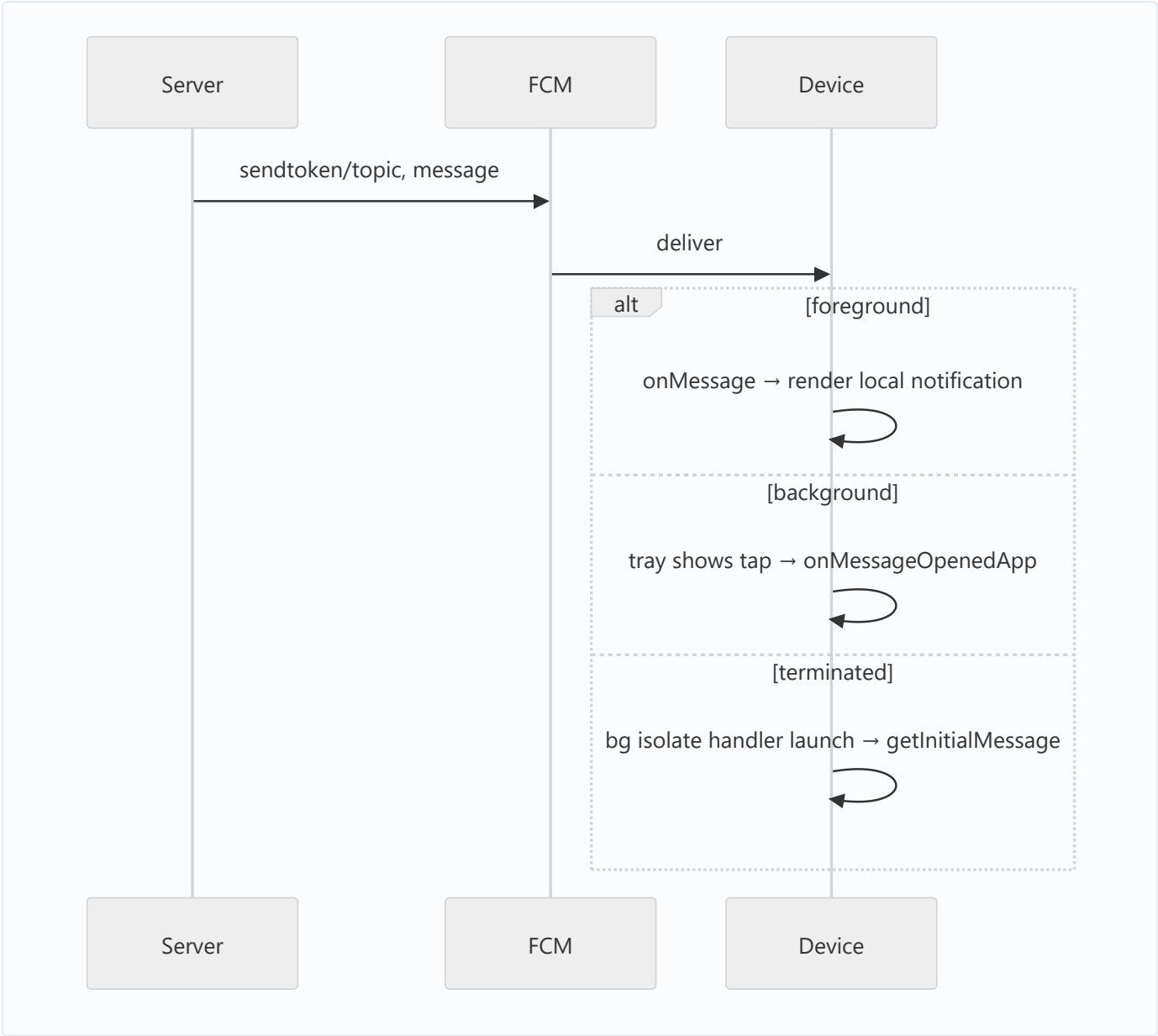
Two isolates: your UI isolate (foreground handlers) and the background isolate (terminated/background data handler). They don't share memory.

Examples

```
import 'package:firebase_messaging/firebase_messaging.dart';
import 'package:firebase_core/firebase_core.dart';
```

```
// TERMINATED/BACKGROUND: top-level, separate isolate — keep minimal
@pragma('vm:entry-point')
Future<void> _bgHandler(RemoteMessage message) async {
  await Firebase.initializeApp();
  // e.g., show a local notification / update local store. NO UI, NO app singletons.
}

Future<void> initFcm({
  required void Function(RemoteMessage) onForeground,
  required void Function(RemoteMessage) onOpened,
}) async {
  final fm = FirebaseMessaging.instance;
  await fm.requestPermission();                              // iOS/Android 13+
  FirebaseMessaging.onBackgroundMessage(_bgHandler);         // register BEFORE runApp

  // Token: get + keep synced to backend
  final token = await fm.getToken();
  await api.syncFcmToken(token);
  fm.onTokenRefresh.listen(api.syncFcmToken);                // token rotates

  // FOREGROUND: you must render it yourself (e.g., local notification)
  FirebaseMessaging.onMessage.listen(onForeground);

  // BACKGROUND tap: app resumes to this message
  FirebaseMessaging.onMessageOpenedApp.listen(onOpened);

  // TERMINATED tap that launched the app
  final initial = await fm.getInitialMessage();
  if (initial != null) onOpened(initial);
}
```

Diagrams


Server
FCM
Device
sendtoken/topic, message
deliver
alt
[foreground]
onMessage → render local notification
[background]
tray shows tap → onMessageOpenedApp
[terminated]
bg isolate handler launch → getInitialMessage
Server
FCM
Device

Common Mistakes

Mistake	Why	Fix
Expecting a tray notification in foreground	OS doesn't auto-show	Render yourself (local notification)
Background handler not top-level/ `vm:entry-point`	Not invoked / stripped	Top-level/static + `@pragma('vm:entry-point')`
Touching UI/singletons in bg handler	Separate isolate, no access	Keep minimal; use storage/local notif
Not syncing/refreshing token	Can't target device; breaks on rotation	Sync `getToken` + `onTokenRefresh`
Ignoring `getInitialMessage`	Terminated taps lost	Handle on startup
Missing iOS APNs setup	No push on iOS	Configure APNs key + capability
Testing push on iOS Simulator	Unsupported	Use a real device

Best Practices

- **Register** `onBackgroundMessage` **before** `runApp` with a top-level `@pragma('vm:entry-point')` handler; keep it minimal (no UI/singletons).
- Handle **all three states**: `onMessage` (render yourself in foreground), tray+ `onMessageOpenedApp` (background tap), `getInitialMessage` (terminated launch).
- **Sync the token** (`getToken` + `onTokenRefresh`) to your backend tied to the user; **delete on logout**; request permission + handle denial.
- Configure **iOS APNs** (key + capability + background modes — 28 · ios_integration); test on **real devices**; wrap behind a service.

Performance

Push is event-driven (no polling — battery-friendly). The background isolate spins up briefly per message — keep the handler fast. Delivery is best-effort; don't rely on instant/guaranteed timing.

Advantages / Disadvantages

- **+** Reach devices when not running, battery-efficient (OS push), topic/token targeting, cross-platform via FCM/APNs.
- **−** Complex three-state model, background-isolate constraints, iOS APNs setup, best-effort delivery, token lifecycle management.

Interview Questions

1. 🟢 **What identifies a device for push?** — Its FCM registration token (which rotates — sync `getToken` / `onTokenRefresh` to your backend).
2. 🟢 **What happens to a push while the app is in the foreground?** — `onMessage` fires; the OS doesn't show a tray notification, so you render it yourself (e.g., local notification).
3. 🟡 **How is a terminated-state message handled?** — A top-level `@pragma('vm:entry-point')` background handler (separate isolate) processes data; a launching tap is read via `getInitialMessage()`.
4. 🟡 **Why can't the background handler touch app state?** — It runs in a separate background isolate with no shared memory/UI; keep it minimal (storage/local notification).
5. 🟡 **What's needed for iOS push?** — APNs key/cert in Firebase + Push Notifications capability/background modes; a real device (not Simulator).
6. 🔴 **Differentiate `onMessageOpenedApp` vs `getInitialMessage`.** — Both handle taps that open the app: `onMessageOpenedApp` when resuming from background; `getInitialMessage` for the tap that cold-started a terminated app.
7. 🔴 **Why sync and delete tokens?** — The backend needs the current token to target the device; tokens rotate (refresh) and should be removed on logout to stop delivery to that user.

Senior Engineer Tips

- Wire all three states on day one and test each explicitly (foreground, backgrounded, force-killed) — "push works" usually means only one state was tested.
- Treat the token as user-scoped mutable state: sync on login/refresh, delete on logout; stale tokens cause misrouted or lost notifications.
- Keep the background isolate handler tiny and self-contained (init Firebase, show a local notification, write storage) — it can't reach your DI/UI.

Architect Perspective

FCM is an event-driven, multi-isolate delivery system with a state-dependent contract. A `PushService` that centralizes permission, token sync, the three-state handlers, and foreground rendering (delegating to the local-notifications service) turns a notoriously flaky area into a testable, reliable one — with the backend owning who/when to send (03_message_types_and_targeting.md, 05_notifications_integration.md, Module 18).

Summary

- FCM push targets a rotating **token** (sync to backend); iOS needs APNs; test on real devices.

- Handle three states: **foreground** (`onMessage` , render yourself), **background** (tray + `onMessageOpenedApp`), **terminated** (top-level bg isolate handler + `getInitialMessage`).
- Background handler must be top-level `@pragma('vm:entry-point')` , minimal, no UI/singletons; wrap behind a service.

Revision Notes

- `firebase_messaging` : `requestPermission` , `getToken` / `onTokenRefresh` (sync + delete on logout), APNs for iOS, real device.
- Foreground `onMessage` (render via local notif); background tray + `onMessageOpenedApp` ; terminated `onBackgroundMessage` (top-level `@pragma('vm:entry-point')` , isolate) + `getInitialMessage` .
- Best-effort delivery; bg handler minimal (no UI/singletons); register `onBackgroundMessage` before `runApp` .

Practice Questions

1. What happens to a push in each of the three app states?
2. Why must the background handler be top-level and minimal?
3. How do you keep the device targetable over time?

Coding Questions

1. Initialize FCM with permission, token sync, and all three state handlers.
2. Write a compliant top-level background message handler.
3. Handle a terminated-state launch via `getInitialMessage` .

Mini Project

Push receiver (Flutter + Firebase): Build a `PushService` that requests permission, syncs the FCM token (+ refresh) to a backend, and handles push in foreground (rendered via the local-notifications service), background (tray + open handler), and terminated (top-level isolate handler + `getInitialMessage`). Configure iOS APNs. Acceptance: token synced + refreshed; all three states handled + tested on a real device; foreground push rendered locally; background handler minimal + compliant; behind a service.

Message Types & Targeting (Notification vs Data, Topics, Tokens)

> FCM messages come in two shapes with very different behavior: **notification messages** (a `notification` block the OS auto-displays in background/terminated — convenient but you don't control display, and your Dart handler *doesn't run* on tap-less background delivery) vs **data messages** (custom key/value only — *your* code always handles them, giving full control but you must render). For reliable, routable push, most apps send **data-only** (or notification+data) and target by **device token** (one user), **topic** (broadcast/segments), or **condition**.

Introduction

Choosing the message type is the second-biggest push decision after the three-state model. It determines who displays the notification (OS vs your app) and whether your handler runs. This file explains both types, the recommended hybrid, and the targeting methods (token/topic/condition) — all sent from your **server**, never the client.

Why this concept exists

FCM supports a "just show this" mode (notification messages, zero client code) and a "let the app decide" mode (data messages, full control). The distinction exists because some pushes are simple alerts and others need custom rendering/routing/background work. Targeting methods exist to send to one device, a broadcast audience, or logical segments efficiently.

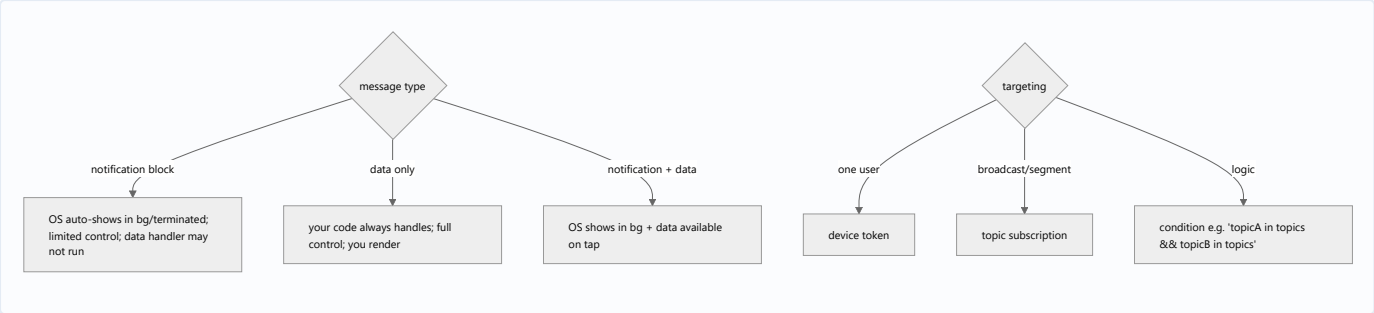
Real-world analogy

A **notification message** is a **pre-printed postcard** the postal service reads aloud and displays for you — easy, but you can't change how it looks or reliably act on it. A **data message** is a **sealed envelope with instructions** only you open and act on — more work, full control. **Topics** are a **magazine subscription** (broadcast to all subscribers); **tokens** are a **personal letter** to one address.

Problem Statement

Send (a) a personalized "order shipped, tap to track" push to one user that deep-links reliably, and (b) a "flash sale" broadcast to everyone who opted in. Decide message type + targeting, and send from the server. You'll model payloads and targeting.

Internal Working



- **Notification message**: has a `notification: {title, body}` block. In **background/terminated** the **OS displays it automatically**; your `onMessage` /background handler behavior is limited, and **data processing may not run** unless the user taps. Easy but low control.
- **Data message**: only a `data: {k:v}` map (no `notification` block). **Your handler always runs** (foreground `onMessage`; terminated/background isolate handler), and **you render** (via local notifications). Full control over display, routing, and background work. **Recommended** when you need reliable routing/custom UI.
- **Hybrid (notification + data)**: OS shows the notification in background; the `data` is available on tap (`onMessageOpenedApp` / `getInitialMessage`). Common compromise, but foreground still needs you to render, and pure-background data work is limited — for guaranteed background processing prefer **data-only** with high priority.
- **Priority/TTL**: set **high priority** for time-sensitive/data messages (esp. Android, to wake the app) and a TTL for expiry.
- **Targeting** (server-side via Admin SDK / HTTP v1 API):
  - **Token**: send to one device's registration token (personal) — requires your backend to store tokens (02_fcm_push.md).
  - **Topic**: clients `subscribeToTopic('deals')`; server sends to the topic (broadcast/segments) — no token management, but public-ish (anyone can subscribe) — don't use for sensitive targeting.
  - **Condition**: boolean combo of topics (`'deals' in topics && 'india' in topics`).
- **Server-only sending**: the FCM **server key/service account** is secret — **never** send from the client (it could spam anyone). The app only subscribes to topics + reports tokens.

Memory Representation

Payloads are small key/value maps (keep under FCM's ~4KB limit; put ids/routes in `data`, not big blobs). Tokens/topic subscriptions are managed per device.

Compiler Behavior

Not applicable.

Runtime Behavior

Notification messages get OS-throttled/auto-shown; data messages invoke your handler (subject to delivery/priority/doze). Topic fan-out is eventually delivered to subscribers (slight latency for large topics).

Flutter Engine Behavior

Data-message background processing runs in the background isolate (02_fcm_push.md); notification-only messages may bypass your Dart code entirely until tapped.

Dart VM Behavior

Not applicable.

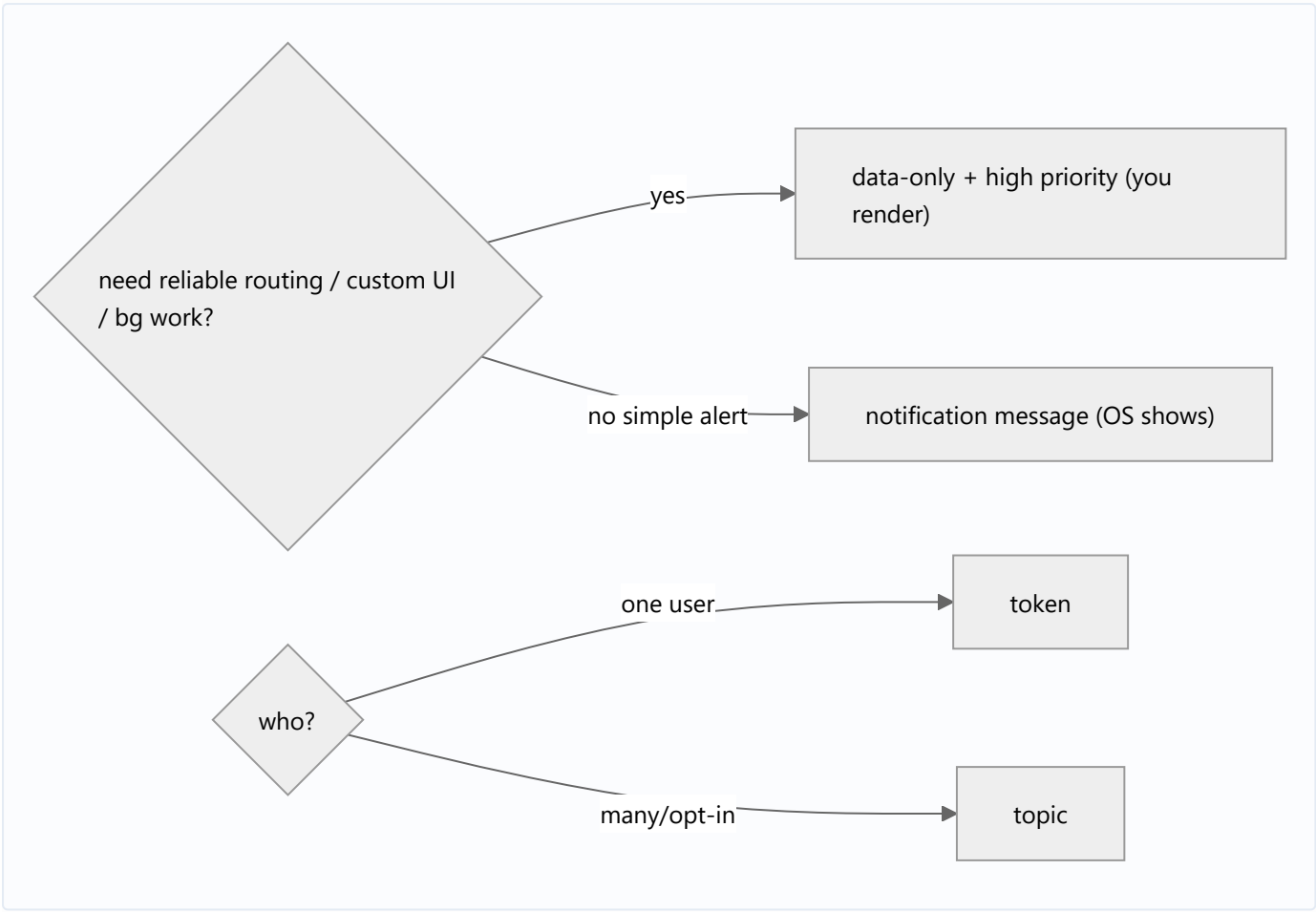
Examples

```
// CLIENT: only subscribe to topics + report token. NEVER send messages from the client.
await FirebaseMessaging.instance.subscribeToTopic('deals');
// await FirebaseMessaging.instance.unsubscribeFromTopic('deals');
```

```
// SERVER (FCM HTTP v1) — DATA-ONLY for reliable routing (you render + route)
{
  "message": {
    "token": "<device_token>",                 // personal target
    "data": { "type": "order", "orderId": "42", "route": "/order/42" },
    "android": { "priority": "high" },
    "apns": { "headers": { "apns-priority": "10" } }
  }
}
```

```
// SERVER — BROADCAST via topic, notification + data hybrid
{
  "message": {
    "topic": "deals",                            // everyone subscribed
    "notification": { "title": "Flash Sale", "body": "50% off today!" },
    "data": { "route": "/sale" }
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Sending from the client	Anyone could spam users	Send server-side (secret key/service account)
Notification-only when routing needed	Data handler may not run	Use data-only / hybrid + handle tap data
Topics for sensitive/private targeting	Anyone can subscribe	Use tokens (+ auth) for personal/sensitive
Not setting high priority for data	Delayed/not woken (Android)	`priority: high` for time-sensitive data
Oversized payloads	>4KB rejected	Send ids/routes; fetch details in-app
Expecting guaranteed delivery	Best-effort	Design for miss/late; reconcile in-app

Best Practices

- Prefer **data-only** (or notification+data hybrid) with **high priority** when you need reliable routing, custom rendering, or background processing; use plain **notification messages** only for simple OS-shown alerts.
- **Send only from the server** (Admin SDK/HTTP v1 with a service account) — never expose the FCM key in the client; the app only **subscribes to topics** and **reports tokens**.

- Target by **token** (personal/sensitive, requires auth) vs **topic/condition** (broadcast/opt-in segments); keep payloads **small** (ids/routes, fetch details in-app).
- Treat delivery as **best-effort**; reconcile important state in-app rather than relying on a single push.

Performance

Small payloads + topic fan-out are efficient. High priority wakes the app (slightly more battery) — use for time-sensitive only. Don't over-broadcast (annoyance + unsubscribes).

Advantages / Disadvantages

- **+** Flexible: OS-simple (notification) or fully controlled (data); efficient targeting (token/topic/condition); server-controlled.
- **−** Type/behavior subtleties (handler-runs?), topics are public-ish, best-effort delivery, 4KB limit, server infra required.

Interview Questions

1. 🟢 **Notification vs data message — what's the difference?** — Notification messages have a `notification` block the OS auto-displays (limited control, handler may not run); data messages carry only `data`, always invoke your handler, and you render.
2. 🟢 **How do you target one user vs a broadcast?** — Token (one device, personal) vs topic/condition (opt-in broadcast/segments).
3. 🟡 **Why send push only from the server?** — The FCM key/service account is secret; a client with it could spam any user — the app only subscribes/reports tokens.
4. 🟡 **When would you choose data-only?** — When you need reliable routing, custom display, or background processing (your handler always runs); set high priority.
5. 🟡 **Why not use topics for sensitive targeting?** — Anyone can subscribe to a topic; use authenticated token targeting for personal/sensitive pushes.
6. 🔴 **Why is notification-only unreliable for routing/background work?** — In background/terminated the OS shows it but your data handler may not run until the user taps; data-only guarantees handler execution.
7. 🔴 **What are payload/delivery constraints?** — ~4KB payload limit and best-effort (not guaranteed/instant) delivery; send ids and reconcile in-app.

Senior Engineer Tips

- Default to data-only + high priority + client-side rendering; it's the only way to get consistent behavior across all three states and platforms.

- Keep payloads to ids and a route; fetch the real content in-app so you're never bound by the 4KB limit or stale push data.
- Use topics for opt-in broadcasts and tokens (behind auth) for anything personal; never let the FCM key touch the client.

Architect Perspective

Message type + targeting is the contract between your backend and the client push handler. Standardizing on data-only (or hybrid) with a `route` / `type` / `id` schema, server-only sending, and clear token-vs-topic rules makes push predictable and routable, and keeps the display/routing logic in your app (behind the notification service) rather than at the mercy of OS auto-display (02_fcm_push.md, 04_handling_and_deeplinks.md).

Summary

- Notification messages = OS-shown, low control, handler may not run; data messages = your code always handles + renders (recommended for routing/bg); hybrid = both.
- Target by token (personal, auth) or topic/condition (broadcast/opt-in); send **server-side only**.
- Keep payloads small (ids/routes), set high priority for time-sensitive data, treat delivery as best-effort.

Revision Notes

- `notification` block → OS auto-shows (bg/terminated), limited handler; `data` only → your handler always runs, you render; hybrid = both.
- Data-only + `priority: high` for reliable routing/bg; payload ≤ ~4KB (send ids/route).
- Target: token (one user, requires stored token + auth) / topic (`subscribeToTopic` , broadcast, public-ish) / condition. Send from server (service account) only.

Practice Questions

1. When does your Dart handler run for notification vs data messages?
2. Token vs topic targeting — when each?
3. Why must sending be server-side?

Coding Questions

1. Client: subscribe/unsubscribe to a topic and report the token.
2. Server payloads: a data-only personal push and a topic broadcast (JSON).
3. Design a small `data` schema (`type` / `id` / `route`) for routable push.

Mini Project

Targeted push (Flutter + backend): Implement client topic subscription + token reporting, and a server (Admin SDK/HTTP v1) that sends (a) a data-only personal "order shipped" push (token, high priority, `route`) and (b) a topic broadcast "flash sale" (notification+data). The client renders/routes from the `data`. Acceptance: data-only push reliably handled/routed in all states; topic broadcast reaches subscribers; sending is server-only (no key in app); payloads small (ids/route); token/topic targeting used appropriately.

Handling Taps & Deep Links (Cold Start Included)

> A tapped notification must open the right screen — reliably from **foreground, background, and cold start**. The pattern: put a **route/deep link in the payload** (`data.route`), collect the tap from the three entry points (`onMessageOpenedApp` , `getInitialMessage` , and the local-notification tap handler / `getNotificationAppLaunchDetails`), and funnel them all into **one router call** (`go_router`) — deferring the terminated-launch case until the router is ready.

Introduction

Reception is only half the job; the payoff is routing the user to the relevant content on tap. This file covers extracting the route from the payload, the multiple tap-entry points across states (and the tricky cold-start timing), and consolidating them into a single deep-link handler with `go_router` .

Why this concept exists

Notifications are re-engagement: "order shipped" should open the order, not the home screen. But the tap arrives through different callbacks depending on app state, and on cold start the router may not exist yet when the launch message is available — so a single, timing-safe routing funnel is needed to avoid lost or mis-timed navigation.

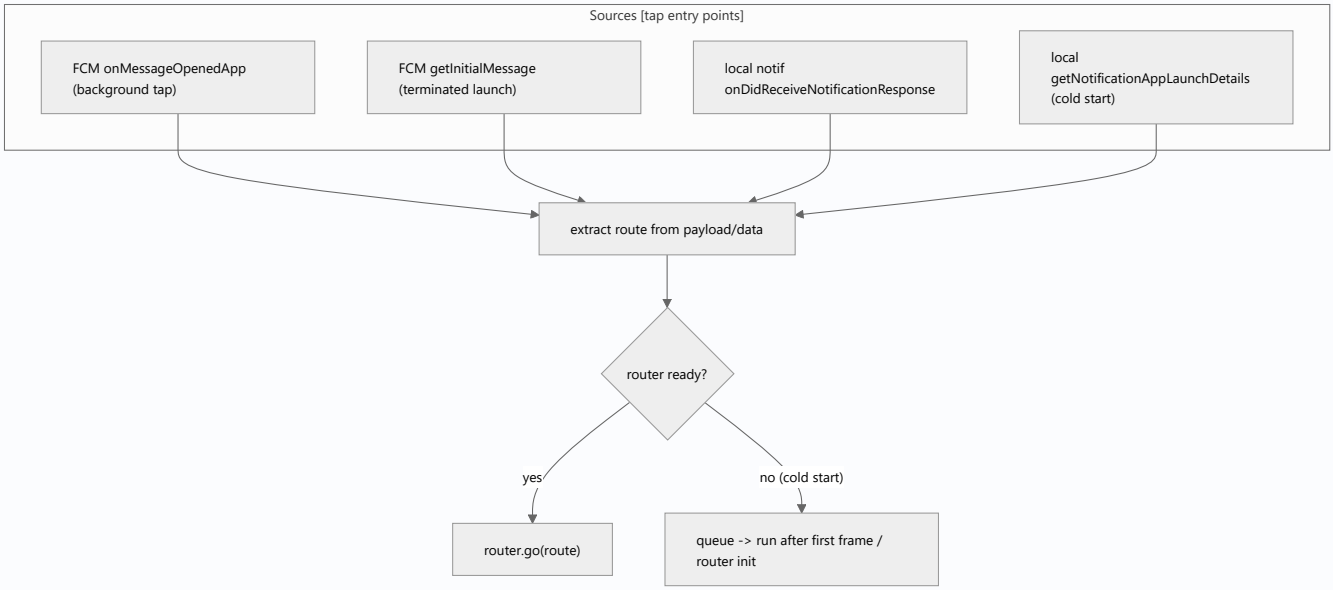
Real-world analogy

The notification payload is a **luggage tag with a destination**; no matter which **door the traveler enters** (already inside = foreground, side door = background, front door after being away = cold start), the **concierge (router)** reads the same tag and walks them to the right room. The only catch: at the front door on cold start, you must **wait for the concierge to arrive** before handing over the tag.

Problem Statement

Ensure tapping "order #42 shipped" opens `/order/42` whether the app was open, backgrounded, or killed — for both FCM and local notifications — using one handler. You'll extract routes from payloads and wire all tap sources into `go_router` .

Internal Working



- **Route in payload**: standardize a field — FCM `data['route']` (or `type` + `id` you map to a route), local-notification `payload` string (01_local_notifications.md). Keep it a real app route (`/order/42`) so it maps 1:1 to `go_router`.
- **Tap entry points** (must handle all):
  - **FCM background tap** → `FirebaseMessaging.onMessageOpenedApp` (app resumes).
  - **FCM terminated launch** → `FirebaseMessaging.getInitialMessage()` at startup (returns the message that launched the app, or null).
  - **Local notification tap** → `onDidReceiveNotificationResponse` (foreground/background) and `getNotificationAppLaunchDetails()` (cold start).
  - **Foreground FCM** (`onMessage`) usually has **no tap** yet — you render a local notification whose tap then routes.
- **Cold-start timing (the gotcha)**: on terminated launch, the launch message/details are available *before* the widget tree/router is built. **Capture the pending route** at startup and **navigate after the router is initialized** (e.g., in `go_router`'s first build / a post-frame callback / a `redirect`), or you'll navigate into a non-existent router.
- **One funnel**: convert every source to `handleRoute(String route)` that either navigates now (router ready) or stores a `pendingRoute` consumed once the router exists — avoids duplicated, racy navigation logic.
- **go_router integration**: `router.go(route)` / `router.push(route)`; or feed the pending route via the router's `initialLocation` / `redirect`.
- **Auth/guarding**: a deep link may require login — let `go_router` `redirect` gate it (send to login, then resume) (Module 13).

Memory Representation

A single nullable `pendingRoute` holds a cold-start deep link until consumed. Otherwise routing is stateless (payload → router call).

Compiler Behavior

Not applicable.

Runtime Behavior

Foreground/background taps route immediately (router exists). Cold-start taps must defer until the router mounts — the classic "notification opens app but lands on home" bug is a missed deferral.

Flutter Engine Behavior

On terminated launch the engine starts, runs `main`, and only then builds the tree; launch messages retrieved in `main` /early must wait for the first frame/router to navigate.

Dart VM Behavior

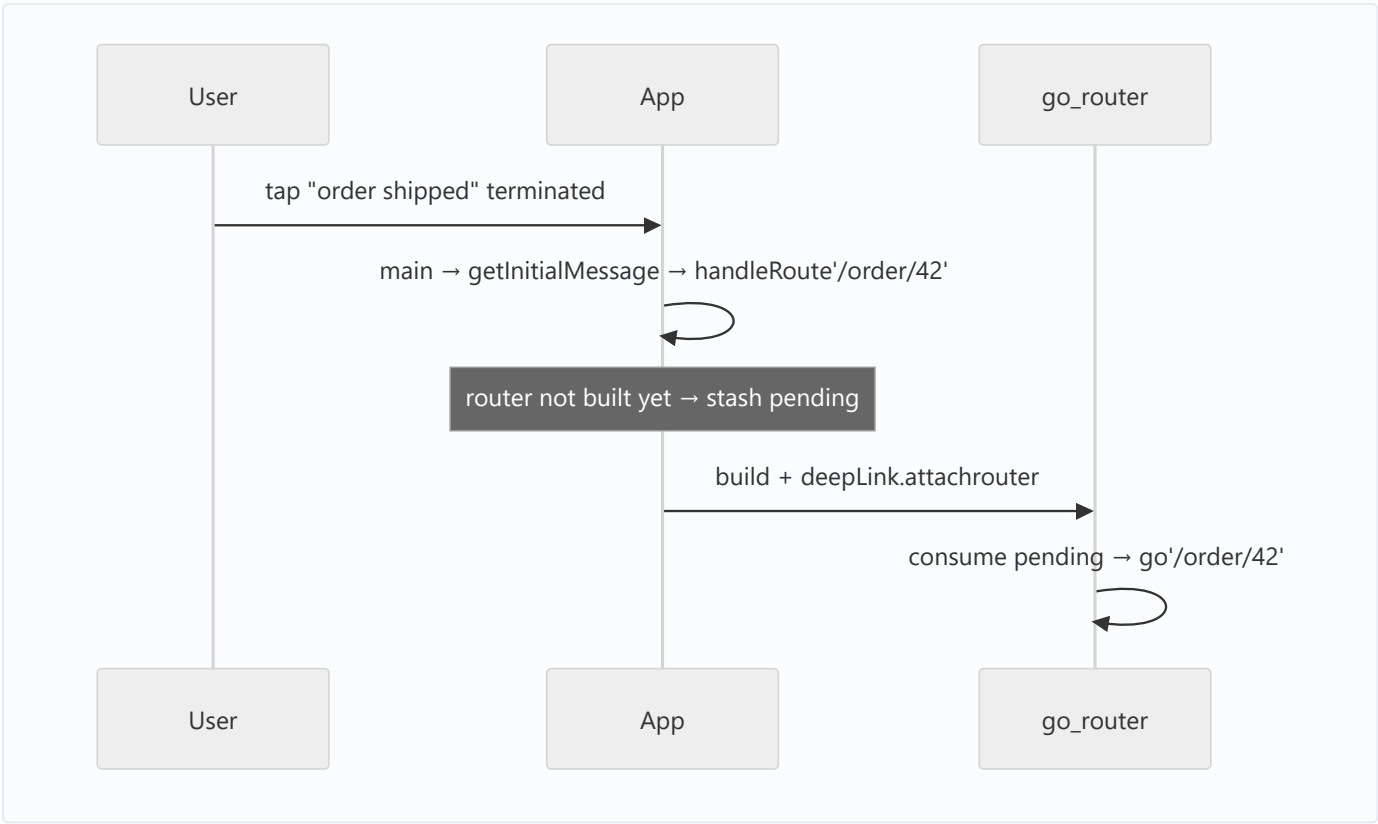
Not applicable.

Examples

```
import 'package:firebase_messaging/firebase_messaging.dart';
import 'package:flutter_local_notifications/flutter_local_notifications.dart';

// Single funnel: navigate now if ready, else stash until the router mounts.
class DeepLinkRouter {
  GoRouter? _router;
  String? _pending;

  void attach(GoRouter router) {           // called once the router exists
    _router = router;
    final p = _pending;
    if (p != null) { _pending = null; router.go(p); } // consume cold-start route
  }

  void handleRoute(String? route) {
```

```
    if (route == null) return;
    final r = _router;
    if (r != null) { r.go(route); } else { _pending = route; } // defer on cold start
  }
}

final deepLink = DeepLinkRouter();

Future<void> wireNotificationTaps() async {
  final fm = FirebaseMessaging.instance;
  // FCM: background tap + terminated launch
  fm.onMessageOpenedApp.listen((m) => deepLink.handleRoute(m.data['route']));
  final initial = await fm.getInitialMessage();
  deepLink.handleRoute(initial?.data['route']);            // may set _pending

  // Local notifications: cold-start launch details
  final details = await FlutterLocalNotificationsPlugin().getNotificationAppLaunchDetails();
  if (details?.didNotificationLaunchApp ?? false) {
    deepLink.handleRoute(details!.notificationResponse?.payload);
  }
  // (local tap handler onDidReceiveNotificationResponse also calls deepLink.handleRoute)
  // (foreground onMessage renders a local notification -> its tap routes)
}
// In app build: deepLink.attach(myGoRouter);  // consumes any pending route
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Ignoring `getInitialMessage`/launch details	Cold-start taps lost (land on home)	Handle terminated launch on startup
Navigating before router exists	Crash/no-op on cold start	Defer via pending route → attach
Separate routing logic per source	Duplication/races	One `handleRoute` funnel
No route in payload	Can't deep-link	Standardize `data['route']`/payload
Foreground push not tappable	User can't act	Render a local notification that routes
Ignoring auth on deep link	Opens gated screen unauthenticated	`go_router` redirect/guard

Best Practices

- Put a **real app route** in every payload (`data['route']` / local `payload`) and funnel **all tap sources** into one `handleRoute`.
- **Handle cold start**: capture the launch message/details at startup and **navigate after the router mounts** (pending-route pattern).

- Render **foreground** FCM as a local notification so it's tappable → routes; **guard** deep links via `go_router` redirect (auth/existence).
- Keep routing **stateless except the single pending route**; wrap in a service so tests can assert route extraction/deferral.

Performance

Trivial. The only concern is timing (defer cold-start navigation) and idempotency (don't double-navigate if a source fires twice).

Advantages / Disadvantages

- + Reliable re-engagement (tap → exact content) across all states, unified handling, auth-aware routing.
- − Cold-start timing complexity, multiple entry points to wire, need a payload route convention, dedupe/guard care.

Interview Questions

1. 🟢 **How do you route the user when they tap a notification?** — Extract a route from the payload (`data['route']` /local `payload`) and call the router (`go_router`) via a single handler.
2. 🟢 **What are the tap entry points across app states?** — FCM `onMessageOpenedApp` (background), `getInitialMessage` (terminated launch); local `onDidReceiveNotificationResponse` and `getNotificationAppLaunchDetails` (cold start).
3. 🟡 **What's the cold-start routing gotcha?** — The launch message is available before the router is built; you must defer navigation until the router mounts (pending-route pattern).
4. 🟡 **Why render foreground FCM as a local notification?** — Foreground `onMessage` has no OS notification/tap; rendering a local one makes it tappable → routes.
5. 🟡 **Why funnel all sources into one handler?** — To avoid duplicated, racy per-source navigation and centralize dedupe/guarding.
6. 🔴 **How do you handle a deep link to a gated screen?** — Let `go_router` `redirect` check auth and send to login, then resume to the target after authentication.
7. 🔴 **How do you prevent double navigation?** — Consume the pending route once and dedupe sources (e.g., don't process both `getInitialMessage` and a local launch for the same tap).

Senior Engineer Tips

- Build the single `handleRoute` funnel + pending-route deferral first; every notification feature then just supplies a route.

- Test the terminated-tap path explicitly (force-kill → tap) — it's the one that silently lands users on home in most apps.
- Encode routes (not opaque ids) in payloads and let `go_router` redirect handle auth/existence, so deep links behave like any in-app navigation.

Architect Perspective

Tap handling is the bridge from delivery to product value, and its complexity is entirely about state/timing. A `DeepLinkRouter` funnel with cold-start deferral, fed by all reception sources and integrated with `go_router`'s guards, makes notification routing reliable and testable — and reuses the same deep-link infrastructure as universal/app links (Module 13, 28 · ios_integration, 05_notifications_integration.md).

Summary

- Standardize a route in the payload; funnel `onMessageOpenedApp`, `getInitialMessage`, local tap, and launch details into one `handleRoute`.
- Defer cold-start navigation until the router mounts (pending-route pattern); render foreground push as a tappable local notification.
- Guard deep links via `go_router` redirect; dedupe to avoid double navigation.

Revision Notes

- Route in payload: FCM `data['route']` / local `payload`; one `handleRoute` funnel → `router.go(route)`.
- Sources: `onMessageOpenedApp` (bg), `getInitialMessage` (terminated), local `onDidReceiveNotificationResponse` + `getNotificationAppLaunchDetails` (cold start).
- Cold start: stash pending route, navigate after router mounts (`attach`); foreground → render local notif (tappable); guard via `go_router` redirect.

Practice Questions

1. What are all the tap entry points and which state does each cover?
2. Why and how do you defer cold-start navigation?
3. How do you make a foreground push tappable?

Coding Questions

1. Implement a `DeepLinkRouter` with pending-route deferral + `attach`.
2. Wire all FCM + local tap sources into `handleRoute`.

3. Guard a deep link to a login-required screen via `go_router` redirect.

Mini Project

Deep-link routing (Flutter): Build a `DeepLinkRouter` funnel that routes notification taps to `go_router` from all sources (FCM background/terminated, local tap/cold-start), using a `data['route']`/payload convention and the pending-route deferral for cold start. Render foreground FCM as a tappable local notification; guard a gated route via redirect. Acceptance: tapping opens the exact screen in foreground, background, and cold start; foreground push is tappable; gated deep link routes through login; no double navigation; behind a service.

Notifications Integration (Capstone: One Service Ties It Together)

> The maintainable shape: a single `NotificationService` that owns permissions, FCM setup + token sync, the three-state handlers, foreground rendering (delegating to the local-notifications plugin), and the deep-link funnel — exposing a small API (`init` , `subscribe` , entitlement/route stream) to the app while the **backend owns targeting/sending**. This unifies local + push + routing into one testable seam instead of scattered `firebase_messaging` / `flutter_local_notifications` calls.

Introduction

This module capstone composes local notifications, FCM, message types, and deep-link handling into one architecture. Each piece is tricky alone; together they must cooperate (foreground FCM → local notification → tap → route). A `NotificationService` centralizes this so the rest of the app stays ignorant of the plumbing. This file shows the service design and the end-to-end flow.

Why this concept exists

Notification code is stateful (permissions, tokens), multi-isolate (background handler), state-dependent (foreground/background/terminated), and cross-cutting (routing, auth). Scattering it across widgets/services makes it untestable and fragile. One service with a clean API — like repositories for data — isolates the mess behind a boundary consistent with clean architecture (Module 40).

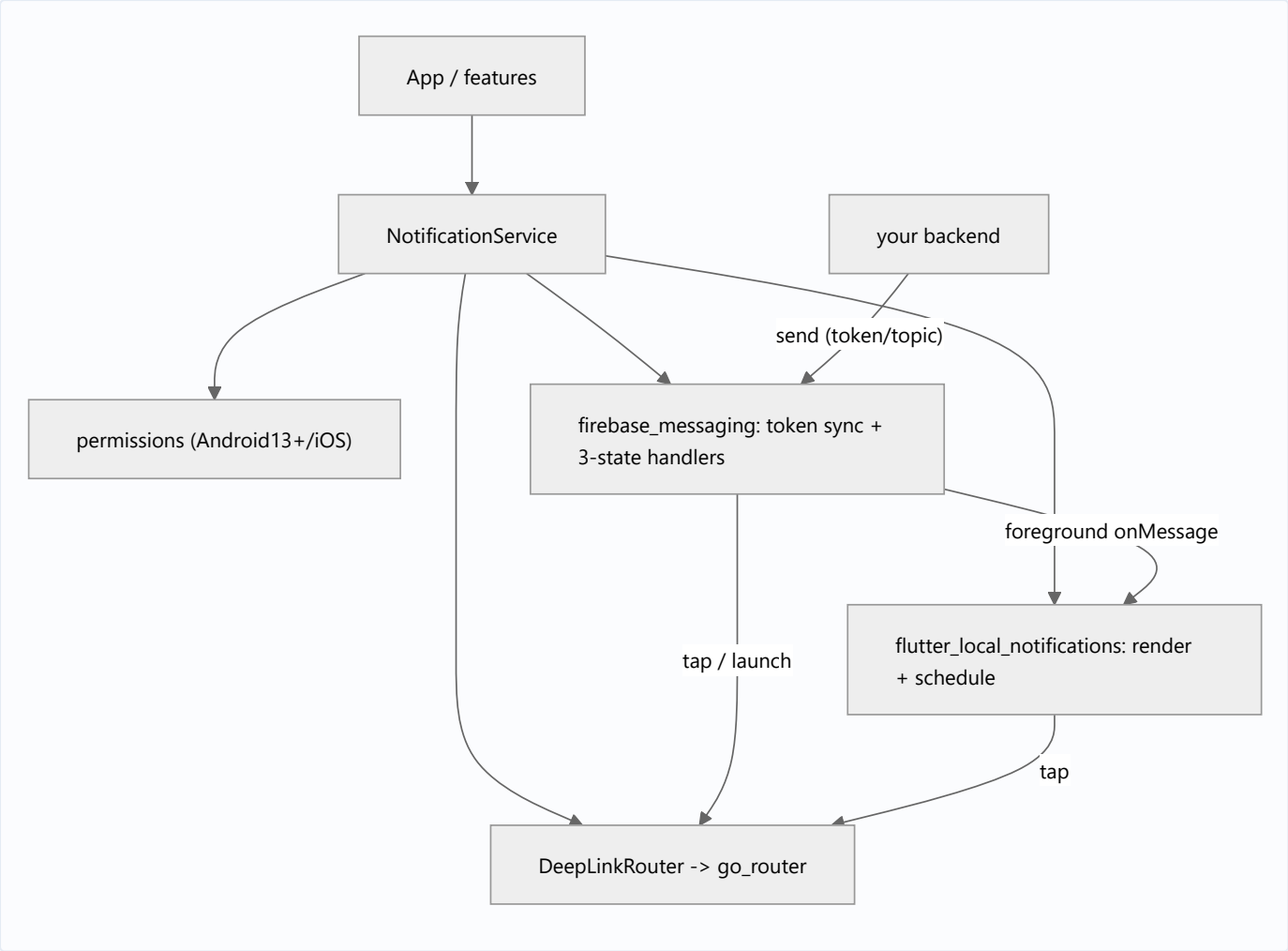
Real-world analogy

The `NotificationService` is the **mailroom** of the app: it handles the address book (tokens/permissions), sorts incoming mail by how it arrived (foreground/background/terminated), decides what to display (local rendering), and routes each item to the right desk (deep links). The rest of the company just says "subscribe me to deals" or "here's where a tap should go" — they don't run the mailroom.

Problem Statement

Deliver: FCM push shown in all three states, foreground push rendered via a local notification, taps deep-linking through `go_router` (incl. cold start), topic subscription for broadcasts, scheduled local reminders, permission handling, and token sync — all behind one `NotificationService` with the backend sending. You'll compose every file in this module.

Internal Working



- `NotificationService` **responsibilities**:
  - **Permissions**: request + expose status; degrade gracefully (27/28).
  - **FCM**: `getToken` / `onTokenRefresh` → backend; register the top-level background handler before `runApp`; wire `onMessage` / `onMessageOpenedApp` / `getInitialMessage` (02_fcm_push.md).
  - **Foreground rendering**: on `onMessage`, build a **local notification** (channel + payload = route) so foreground push is visible + tappable (01_local_notifications.md).
  - **Deep links**: feed every tap into the `DeepLinkRouter` funnel with cold-start deferral → `go_router` (04_handling_and_deeplinks.md).
  - **Topics/scheduling**: `subscribe/unsubscribe`, `scheduleReminder(...)`.
- **Backend owns sending/targeting** (data-only/hybrid, token/topic, server key) (03_message_types_and_targeting.md); the client never sends.
- **Bootstrap order**: init Firebase → register bg handler → init local plugin (+channels) → request permission → sync token → wire handlers → (later) `deepLink.attach(router)`.
- **Testability**: depend on abstractions (a messaging interface, a local-notifications interface, a router) so the service is unit-testable with fakes (assert: foreground message → local notification shown; tap → route handled).

Memory Representation

The service holds init state, the token, permission status, and the single pending route. Background messages run in their own isolate (minimal handler). Notifications live in the OS.

Compiler Behavior

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

Runtime Behavior

The service orchestrates async setup + event handlers; foreground push renders locally; taps route (deferring cold start). Backend fan-out is best-effort; the app reconciles important state rather than trusting a single push.

Flutter Engine Behavior

One UI isolate (service + handlers + rendering) and one background isolate (terminated/background data handler); notifications and taps bridge native ↔ Flutter.

Dart VM Behavior

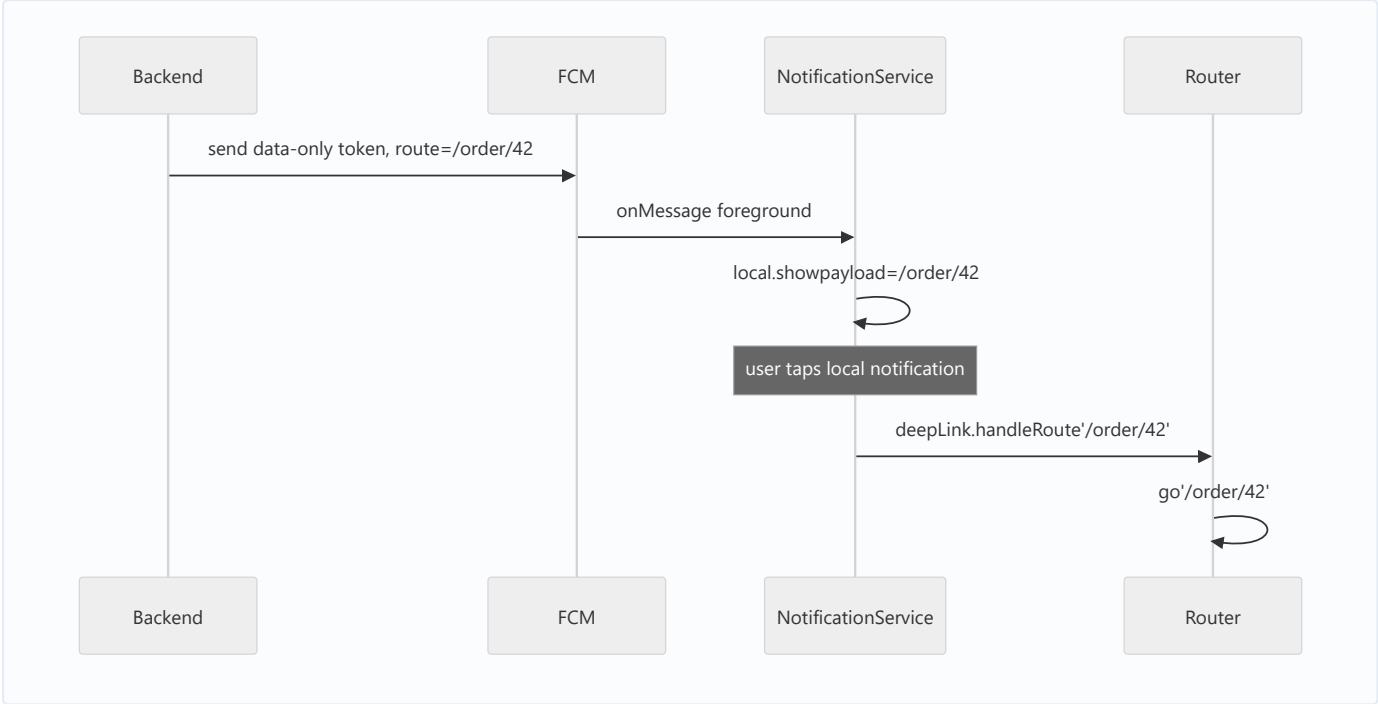
Two isolates as in 02_fcm_push.md; the service's state lives only in the UI isolate.

Examples

```
class NotificationService {
  final MessagingClient messaging;       // wraps firebase_messaging (mockable)
  final LocalNotifier local;             // wraps flutter_local_notifications
  final DeepLinkRouter deepLink;
  final Backend backend;
  NotificationService(this.messaging, this.local, this.deepLink, this.backend);

  Future<void> init() async {
    await local.init(onTap: deepLink.handleRoute);           // local taps -> route
    await messaging.requestPermission();                     // + handle denial
    await backend.syncToken(await messaging.getToken());
    messaging.onTokenRefresh(backend.syncToken);

    // Foreground: render a local notification (visible + tappable)
    messaging.onMessage((m) => local.show(
      title: m.title, body: m.body, payload: m.data['route']));

    // Taps: background + terminated launch -> funnel
    messaging.onMessageOpenedApp((m) => deepLink.handleRoute(m.data['route']));
    deepLink.handleRoute((await messaging.getInitialMessage())?.data['route']);
  }

  Future<void> subscribe(String topic) => messaging.subscribeToTopic(topic);
  Future<void> scheduleReminder(DateTime at, String route) =>
      local.schedule(at: at, payload: route);
}
// Bootstrap: Firebase.initializeApp -> messaging.registerBackgroundHandler(_bg) -> service.init()
// App build: deepLink.attach(router);
```

```
// Unit test with fakes — no device/Firebase
test('foreground push renders a local notification', () async {
  final local = FakeLocalNotifier();
  final svc = NotificationService(FakeMessaging(), local, DeepLinkRouter(), FakeBackend());
  await svc.init();
  FakeMessaging.emitForeground({'route': '/order/42'}, title: 'Shipped');
  expect(local.shown.single.payload, '/order/42');
});
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Scattered messaging/local calls	Untestable, inconsistent	One `NotificationService`
No foreground rendering	Push invisible when app open	Render local notification on `onMessage`
Wrong bootstrap order	Bg handler/permission/token issues	Firebase → bg handler → local → perm → token → handlers
Router attached too late/never	Cold-start taps lost	`deepLink.attach(router)` on build
Client-side sending	Security	Backend sends; client subscribes/reports
No graceful denial handling	Crash/blank	Handle permission denial
Untestable (direct plugins)	Needs device	Depend on interfaces + fakes

Best Practices

- Put **all** notification concerns in **one** `NotificationService` with a small API; depend on **interfaces** (messaging/local/router) for testability.
- Follow the **bootstrap order** (Firebase → bg handler before `runApp` → local init/channels → permission → token sync → handlers → `attach` router).
- **Render foreground push locally**, funnel every tap into the **deep-link router** (cold-start deferral), and let the **backend own targeting/sending** (data-only/hybrid).

- Handle **permission denial** gracefully; **reconcile** important state in-app (don't rely on a single best-effort push); write **unit tests with fakes**.

Performance

Push is event-driven (battery-friendly); the service adds no overhead. Keep the background handler minimal; don't over-notify (throttle/collapse). Reconcile via in-app fetches rather than depending on delivery.

Advantages / Disadvantages

- **+** Testable, consistent, single seam for local+push+routing; permission/token/state handling in one place; backend-controlled sending.
- **−** Upfront structure/boilerplate, strict bootstrap ordering, multi-isolate constraints, best-effort delivery to design around.

Interview Questions

1. 🟢 **Why centralize notifications in one service?** — Notification code is stateful, multi-isolate, state-dependent, and cross-cutting; one service makes it consistent and testable instead of scattered plugin calls.
2. 🟢 **How is a foreground push made visible and tappable?** — On `onMessage`, render a local notification whose payload carries the route; its tap feeds the deep-link router.
3. 🟡 **What's the correct bootstrap order?** — Firebase init → register top-level background handler before `runApp` → init local plugin/channels → request permission → sync token → wire handlers → attach router.
4. 🟡 **How do you make the service testable?** — Depend on interfaces (messaging/local/router) and inject fakes; assert foreground→local-render and tap→route.
5. 🟡 **Who sends notifications and why?** — The backend (secret key, targeting); the client only subscribes to topics and reports tokens.
6. 🔴 **How do all pieces cooperate for an order-shipped push?** — Backend sends data-only (route) → foreground renders local / background tray → tap funnels to `go_router` → order screen (deferred if cold start).
7. 🔴 **Why reconcile state in-app rather than trust the push?** — Delivery is best-effort; the app should fetch/verify important state so a missed/late push doesn't leave stale data.

Senior Engineer Tips

- Nail the bootstrap order and the router `attach` first — most "notifications are flaky" reports are ordering/cold-start bugs, not delivery.

- Keep the service behind interfaces so CI can test the foreground-render and tap-routing logic without a device or Firebase.
- Treat push as a hint to refresh, not a data source: reconcile in-app so best-effort delivery never corrupts state.

Architect Perspective

Notifications integration applies the same boundary discipline as the rest of the app to an unusually stateful, multi-isolate domain: one service owns permissions, tokens, the three-state handlers, foreground rendering, and deep-link routing; the backend owns targeting/sending. This yields reliable re-engagement, a testable seam, and clean cooperation between local and push — consistent with clean architecture and the app's routing/auth infrastructure (Module 40, Module 13, Module 17).

Summary

- One `NotificationService` owns permissions, FCM token sync, three-state handlers, foreground local rendering, scheduling, topics, and the deep-link funnel.
- Backend owns targeting/sending (data-only/hybrid); client subscribes/reports tokens.
- Follow the bootstrap order, defer cold-start routing, handle denial, reconcile in-app, and test with fakes.

Revision Notes

- Service centralizes: permissions, `getToken` / `onTokenRefresh` →backend, bg handler (top-level), `onMessage` →local render, taps→ `DeepLinkRouter` , topics, scheduling.
- Bootstrap: Firebase → bg handler (before `runApp`) → local init/channels → permission → token → handlers → `deepLink.attach(router)` .
- Backend sends (data-only/hybrid, token/topic); reconcile important state in-app; interfaces + fakes for tests.

Practice Questions

1. What does the `NotificationService` own, and what does the backend own?
2. What is the correct bootstrap order and why?
3. How do local + FCM + routing cooperate for a foreground push tap?

Coding Questions

1. Design a `NotificationService` over messaging/local/router interfaces.

2. Implement the foreground `onMessage` → local render → tap → route path.
3. Write a unit test (fakes) asserting tap → correct route.

Mini Project

Unified notifications (capstone — Flutter + Firebase + backend): Build a `NotificationService` composing FCM (token sync, three-state handlers), local notifications (foreground rendering + scheduled reminders + channels), topic subscription, and the `DeepLinkRouter` funnel into `go_router` — with the backend sending data-only/hybrid pushes by token/topic. Provide interfaces + fakes and unit tests. Acceptance: push handled in all three states; foreground rendered locally + tappable; taps deep-link (incl. cold start) to the right screen; topic broadcast + scheduled reminders work; permissions handled; backend-only sending; unit-tested with fakes; runs end-to-end on a device.