26 · Platform Channels

Flutter Practice Notes

Contents

26 · Platform Channels

Introduction

Platform channels are Flutter's bridge to **native code** (Kotlin/Java, Swift/Obj-C, C/C++, platform APIs) when Dart/plugins can't do the job. This module covers the messaging model, `MethodChannel` (call native methods), `EventChannel` (native→Dart streams), writing plugins with **Pigeon** (type-safe), and **FFI** (direct C interop).

Why this module exists

Some capabilities live only in native SDKs/OS APIs (sensors, native UI, platform features, C libraries). Channels let Dart and native exchange messages asynchronously; FFI calls C directly. Doing this correctly (async, threading, type-safety) is what makes native integration reliable.

Module map

#	File	Topic	Level
1	01_platform_channel_fundamentals.md	Messaging model, codec, async, threading	🔴
2	02_method_channel.md	`MethodChannel` (invoke/handle/errors)	🔵
3	03_event_channel.md	`EventChannel` (native→Dart streams)	🔵
4	04_plugins_and_pigeon.md	Writing plugins, federated, Pigeon	🔴
5	05_ffi.md	`dart:ffi` direct C interop	🔴

> **Cross-references:** Engine/embedder boundary + task runners: 10 · threading_model, 10 · engine_internals. Native Android/iOS: Modules 27/28. Device features (via plugins): Module 29. Isolates/async: 02. Background isolate channels: 02 · isolates.

Prerequisites

10 Flutter Architecture (threading/embedder), 02 Advanced Dart (async/streams/isolates).

What you'll be able to do after this module

- Explain the channel messaging model (codec, async, threading).
- Call native methods (`MethodChannel`) and stream native events (`EventChannel`).
- Write plugins and use Pigeon for type-safe channels.
- Call C libraries directly via FFI.

- Decide channel vs FFI vs existing plugin.

Capstone

Native battery + sensor bridge: A `MethodChannel` to read battery level, an `EventChannel` streaming battery/charging changes, wrapped behind a repository — plus a Pigeon-generated type-safe interface, and a small FFI call to a C function.

Summary

Platform channels pass async messages between Dart and native (via a codec, across threads); `MethodChannel` / `EventChannel` are the primitives, Pigeon adds type-safety, and FFI bypasses channels for direct C calls. Wrap all native access behind repositories to keep the app clean and testable.

Platform Channel Fundamentals (Messaging Model)

> A platform channel is a **named, asynchronous message pipe** between Dart and native code: messages are serialized by a **codec** (standard/JSON/binary), passed across the engine's platform/UI task runners, and handled on the native side — always async, and (by default) on the platform's main thread.

Introduction

Before the specific channels, understand the model: named channels, message codecs, asynchrony, threading, and where handlers run. This explains why calls are `Future`-based, what data types cross, and the threading pitfalls — grounding `MethodChannel` / `EventChannel` /plugins/FFI.

Why this concept exists

Dart (UI isolate) and native code run separately; they can't call each other directly. Channels provide a standardized, serialized, async message-passing mechanism across that boundary — the embedder-level bridge (10 · engine_internals).

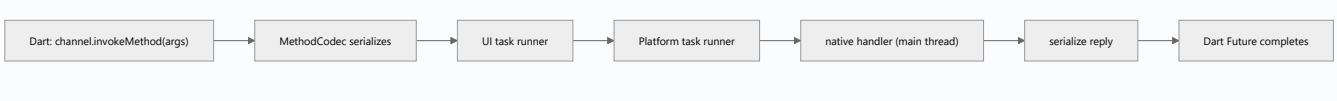
Real-world analogy

A platform channel is a **labeled pneumatic tube** between two offices (Dart and native): you put a message in a canister (serialized by the codec), send it through the tube (async), the other office processes it and (maybe) sends a reply back. You never reach into the other office directly.

Problem Statement

You need to call a native API from Dart and understand: is it sync or async? what data types can you pass? which thread does the native handler run on, and why does that matter? You'll map the channel model before writing code.

Internal Working



- **Named channels**: identified by a unique string (e.g., `'app/battery'`) shared by Dart and native — both sides register the same name.
- **Codecs**: messages are serialized/deserialized by a codec:

    - `StandardMessageCodec` (default): supports null, bool, num, String, `Uint8List`/typed lists, `List`, `Map` — efficient binary; **no custom classes** (map to primitives/maps).
    - Also `JSONMessageCodec`, `StringCodec`, `BinaryCodec` for special cases; Pigeon generates typed codecs (04_plugins_and_pigeon.md).

- **Asynchronous**: all channel calls return a `Future`/`Stream` — messages hop between the **UI task runner** and **platform task runner** (10 · threading_model); never assume synchronous.
- **Threading**: native handlers typically run on the **platform main thread** (UI thread on Android, main queue on iOS). Do heavy native work off that thread and return results; long native work blocks native UI. Dart callers await on the UI isolate.
- **Background isolates**: calling channels from a **background Dart isolate** needs `RootIsolateToken` + `BackgroundIsolateBinaryMessenger.ensureInitialized` (channels aren't wired there by default) (02 · isolates).
- **Three channel types**: `MethodChannel` (call methods, get a reply), `EventChannel` (native→Dart event streams), `BasicMessageChannel` (arbitrary async messages both ways).

Memory Representation

Messages are serialized to bytes for transit (copied across the boundary) — large payloads cost serialization time/memory; prefer small messages or `Uint8List`/FFI for big binary data.

Compiler Behavior

Untyped by default (`invokeMethod` returns `dynamic`) — you cast on the Dart side (Pigeon adds compile-time types).

Runtime Behavior

`invokeMethod` sends a message and awaits a reply (or `PlatformException`/`MissingPluginException`). Codec-unsupported types throw. Handlers run async on the platform side.

Flutter Engine Behavior

The engine's platform-channel infrastructure marshals messages across the platform/UI runners via the embedder; the embedder registers native handlers (10 · embedder_and_startup).

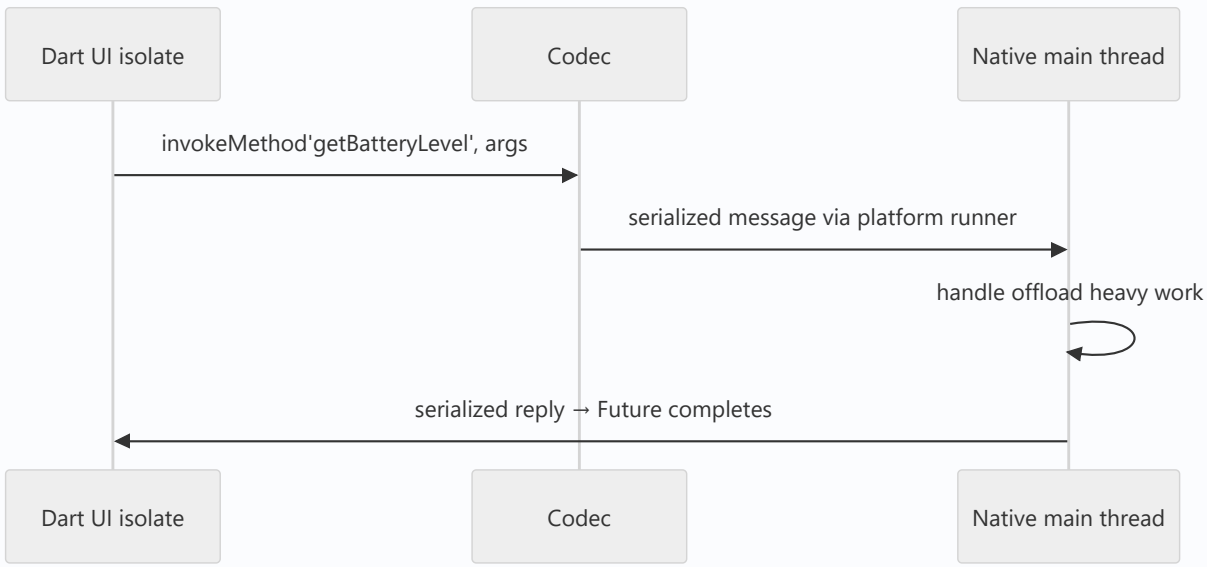
Dart VM Behavior

Dart-side calls run on the root isolate's event loop; background isolates need explicit messenger setup (02 · event_loop).

Examples

```
import 'package:flutter/services.dart';

// A named channel shared by Dart and native (same string on both sides)
const channel = MethodChannel('app/battery');

Future<int> batteryLevel() async {
  // Async: returns a Future; args/results limited to codec-supported types
  final level = await channel.invokeMethod<int>('getBatteryLevel', {'unit': 'percent'});
  return level ?? -1;
}
// Data types crossing: null/bool/num/String/Uint8List/List/Map — NOT custom classes.
// Native handler runs on the platform main thread (Android UI thread / iOS main queue).
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Passing custom class objects	Codec supports only primitives/maps/lists	Serialize to maps/primitives (or Pigeon)
Assuming synchronous calls	Channels are async	Always `await`; handle errors
Heavy native work on the main thread	Blocks native UI/janks	Offload natively; return the result
Channels from a background isolate without setup	Not wired there	`RootIsolateToken` + background messenger
Mismatched channel name/method	`MissingPluginException`/no handler	Match names/methods exactly both sides
Large payloads over channels	Serialization cost	Small messages / `Uint8List` / FFI

Best Practices

- Treat channels as **async message pipes**; always `await` and handle `PlatformException` / `MissingPluginException`.
- Pass **codec-supported types** (primitives/ `Map` / `List` / `Uint8List`); serialize custom data (or use **Pigeon** for typed messages).
- Do **heavy native work off the main thread** natively; keep messages small.
- Match **channel names/methods** exactly on both sides; namespace them (`com.app/feature`).
- For background-isolate channel use, set up the **background messenger**.
- **Wrap native access behind a repository** so the app depends on domain APIs, not channels.

Performance

Serialization + cross-thread hops add latency; keep messages small and calls off hot paths. Big binary → `Uint8List` (efficient) or FFI (zero-copy-ish). Native work off the main thread avoids native jank (Module 21).

Advantages / Disadvantages

- **+** Standard, async, cross-platform bridge to any native capability; multiple codecs; three channel types.
- **−** Untyped (cast/serialize), async-only, threading pitfalls, serialization cost, name-matching fragility — FFI/Pigeon mitigate some.

Interview Questions

1. 🟢 **What is a platform channel?** — A named async message pipe between Dart and native code, with messages serialized by a codec.
2. 🟢 **Are channel calls sync or async?** — Async — they return `Future`s/`Stream`s; never assume synchronous.
3. 🟡 **What data types can cross a standard channel?** — Codec-supported: null, bool, num, String, `Uint8List`/typed lists, `List`, `Map` — not custom classes.
4. 🟡 **Which thread do native handlers run on?** — The platform main thread (Android UI thread / iOS main queue); offload heavy work natively.
5. 🟡 **What are the three channel types?** — `MethodChannel` (call+reply), `EventChannel` (native→Dart streams), `BasicMessageChannel` (arbitrary two-way messages).
6. 🔴 **How do you call channels from a background isolate?** — Register `RootIsolateToken` and `BackgroundIsolateBinaryMessenger.ensureInitialized` — channels aren't wired in background isolates by default.
7. 🔴 **How do you pass a custom object across a channel?** — Serialize it to primitives/maps (or use Pigeon for generated typed messages); the standard codec can't send arbitrary classes.

Senior Engineer Tips

- Namespace channels (`com.yourapp/feature`) and match names/methods exactly; a typo = `MissingPluginException`.
- Keep messages small and calls infrequent; batch where possible and use `Uint8List` for binary.
- Always wrap native calls behind a repository/service so the app is testable (mock the repo) and channels stay isolated.

Architect Perspective

Platform channels are the embedder-boundary integration seam. Isolating them behind repositories (returning domain types, mapping errors) keeps the app portable and testable; understanding the codec/async/threading model prevents the subtle bugs (serialization, main-thread blocking, background isolates) that plague ad-hoc native integration (10 · threading_model, Module 27/28).

Summary

- Channels = named, async, codec-serialized message pipes between Dart and native, crossing platform/UI runners.
- Pass primitives/maps/lists/bytes (not custom classes); handlers run on the native main thread; always async.

- Wrap behind repositories; use Pigeon for types, FFI for direct C, background messenger for isolates.

Revision Notes

- Named channel + codec (Standard: primitives/ `Map` / `List` / `Uint8List`; also JSON/String/Binary); async only.
- Types: `MethodChannel` (call+reply), `EventChannel` (streams), `BasicMessageChannel` (two-way).
- Native handlers on platform main thread → offload heavy work; keep messages small.
- Background isolate → `RootIsolateToken` + background messenger; wrap behind repository; Pigeon (types)/FFI (C).

Practice Questions

1. What data types can cross a standard channel, and how do you send a custom object?
2. Which thread runs native handlers, and why does it matter?
3. Why must channel calls be awaited/handled for errors?

Coding Questions

1. Define a namespaced `MethodChannel` and call a method with typed args/result (with cast).
2. Explain (comments) the serialize→cross-runner→handle→reply flow for a call.
3. Set up background-isolate channel access (token + messenger) conceptually.

Mini Project

Channel model map (docs + skeleton): Write `CHANNELS.md` diagramming the messaging model (codec, runners, threading), listing supported data types and the three channel types, and sketch a repository wrapping a namespaced `MethodChannel` (typed Dart API, error mapping). Acceptance: correct model/threading explanation; supported types listed; repository-wrapped native access; runs (Dart side).

`MethodChannel`

> `MethodChannel` is the request/reply channel: Dart calls `invokeMethod('name', args)` and awaits a result; the native side registers a handler that switches on the method name, does the work, and returns a result or a `PlatformException` — the workhorse for one-off native calls.

Introduction

`MethodChannel` handles "call a native method, get a value back" — battery level, share intent, biometric prompt, native dialog. This file covers the Dart side (`invokeMethod` + error handling) and the native handler pattern (method dispatch, `result.success` / `error` / `notImplemented`), wrapped behind a repository.

Why this concept exists

Most native integration is discrete calls with a result. `MethodChannel` standardizes this: a named channel + method dispatch + typed-ish result/error, async over the channel infrastructure (01_platform_channel_fundamentals.md).

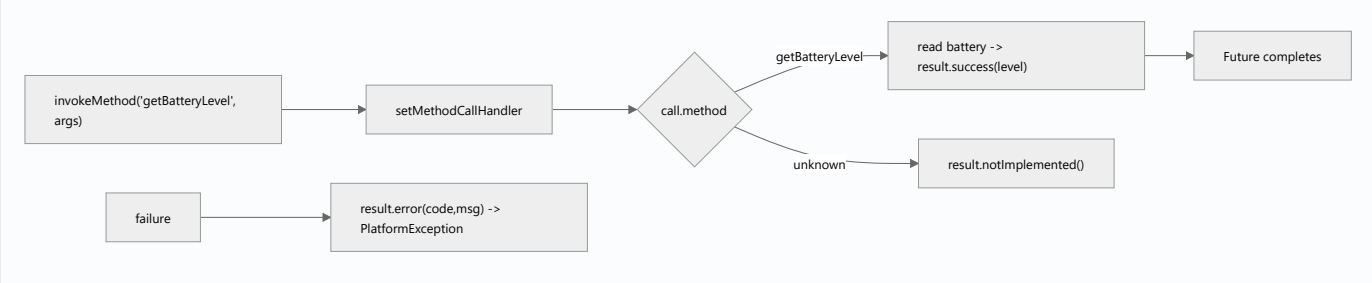
Real-world analogy

`MethodChannel` is **calling a service desk and asking for something specific** ("what's the battery level?"): you state the request by name with details (args), they look it up and reply with the answer (result) or "can't do that" (error). One question, one answer.

Problem Statement

Read the battery level and trigger a native share sheet from Dart, handling the case where the platform doesn't implement it and native errors. You'll implement the Dart caller + native handler + repository wrapper + error handling.

Internal Working



- **Dart side**:
  - `const channel = MethodChannel('app/battery');`
  - `final v = await channel.invokeMethod<int>('getBatteryLevel', {args});` — returns the typed result (cast).
  - **Errors**: catch `PlatformException` (native `result.error`), `MissingPluginException` (no handler registered on this platform), and general errors; map to domain failures.
- **Native side** (registered by the plugin/app):
  - **Android (Kotlin)**: `MethodChannel(binaryMessenger, 'app/battery').setMethodCallHandler { call, result -> when (call.method) { "getBatteryLevel" -> result.success(level); else -> result.notImplemented() } }`
  - **iOS (Swift)**: `channel.setMethodCallHandler { call, result in ... result(level) / result(FlutterError(...)) / result(FlutterMethodNotImplemented) }`
  - Return via **`result.success`** / **`result.error`** / **`result.notImplemented`**; do heavy work off the main thread and call `result` on the main thread.
- **Args**: passed as codec-supported types (usually a `Map`); read/cast on native.
- **Wrap in a repository**: expose a typed Dart API (`BatteryRepository.level()`), map channel results/errors → domain — the app never touches the channel directly.

Memory Representation

Small serialized messages both ways; large data should be `Uint8List` or avoided (01_platform_channel_fundamentals.md).

Compiler Behavior

`invokeMethod<T>` returns `T?` but isn't truly type-checked end-to-end (native returns are dynamic) — Pigeon adds real type-safety (04_plugins_and_pigeon.md).

Runtime Behavior

`invokeMethod` sends the call and awaits; native handler dispatches by method name; unknown methods → `notImplemented` → Dart `MissingPluginException`; native errors → `PlatformException`.

Flutter Engine Behavior

Message crosses platform/UI runners; native handler runs on the platform main thread (10 · threading_model).

Dart VM Behavior

Dart caller awaits on the root isolate; background isolates need messenger setup (02 · isolates).

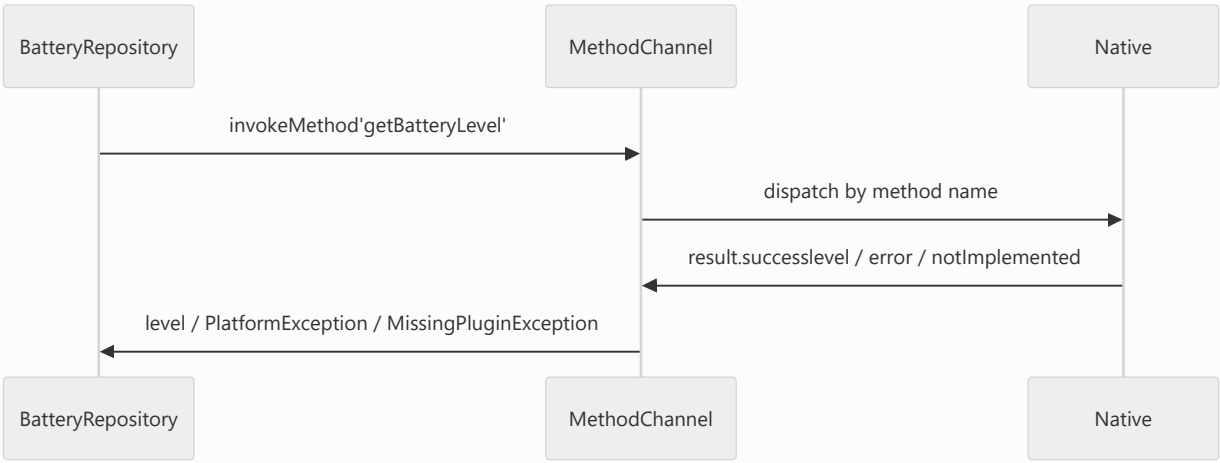
Examples

```dart
import 'package:flutter/services.dart';


// Repository wrapping the channel (typed API + error mapping)
class BatteryRepository {
  static const _channel = MethodChannel('app/battery');


  Future<int> level() async {
    try {
      final level = await _channel.invokeMethod<int>('getBatteryLevel');
      return level ?? -1;
    } on MissingPluginException {
      throw UnsupportedError('Battery API not available on this platform');
    } on PlatformException catch (e) {
      throw StateError('Battery error: ${e.code} ${e.message}'); // map native error
    }
  }


  Future<void> share(String text) =>
      _channel.invokeMethod('share', {'text': text});
}
```

```kotlin
// Android (Kotlin) — register handler (e.g., in MainActivity.configureFlutterEngine)
MethodChannel(flutterEngine.dartExecutor.binaryMessenger, "app/battery")
  .setMethodCallHandler { call, result ->
    when (call.method) {
      "getBatteryLevel" -> {
        val bm = getSystemService(Context.BATTERY_SERVICE) as BatteryManager
        result.success(bm.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPACITY))
      }
      "share" -> { /* start share intent */ result.success(null) }
      else -> result.notImplemented()   // unknown method
    }
  }
```

```
// iOS (Swift) — register handler in AppDelegate/FlutterViewController
let channel = FlutterMethodChannel(name: "app/battery", binaryMessenger: controller.binaryMessenger)
channel.setMethodCallHandler { call, result in
  switch call.method {
  case "getBatteryLevel":
    UIDevice.current.isBatteryMonitoringEnabled = true
    result(Int(UIDevice.current.batteryLevel * 100))
  default:
    result(FlutterMethodNotImplemented)
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Not handling `PlatformException` / `MissingPluginException`	Crashes/uncaught	Catch + map to domain failures
Assuming a result type without cast	Runtime type errors	`invokeMethod<T>` + validate
Blocking the native main thread	Native jank/ANR	Offload native work; call `result` on main
Method name mismatch	`notImplemented` /missing	Match names exactly; namespace
Calling channel directly from UI	Coupling/untestable	Wrap in a repository
Passing custom objects	Codec unsupported	Send maps/primitives

Best Practices

- **Wrap in a repository** exposing a typed Dart API; map `PlatformException` / `MissingPluginException` → domain failures.
- Handle the **not-implemented** case (platform without the handler) gracefully.
- On native, **dispatch by method name**, return via `result.success/error/notImplemented`, and **offload heavy work** (call `result` on the main thread).
- Pass **codec-supported args** (a `Map`); validate/cast results.
- Namespace channels; for type-safety across the boundary, use **Pigeon** (04_plugins_and_pigeon.md).

Performance

Small request/reply latency (serialize + cross-thread) — keep calls off hot paths and results small; offload heavy native work so `result` returns quickly (Module 21).

Advantages / Disadvantages

- **+** Simple request/reply to native, structured results/errors, cross-platform, wraps any native SDK.
- **−** Untyped (cast), async-only, error/threading handling required, name-matching fragility (Pigeon fixes typing).

Interview Questions

1. 🟢 **What is `MethodChannel` for?** — Calling a native method by name from Dart and awaiting a result (request/reply).
2. 🟢 **How does the native side respond?** — A handler dispatches by `call.method` and returns `result.success(value)`, `result.error(...)`, or `result.notImplemented()`.
3. 🟡 **What Dart exceptions must you handle?** — `PlatformException` (native error), `MissingPluginException` (no handler on this platform), plus general errors — map to domain failures.
4. 🟡 **How do you pass arguments?** — As codec-supported types (usually a `Map`), read/cast on the native side.
5. 🟡 **Which thread runs the native handler, and what's the risk?** — The platform main thread; heavy work there blocks native UI — offload and return via `result` on main.
6. 🔴 **How do you make method calls type-safe across the boundary?** — Use **Pigeon** to generate typed interfaces/codecs instead of stringly-typed `invokeMethod`.
7. 🔴 **Why wrap `MethodChannel` in a repository?** — To expose a typed, testable domain API and keep channel details (names/casts/errors) out of the app.

Senior Engineer Tips

- Centralize each native feature in a repository with a typed API + error mapping; screens/logic never see `MethodChannel`.
- Always handle `MissingPluginException` — the same code runs on platforms that may lack the handler (e.g., web/desktop).
- Do native heavy-lifting off the main thread and reply on the main thread; forgetting this causes ANRs/jank.

Architect Perspective

`MethodChannel` is the primary discrete native-call mechanism; behind repositories (typed API, mapped errors) it keeps the app portable/testable. For growing native surfaces, graduate to **plugins + Pigeon** for type-safety and reuse; for streaming events use `EventChannel` (03_event_channel.md); for C libraries use FFI (05_ffi.md).

Summary

- `MethodChannel`: Dart `invokeMethod('name', args)` → native handler dispatches by name → `result.success/error/notImplemented` → Dart `Future` / `PlatformException` / `MissingPluginException`.
- Pass codec types, offload native heavy work, handle errors, wrap in a repository (typed API).
- Untyped by default — use Pigeon for type-safety; namespace channels.

Revision Notes

- Dart: `channel.invokeMethod<T>('name', mapArgs)`; catch `PlatformException` / `MissingPluginException`.
- Native: `setMethodCallHandler` → switch `call.method` → `result.success/error/notImplemented`; offload heavy work, reply on main.
- Codec-supported args (Map/primitives); wrap in repository (typed API + error mapping).
- Untyped → Pigeon for types; namespace channels.

Practice Questions

1. How does the native side return a value vs an error vs not-implemented?
2. Which Dart exceptions must you handle and why?
3. Why wrap the channel in a repository?

Coding Questions

1. Build a `BatteryRepository` over a `MethodChannel` (typed `level()`, error mapping).

2. Write the Android/iOS handler dispatching `getBatteryLevel` / `share`.
3. Handle `MissingPluginException` for a platform without the handler.

Mini Project

Battery + share bridge (Flutter + native): Implement a `MethodChannel` for `getBatteryLevel` and `share`, native handlers (Kotlin/Swift) that offload work and return results/errors, and a `BatteryRepository` exposing a typed Dart API with error mapping (incl. `MissingPluginException`). Acceptance: request/reply works; errors mapped; native work off main thread; repository-wrapped; runs on a device. (Streams continue in 03_event_channel.md.)

`EventChannel`

> `EventChannel` streams **native→Dart** events over time: Dart `receiveBroadcastStream()` gives a `Stream`, and the native side pushes events via an event sink (`onListen` / `onCancel`) — the channel for continuous native data like sensor readings, battery/charging changes, connectivity, and location updates.

Introduction

Where `MethodChannel` is one-shot request/reply, `EventChannel` is a **stream** of native events. This file covers the Dart side (`receiveBroadcastStream` → `Stream`, subscription lifecycle) and the native side (`StreamHandler` with `onListen` / `onCancel` + event sink), wrapped behind a repository.

Why this concept exists

Many native sources emit *ongoing* data (accelerometer, GPS, battery state, connectivity, BLE). Polling via `MethodChannel` is wasteful/laggy; `EventChannel` lets native **push** events to a Dart `Stream`, with proper start/stop lifecycle tied to subscription.

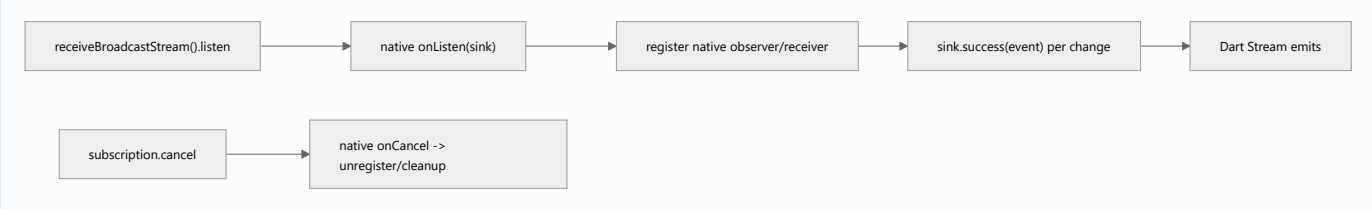
Real-world analogy

`EventChannel` is **subscribing to a live feed**: you tune in (`listen`), the broadcaster starts sending updates (`onListen` → sink events), and when you tune out (`cancel`), they stop (`onCancel`). Continuous, not one question.

Problem Statement

Stream battery-level/charging changes from native to a Dart `Stream`, starting the native listener when subscribed and stopping it (and cleaning up native resources) when cancelled — behind a repository. You'll implement both sides + subscription lifecycle.

Internal Working



- **Dart side**:

    - `const events = EventChannel('app/battery/events');`
    - `Stream<dynamic> get stream => events.receiveBroadcastStream([optionalArgs]);`
    - `listen(onData, onError, onDone)`; **cancel** the subscription to stop (and to avoid leaks — 08 · dispose_and_leaks). Errors arrive as `PlatformException` on the stream.
- **Native side** (a `StreamHandler`):
    - `onListen(arguments, eventSink)`: start emitting — register the native observer/receiver and call `eventSink.success(value)` on each event (`.error(...)` for failures). Runs when the first listener subscribes.
    - `onCancel(arguments)`: stop and **release native resources** (unregister receivers/observers) when the last listener cancels.
    - Emit on the **main thread** (marshal from background callbacks).
- **Broadcast**: `receiveBroadcastStream` supports multiple listeners; native `onListen` / `onCancel` fire on first-subscribe/last-cancel.
- **Repository**: expose a typed `Stream<BatteryState>` (map raw events → domain), and ensure consumers cancel subscriptions.

Memory Representation

The native observer/receiver + the Dart subscription are live resources; not cancelling leaks both (and keeps native sensors running/battery drain) (08 · dispose_and_leaks).

Compiler Behavior

Events are untyped (`dynamic`) — cast/map on the Dart side (Pigeon can type this).

Runtime Behavior

First `listen` triggers native `onListen` (start); last `cancel` triggers `onCancel` (stop). Native pushes events asynchronously; errors surface as stream errors.

Flutter Engine Behavior

Events cross the platform/UI runners; native emits on the main thread (marshal sensor callbacks) (10 · threading_model).

Dart VM Behavior

Stream events dispatch on the root isolate's event loop; keep handlers light (02 · streams).

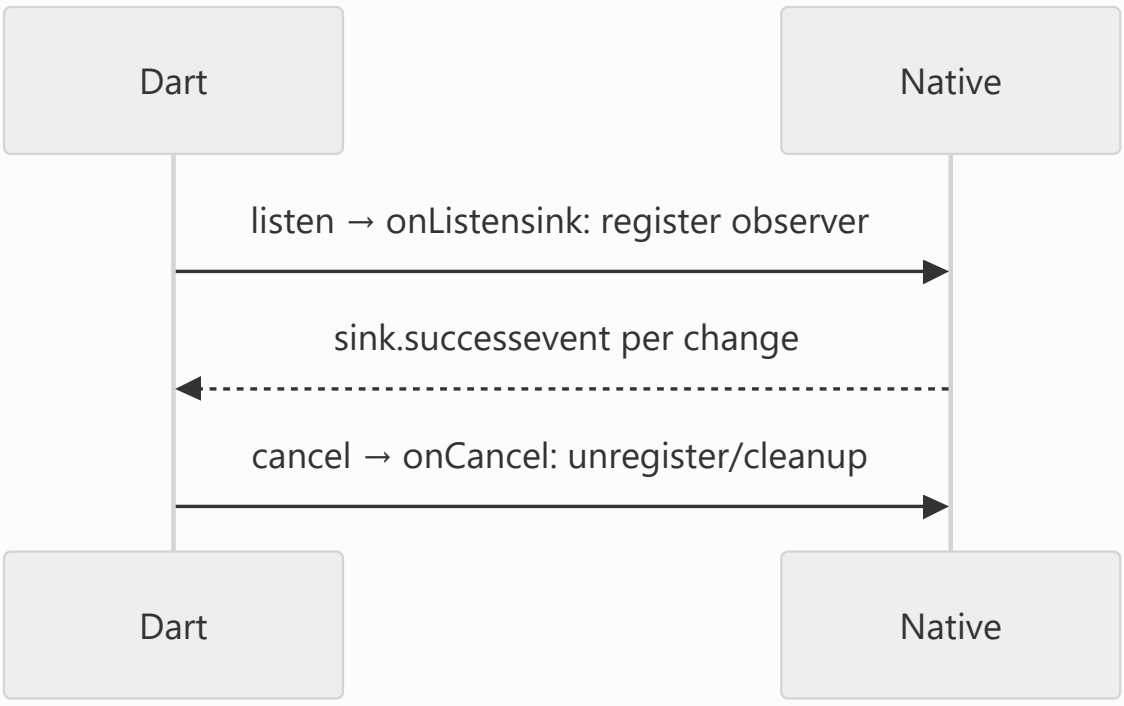
Examples

```
import 'package:flutter/services.dart';

// Repository exposing a typed Stream (native events mapped to domain)
enum ChargingStatus { charging, discharging, full, unknown }

class BatteryEventsRepository {
  static const _events = EventChannel('app/battery/events');

  Stream<ChargingStatus> chargingStatus() =>
      _events.receiveBroadcastStream().map((event) => switch (event as String) {
            'charging' => ChargingStatus.charging,
            'discharging' => ChargingStatus.discharging,
            'full' => ChargingStatus.full,
            _ => ChargingStatus.unknown,
          });
}
// Consumer: final sub = repo.chargingStatus().listen(...); ... sub.cancel(); // stop native
```

```
// Android (Kotlin) — StreamHandler starts/stops a BroadcastReceiver
class ChargingStreamHandler(private val context: Context) : EventChannel.StreamHandler {
  private var receiver: BroadcastReceiver? = null
  override fun onListen(args: Any?, events: EventChannel.EventSink) {
    receiver = object : BroadcastReceiver() {
      override fun onReceive(c: Context, i: Intent) {
        events.success(if (/* charging */ true) "charging" else "discharging") // emit
      }
    }
    context.registerReceiver(receiver, IntentFilter(Intent.ACTION_BATTERY_CHANGED))
  }
  override fun onCancel(args: Any?) {
    context.unregisterReceiver(receiver) // cleanup native resource
    receiver = null
  }
}
// EventChannel(messenger, "app/battery/events").setStreamHandler(ChargingStreamHandler(context))
```

```
// iOS (Swift) — FlutterStreamHandler with NotificationCenter observer
class ChargingStreamHandler: NSObject, FlutterStreamHandler {
  var sink: FlutterEventSink?
  func onListen(withArguments args: Any?, eventSink: @escaping FlutterEventSink) -> FlutterError? {
    sink = eventSink
    UIDevice.current.isBatteryMonitoringEnabled = true
    NotificationCenter.default.addObserver(self, selector: #selector(changed),
      name: UIDevice.batteryStateDidChangeNotification, object: nil)
    return nil
  }
  @objc func changed() { sink?( /* map state */ "charging") }
  func onCancel(withArguments args: Any?) -> FlutterError? {
    NotificationCenter.default.removeObserver(self); sink = nil; return nil // cleanup
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Not cancelling the subscription	Leaks + native sensor keeps running (battery)	`cancel()` in `dispose`; `onCancel` cleans native
No cleanup in `onCancel`	Native observer/receiver leaks	Unregister in `onCancel`
Emitting off the main thread	Crashes/threading issues	Marshal to main before `sink.success`
Using `MethodChannel` polling for streams	Wasteful/laggy	Use `EventChannel`
Untyped events unmapped	Runtime cast errors	Map events → domain in a repository
Heavy work in Dart `onData`	Jank	Keep handlers light; offload

Best Practices

- Use `EventChannel` for **continuous native events**; expose a typed `Stream` via a **repository** (map events → domain).
- **Cancel subscriptions** (in `dispose`) — this triggers native `onCancel` to **release resources** (stop sensors, unregister receivers) and prevent battery drain.
- On native, **start in `onListen`, stop/cleanup in `onCancel`**, and **emit on the main thread**.
- Handle stream **errors** (`PlatformException`); keep Dart handlers light.
- Consider `connectivity_plus` / `sensors_plus` / `battery_plus` plugins before hand-rolling (Module 29).

Performance

Push-based is efficient vs polling; native sensor/observer registration costs battery — stop on cancel/background (08 · app_lifecycle). Keep handlers light and events small (Module 21).

Advantages / Disadvantages

- **+** Native→Dart streaming with start/stop lifecycle, efficient for continuous data, multiple listeners (broadcast).
- **−** Subscription/resource lifecycle to manage (leaks/battery), untyped events, threading care — Pigeon/plugins mitigate.

Interview Questions

1. 🟢 **What is `EventChannel` for?** — Streaming continuous native→Dart events (sensors, battery, connectivity) as a Dart `Stream`.

2. 🟢 **How do you consume it in Dart?** — `channel.receiveBroadcastStream().listen(...)`, cancelling the subscription to stop.
3. 🟡 **What do `onListen` / `onCancel` do natively?** — `onListen` starts emitting (register observer, use the event sink); `onCancel` stops and releases native resources.
4. 🟡 **Why must you cancel subscriptions?** — To trigger `onCancel` (stop native sensors/receivers → save battery) and avoid leaks; not cancelling keeps native resources alive.
5. 🟡 **How do errors surface?** — As `PlatformException` events on the stream (native `sink.error`).
6. 🔴 **`EventChannel` vs `MethodChannel` for continuous data?** — `EventChannel` pushes events efficiently; `MethodChannel` polling is wasteful/laggy — use `EventChannel` for streams.
7. 🔴 **What threading rule applies to emitting events?** — Emit on the platform main thread; marshal from background sensor callbacks before calling the sink.

Senior Engineer Tips

- Treat the native `onListen` / `onCancel` pair as acquire/release — always unregister in `onCancel`; pair with Dart subscription cancel in `dispose`.
- Pause/stop streams on app background for battery (08 · app_lifecycle); resume on foreground.
- Prefer maintained `*_plus` plugins (battery/sensors/connectivity) over hand-rolled `EventChannel`s unless you need custom native behavior.

Architect Perspective

`EventChannel` is the streaming native-integration primitive; behind a repository (typed `Stream`, lifecycle-managed) it powers reactive features (sensors/connectivity/location) cleanly. Its acquire/release lifecycle (`onListen` / `onCancel` ↔ subscribe/cancel) is a resource-management concern that, done right, avoids leaks and battery drain (08 · dispose_and_leaks, Module 29).

Summary

- `EventChannel` streams native→Dart events: `receiveBroadcastStream` → `Stream`; native `onListen` (start/emit via sink) / `onCancel` (stop/cleanup).
- Cancel subscriptions (stops native, saves battery); emit on main thread; map events → domain in a repository.
- Use for continuous data; prefer `*_plus` plugins where available.

Revision Notes

- Dart: `EventChannel.receiveBroadcastStream().listen(...)`; cancel to stop; errors = `PlatformException`.
- Native: `StreamHandler` — `onListen(sink)` start/emit, `onCancel` stop+unregister; emit on main thread.

- Cancel → native cleanup (leaks/battery); wrap in repository (typed Stream + domain mapping).
- Continuous data → `EventChannel` (not polling); prefer `*_plus` plugins.

Practice Questions

1. What do `onListen` / `onCancel` correspond to, and why cancel subscriptions?
2. `EventChannel` vs `MethodChannel` for sensor data?
3. How do stream errors arrive on the Dart side?

Coding Questions

1. Build a `BatteryEventsRepository` exposing a typed `Stream` over an `EventChannel`.
2. Write the native `StreamHandler` (register in `onListen`, unregister in `onCancel`).
3. Ensure the consumer cancels the subscription in `dispose` and pauses on background.

Mini Project

Charging-state stream (Flutter + native): Implement an `EventChannel` streaming charging/battery changes, native `StreamHandler`s (Kotlin/Swift) that register in `onListen` and cleanup in `onCancel` (emit on main), and a `BatteryEventsRepository` exposing a typed `Stream<ChargingStatus>`; the consumer cancels on dispose and pauses on background. Acceptance: real-time events; native cleanup on cancel; no leaks/battery drain; repository-wrapped; runs on device.

Plugins & Pigeon (Type-Safe Channels)

> Package reusable native integration as a **plugin** (a federated package with platform implementations); use **Pigeon** to generate type-safe Dart↔native interfaces from a schema — eliminating stringly-typed `invokeMethod` and hand-written serialization/dispatch.

Introduction

Ad-hoc channels don't scale or share. **Plugins** package native code + Dart API for reuse (and publishing); the **federated plugin** model splits per-platform implementations. **Pigeon** generates typed message code (Dart + Kotlin/Swift) from a Dart schema, replacing manual channels with compile-checked interfaces. This file covers both.

Why this concept exists

Raw `MethodChannel` is untyped, error-prone (name/type mismatches), and not reusable. Plugins make native integration a shareable package; Pigeon makes the Dart↔native contract **type-safe and generated** — fewer bugs, less boilerplate, and refactor-safety across the boundary.

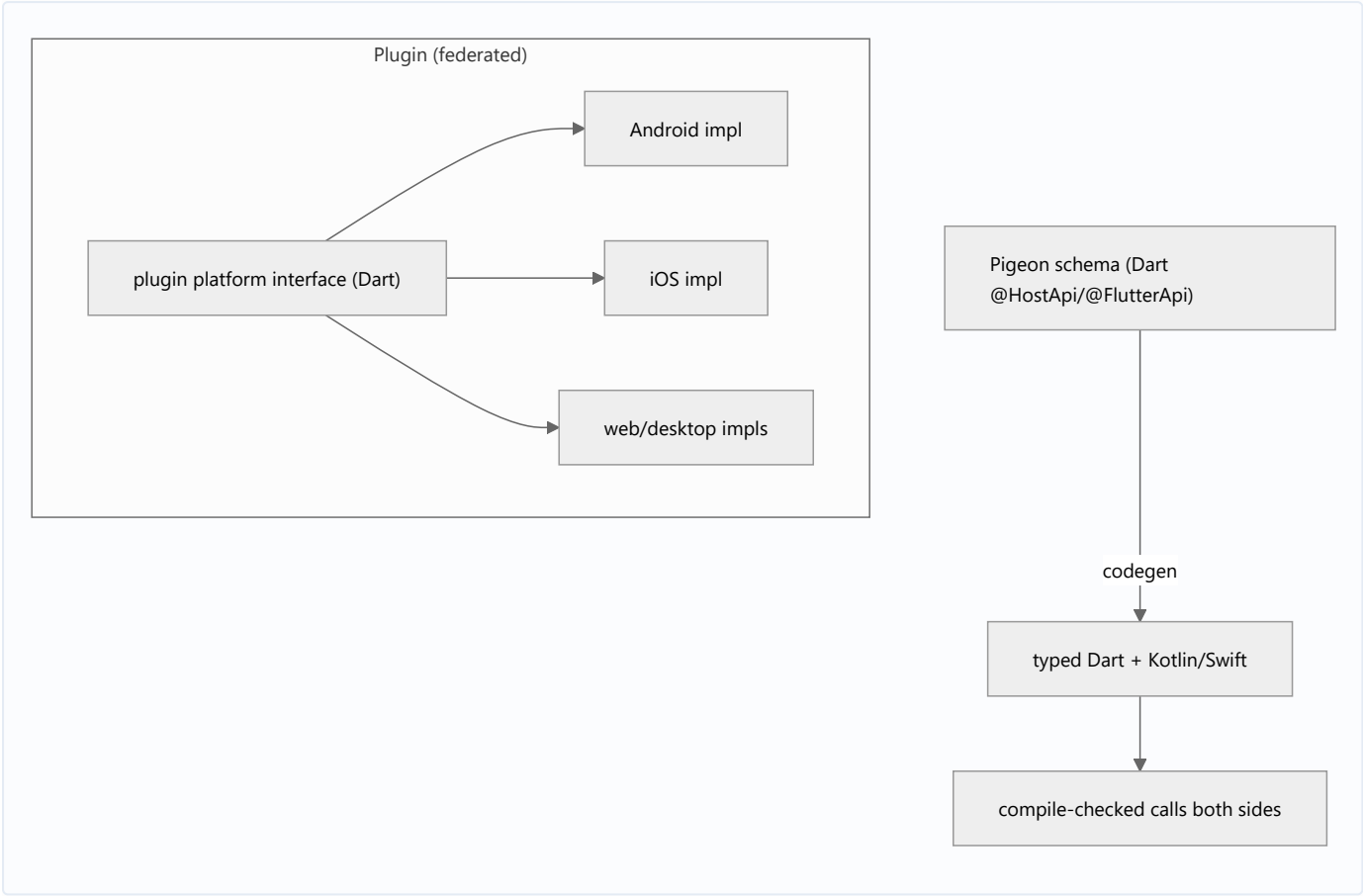
Real-world analogy

A plugin is a **sealed appliance** you install and use via its interface (not its wiring). Pigeon is having an **engineer auto-generate matching plugs/sockets on both ends from a spec** — guaranteed to fit — instead of soldering wires by hand and hoping the pins line up.

Problem Statement

You have growing native calls (battery, share, device info) that are untyped and duplicated. Package them as a plugin and generate a type-safe interface with Pigeon so Dart and native share a checked contract. You'll structure the plugin + define a Pigeon schema.

Internal Working



- **Plugin structure**: `flutter create --template=plugin`; a plugin exposes a Dart API and native implementations. **Federated plugins** split into: `*_platform_interface` (abstract API), platform packages (`*_android`, `*_ios`, `*_web`, ...), and an app-facing package — so platforms evolve independently and third parties can add support.
- **Registration**: plugins register native handlers automatically (via `GeneratedPluginRegistrant`); the Dart API wraps the channels.
- **Pigeon**: define the contract in a Dart file with typed classes + `@HostApi()` (Dart→native) / `@FlutterApi()` (native→Dart) interfaces; run `pigeon` to **generate** the Dart API + native (Kotlin/Swift) protocol + codec. You implement the native protocol; Dart calls typed methods — **no** `invokeMethod` **strings, no manual serialization**.
- **Benefits**: compile-time type checks on both sides, generated (de)serialization, refactor-safe, less boilerplate; supports typed classes (which raw channels can't).
- **When**: use a **plugin** for reusable/publishable native features; use **Pigeon** for any non-trivial typed Dart↔native API; raw channels only for quick one-offs.

Memory Representation

Pigeon generates efficient typed codecs; plugins are ordinary packages. Same message-passing model underneath (01_platform_channel_fundamentals.md).

Compiler Behavior

Pigeon's generated Dart/native code is **compile-checked** — mismatched signatures/types fail at build, not runtime (unlike raw channels). Requires a codegen step (`dart run pigeon ...`).

Runtime Behavior

Typed calls serialize via the generated codec and dispatch to the implemented native methods; errors are typed/structured. Plugins auto-register their handlers.

Flutter Engine Behavior

Same platform/UI runner model; native implementations run on the platform side (10 · threading_model).

Dart VM Behavior

Not applicable.

Examples

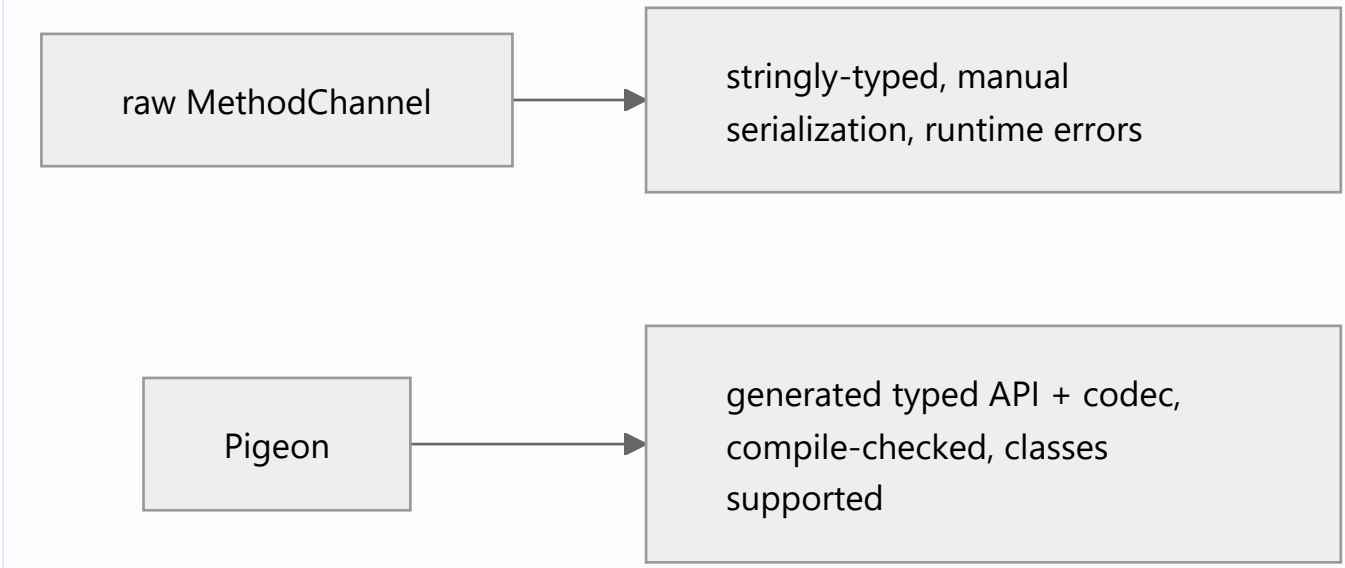
```
// pigeons/messages.dart — the SCHEMA (run: dart run pigeon --input pigeons/messages.dart ...)
import 'package:pigeon/pigeon.dart';

class DeviceInfo {            // typed class crosses the boundary (raw channels can't)
  late String model;
  late int sdkVersion;
  late double batteryLevel;
}

@HostApi()                    // Dart -> native
abstract class DeviceApi {
  DeviceInfo getInfo();       // typed, compile-checked (no invokeMethod strings)
  void share(String text);
}

@FlutterApi()                 // native -> Dart (callbacks/events)
abstract class DeviceEvents {
  void onBatteryChanged(double level);
}
```

```
// App side — use the GENERATED typed API (no channels/casts):
// final api = DeviceApi();                  // generated
// final info = await api.getInfo();          // typed DeviceInfo, checked
// print('${info.model} ${info.batteryLevel}');
```

```
// Android — implement the GENERATED DeviceApi interface (Kotlin), register it.
// class DeviceApiImpl : DeviceApi { override fun getInfo(): DeviceInfo { ... } }
// DeviceApi.setUp(binaryMessenger, DeviceApiImpl())
```

```
Plugin scaffolding:
  flutter create --template=plugin --platforms=android,ios,web my_device_plugin
Pigeon codegen (Dart + Kotlin + Swift):
  dart run pigeon --input pigeons/messages.dart \
    --dart_out lib/messages.g.dart \
    --kotlin_out android/.../Messages.g.kt \
    --swift_out ios/.../Messages.g.swift
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Stringly-typed channels for a big API	Runtime errors, boilerplate	Use Pigeon (typed, generated)
Not federating a reusable plugin	Hard to extend per platform	Federated structure (`*_platform_interface` + impls)
Forgetting to run pigeon codegen	Stale generated code	Regenerate on schema change (CI)
Custom objects over raw channels	Codec can't send classes	Pigeon supports typed classes
Duplicating native code across apps	Not reusable	Package as a plugin
App depending on generated API directly everywhere	Coupling	Still wrap behind a repository/domain API

Best Practices

- Package **reusable/publishable** native features as **plugins**; use the **federated** structure for multi-platform maintainability.
- Use **Pigeon** for any non-trivial typed Dart↔native API — get compile-time safety, generated serialization, and support for typed classes; **automate codegen** (CI/pre-commit).
- Keep raw `MethodChannel` / `EventChannel` for quick one-offs; graduate to Pigeon/plugins as the surface grows.
- Still **wrap the generated/plugin API behind a repository/domain layer** so the app depends on domain types.
- Handle platform-not-supported gracefully (federated impls or fallbacks).

Performance

Pigeon's generated codec is efficient; plugins add no overhead beyond the channel model. Type-safety prevents costly runtime bugs. Same threading/serialization considerations apply (01_platform_channel_fundamentals.md).

Advantages / Disadvantages

- **+ Plugins:** reusable/publishable, per-platform (federated), auto-registered. **+ Pigeon:** type-safe, generated, class support, refactor-safe, less boilerplate.
- **−** Codegen/build step + more structure; overkill for a single tiny call; still need repository wrapping.

Interview Questions

1. 🟢 **What is a Flutter plugin?** — A package bundling a Dart API + native implementations, providing reusable native integration.
2. 🟢 **What does Pigeon do?** — Generates type-safe Dart↔native interfaces + codecs from a Dart schema, replacing stringly-typed `invokeMethod` and manual serialization.
3. 🟡 **What is a federated plugin?** — A plugin split into a platform-interface package + per-platform implementation packages (+ app-facing), so platforms evolve independently and third parties can add support.
4. 🟡 `@HostApi` **vs** `@FlutterApi` **in Pigeon?** — `@HostApi`: Dart calls native; `@FlutterApi`: native calls Dart (callbacks/events).
5. 🟡 **Why is Pigeon safer than raw channels?** — Signatures/types are compile-checked on both sides (build errors, not runtime) and it supports typed classes the standard codec can't.
6. 🔴 **When use raw channels vs Pigeon vs a plugin?** — Raw channels for quick one-offs; Pigeon for non-trivial typed APIs; plugins for reusable/publishable native features (often plugin + Pigeon together).
7. 🔴 **Do you still wrap a plugin/Pigeon API in a repository?** — Yes — to depend on domain types and keep the plugin swappable/testable.

Senior Engineer Tips

- For any native API beyond a call or two, reach for **Pigeon** immediately — it removes the entire class of channel name/type bugs.
- Structure shared native features as **federated plugins**; automate pigeon codegen in CI so generated code never drifts.
- Even with generated APIs, wrap them behind a repository so the app depends on your domain, not the plugin.

Architect Perspective

Plugins + Pigeon turn native integration into a typed, reusable, maintainable layer — critical as native surface grows or when publishing packages. Federation supports multi-platform ownership; Pigeon's type-safety mirrors the codegen-over-reflection theme (02 · dart_compilation). Behind repositories, it keeps the app portable and testable (Modules 27/28).

Summary

- Plugins package reusable native integration (federated for multi-platform); Pigeon generates type-safe Dart↔native APIs (+ codecs, class support).
- Use Pigeon for non-trivial typed APIs, plugins for reuse/publishing; automate codegen.

- Still wrap behind repositories; raw channels only for quick one-offs.

Revision Notes

- Plugin: Dart API + native impls; **federated** = platform-interface + per-platform packages; auto-registered.
- Pigeon: Dart schema (`@HostApi` Dart→native, `@FlutterApi` native→Dart, typed classes) → generated Dart+Kotlin+Swift + codec (compile-checked); automate codegen.
- Use Pigeon (non-trivial typed) / plugin (reusable) / raw channels (one-offs); wrap behind repository.

Practice Questions

1. Why is Pigeon safer/less error-prone than raw `MethodChannel`?
2. What is a federated plugin and why use it?
3. When would you still use a raw channel?

Coding Questions

1. Write a Pigeon schema (`@HostApi` + a typed class) and note the codegen command.
2. Scaffold a federated plugin structure (packages) for a native feature.
3. Wrap the generated Pigeon API behind a domain repository.

Mini Project

Type-safe device plugin (Flutter + native): Define a Pigeon schema (`DeviceApi.getInfo()` / `share()` + a `DeviceInfo` class, `@FlutterApi` battery callback), generate Dart+Kotlin+Swift, implement the native side, and wrap the generated API in a `DeviceRepository` (domain types). Package it as a plugin. Acceptance: compile-checked typed calls (no `invokeMethod`); typed class crosses the boundary; repository-wrapped; codegen automated; runs on device.

FFI (`dart:ffi` — Direct C Interop)

> FFI lets Dart call **C** functions directly — synchronously, in-process, with no channel/serialization overhead — ideal for using native C/C++ libraries (crypto, compression, image processing, existing SDKs); `ffigen` generates the Dart bindings from C headers.

Introduction

`dart:ffi` (Foreign Function Interface) binds Dart to native C ABIs: load a shared library, look up functions, and call them with C-compatible types — no platform channels, no serialization. This file covers when to use FFI vs channels, the binding mechanics, `ffigen`, memory management, and threading.

Why this concept exists

Channels are great for platform APIs but add async + serialization overhead and can't directly use C libraries. Many high-performance or existing native libraries are C/C++; FFI calls them **directly and synchronously**, avoiding the channel round-trip — the right tool for heavy computation or reusing C code.

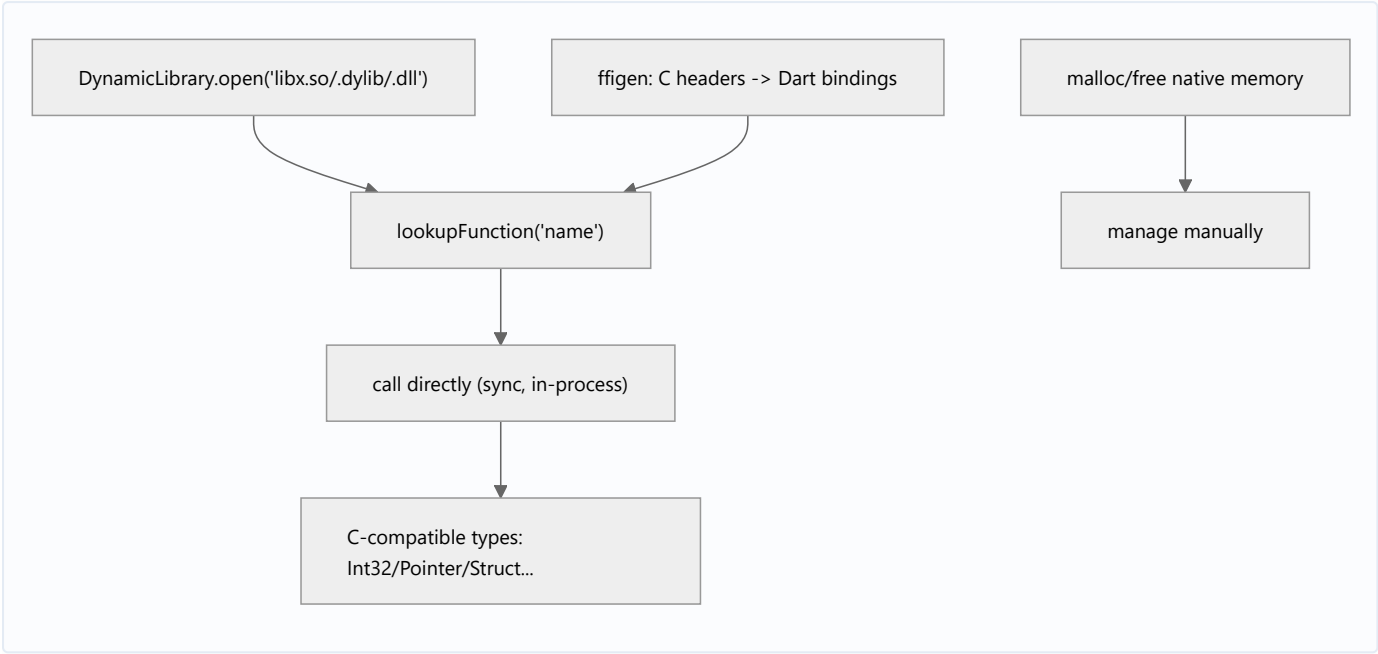
Real-world analogy

Channels are **mailing a request to another department** (async, serialized); FFI is **calling a colleague's function directly in the same office** — instant, in-process, no envelopes. But you must speak their exact language (C types) and manage shared resources (memory) yourself.

Problem Statement

You need to use a C crypto/compression library (or hand-written C) from Dart with minimal overhead — not a platform API. You'll load a shared lib, bind a function (via `ffigen`), call it, and manage native memory correctly.

Internal Working



- **Load the library**: `DynamicLibrary.open('libfoo.so')` (Android), `.process()` /framework on iOS/macOS, `.dll` on Windows — bundle the native lib with the app/plugin.
- **Bind functions**: `lookupFunction<NativeSig, DartSig>('c_name')` maps a C function to a callable Dart function; types must be **C-compatible** (`Int32` , `Double` , `Pointer<Utf8>` , `Pointer<Struct>` , etc. from `dart:ffi` / `package:ffi`).
- `ffigen` : generates Dart bindings from C headers automatically (types, structs, functions) — the practical way to bind non-trivial libraries.
- **Memory management**: you **allocate/free native memory** (`malloc` / `calloc` / `free` from `package:ffi`) for strings/structs/buffers passed to C; Dart's GC doesn't manage native memory — leaks/UAF are on you. Convert strings via `toNativeUtf8` / `toDartString` .
- **Synchronous + in-process**: FFI calls run **on the calling isolate synchronously** — long C calls **block** that isolate (jank if UI isolate). Run heavy FFI on a **background isolate** (`Isolate.run`) (02 · isolates).
- **Callbacks**: C can call back into Dart via `NativeCallable` / `Pointer.fromFunction` (with constraints).
- **Not for platform APIs**: use channels/plugins for OS/platform features; FFI is for C/C++ libraries and perf-critical code.

Memory Representation

Native (C) memory is **outside** the Dart heap — manually allocated/freed; Dart objects wrap pointers. Mismanagement leaks or crashes (use-after-free). Big buffers avoid channel serialization cost.

Compiler Behavior

`ffigen` generates typed bindings at build; FFI signatures are checked against declared C/Dart types (mismatches are your responsibility and can crash if wrong). AOT-compatible (no reflection).

Runtime Behavior

FFI calls execute synchronously in-process; wrong types/ABI or bad pointers crash natively. No async unless you offload to an isolate. Native memory must be freed.

Flutter Engine Behavior

FFI is independent of the platform-channel/embedder path — it's the Dart runtime calling C directly (10 · engine_internals).

Dart VM Behavior

FFI runs on the calling isolate; long calls block it — use `Isolate.run` for heavy work (02 · isolates).

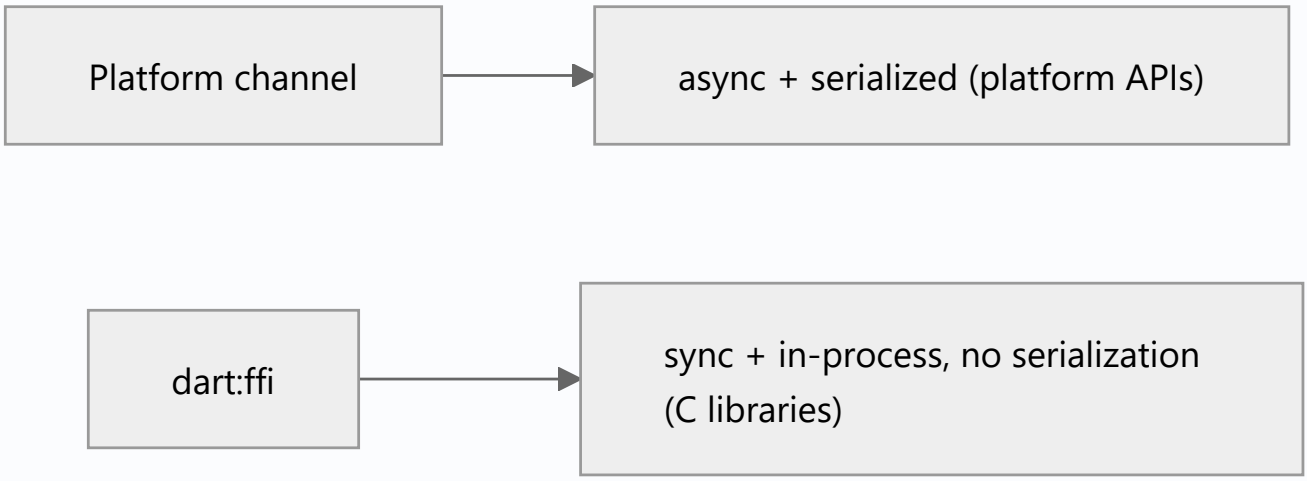
Examples

```
import 'dart:ffi';
import 'package:ffi/ffi.dart'; // malloc, Utf8

// C: int32_t add(int32_t a, int32_t b);
typedef _AddC = Int32 Function(Int32, Int32);
typedef _AddDart = int Function(int, int);

// C: int32_t str_len(const char* s);
typedef _StrLenC = Int32 Function(Pointer<Utf8>);
typedef _StrLenDart = int Function(Pointer<Utf8>);

class NativeMath {
  late final DynamicLibrary _lib;
  late final _AddDart add;
  late final _StrLenDart _strLen;

  NativeMath() {
    _lib = DynamicLibrary.open('libnativemath.so'); // load shared lib
    add = _lib.lookupFunction<_AddC, _AddDart>('add'); // bind (sync, direct)
```

```
    _strLen = _lib.lookupFunction<_StrLenC, _StrLenDart>('str_len');
  }

  int stringLength(String s) {
    final ptr = s.toNativeUtf8();        // allocate native memory
    try {
      return _strLen(ptr);
    } finally {
      malloc.free(ptr);                  // MUST free native memory
    }
  }
}

// Heavy FFI work off the UI isolate:
// final result = await Isolate.run(() => NativeMath().add(2, 3));
```

```
# For non-trivial libs, generate bindings with ffigen (dev dependency):
# dart run ffigen --config ffigen.yaml   # reads C headers -> Dart bindings
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Not freeing native memory	Leaks / crashes	`malloc.free` in `finally`; own the lifecycle
Heavy FFI on the UI isolate	Blocks/janks (synchronous)	Run on a background isolate
Wrong C/Dart type signatures/ABI	Native crash/corruption	Match types exactly; use `ffigen`
Using FFI for platform APIs	Wrong tool	Use channels/plugins for OS features
Forgetting to bundle the native lib	`DynamicLibrary.open` fails	Include/build the lib per platform
Use-after-free on pointers	Crash	Careful pointer lifetime management

Best Practices

- Use FFI for **C/C++ libraries and perf-critical code**, not platform APIs (use channels for those).
- Generate bindings with `ffigen` for anything non-trivial (types/structs/functions from headers).
- **Manage native memory** explicitly (`malloc` / `free` , `finally`); convert strings via `toNativeUtf8` / `toDartString` .
- Run **heavy/blocking FFI on a background isolate** (`Isolate.run`) to keep the UI responsive.
- **Bundle/build** the native library per platform; wrap FFI behind a **repository/service** (safe Dart API, memory handled internally).

Performance

FFI is **fast** (no serialization/async round-trip) — ideal for heavy computation/large buffers. But it's synchronous: offload long calls to an isolate. Native memory management avoids leaks that would degrade the app (Module 21).

Advantages / Disadvantages

- **+** Direct, synchronous, low-overhead C interop; reuse existing C/C++ libs; great performance; no serialization.
- **−** Manual memory management (leaks/UAF), synchronous (blocks isolate), C-type/ABI matching, per-platform native builds, more error-prone than channels.

Interview Questions

1. 🟢 **What is `dart:ffi` ?** — A foreign function interface letting Dart call C functions directly (in-process, synchronous), without platform channels.

2. 🟢 **FFI vs platform channels — when each?** — FFI for C/C++ libraries and perf-critical code (direct/sync); channels for platform/OS APIs (async/serialized).
3. 🟡 **How do you bind a C function?** — Load the lib (`DynamicLibrary.open`) and `lookupFunction<NativeSig, DartSig>('name')` with C-compatible types (or use `ffigen`).
4. 🟡 **How is memory managed?** — Manually: allocate with `malloc` / `calloc` , free with `free` (native memory is outside Dart's GC); mismanagement leaks or crashes.
5. 🟡 **Why offload heavy FFI to an isolate?** — FFI calls are synchronous on the calling isolate; long calls block it (UI jank) — run them via `Isolate.run` .
6. 🔴 **What does `ffigen` do?** — Generates Dart bindings (types/structs/functions) from C headers, automating non-trivial FFI binding.
7. 🔴 **What are FFI's main risks vs channels?** — Manual memory (leaks/use-after-free), synchronous blocking, and C-type/ABI mismatches (native crashes) — more error-prone.

Senior Engineer Tips

- Wrap FFI behind a Dart repository/service that owns memory (allocate/free internally) and exposes a safe, typed Dart API — callers never touch pointers.
- Use `ffigen` for real libraries; hand-writing bindings for many functions/structs is error-prone.
- Always offload heavy/blocking FFI to an isolate; treat native memory like a resource with a strict acquire/free lifecycle (`finally`).

Architect Perspective

FFI is the high-performance/native-library integration tier, distinct from channels (platform APIs). It enables reusing C/C++ ecosystems (crypto/media/ML) and speeding up hot paths, at the cost of manual memory/threading discipline. Encapsulated behind safe repositories with isolate offloading, it composes with the rest of native integration for a complete interop story (01_platform_channel_fundamentals.md, Module 21).

Summary

- `dart:ffi` calls C directly (sync, in-process, no serialization) — for C/C++ libraries and perf-critical code, not platform APIs.
- Load lib + `lookupFunction` (or `ffigen`); manage native memory manually; offload heavy calls to an isolate.
- Wrap behind a safe repository; bundle native libs per platform.

Revision Notes

- FFI = direct C calls (sync/in-process); vs channels (async/serialized, platform APIs).

- `DynamicLibrary.open` + `lookupFunction<NativeSig, DartSig>`; `ffigen` for bindings; C-compatible types.
- Manual memory (`malloc` / `free`, `toNativeUtf8`); offload heavy FFI to `Isolate.run`; bundle native lib.
- Wrap behind repository (owns memory, safe API); AOT-compatible.

Practice Questions

1. When choose FFI over a platform channel?
2. Who manages native memory, and how?
3. Why offload heavy FFI to an isolate?

Coding Questions

1. Bind and call a C `add(int,int)` via `lookupFunction`.
2. Pass/return a C string with proper `malloc` / `free`.
3. Wrap FFI in a repository (memory-safe) and run a heavy call via `Isolate.run`.

Mini Project — Module capstone

C library bridge (Flutter + C): Build a small C library (e.g., a fast hash/compress function), bind it via `ffigen` / `lookupFunction`, and wrap it in a `NativeCryptoRepository` that owns native memory (allocate/free), exposes a safe Dart API, and runs heavy calls on a background isolate. Contrast with a `MethodChannel` approach in notes. Acceptance: direct C call works; no memory leaks (freed in `finally`); heavy work off UI isolate; repository-wrapped; runs on device. (Completes the native-integration story: channels + Pigeon/plugins + FFI.)