27 · Native Android

Flutter Practice Notes

Contents

27 · Native Android

Introduction

This module covers the **Android platform side** of a Flutter app: the Android project structure and Gradle, writing Kotlin (channel handlers, activity/context, lifecycle), embedding native Android views (platform views), the manifest and permissions, and Android-specific integration (services/intents/receivers). It's the Android half of native integration (Module 26).

Why this module exists

Flutter apps still ship as Android apps: you configure Gradle, request permissions, write Kotlin for platform channels/plugins, embed native views, and integrate with Android services/intents. Doing this correctly is essential for real apps and for native integration.

Module map

#	File	Topic	Level
1	01_android_project_and_gradle.md	Project structure, Gradle, build config, flavors	🔵
2	02_kotlin_plugin_code.md	Kotlin channel handlers, `MainActivity`, context, lifecycle	🔴
3	03_platform_views_android.md	Embedding native Android views	🔴
4	04_permissions_and_manifest.md	Manifest, permissions (declared + runtime)	🔵
5	05_android_integration.md	Services, intents, broadcast receivers, background	🔴

> **Cross-references:** Platform channels/plugins: Module 26. Embedder/startup: 10 · embedder_and_startup. Device features (plugins): Module 29. Background: Module 33. Deployment/signing: Module 51. iOS counterpart: Module 28.

Prerequisites

26 Platform Channels, 10 Flutter Architecture. Basic Kotlin helps.

What you'll be able to do after this module

- Navigate the Android project + configure Gradle (SDK versions, flavors, signing basics).
- Write Kotlin channel handlers with correct activity/context/lifecycle usage.
- Embed native Android views via platform views.
- Declare and request Android permissions correctly (incl. runtime).

- Integrate with Android services, intents, and broadcast receivers.

Capstone

Android integration slice: A Kotlin `MethodChannel` / `EventChannel` reading a system value, a permission-gated feature (runtime permission), an embedded native Android view, and a foreground-service/intent integration — wired to the Flutter side via a repository.

Summary

The Android side is Gradle config + Kotlin (channels/activity/context/lifecycle) + manifest/permissions + platform views + service/intent integration. Master it to ship real Android apps and to implement native features that Dart/plugins can't.

Android Project Structure & Gradle

> A Flutter app's `android/` folder is a real Android/Gradle project: `build.gradle` files set SDK versions, dependencies, flavors, and signing; `MainActivity` hosts the Flutter engine; and `AndroidManifest.xml` declares the app — you configure these for versions, flavors, and native dependencies.

Introduction

`flutter create` generates a standard Android project under `android/`. This file maps its structure (Gradle files, `MainActivity`, manifest, resources), the key Gradle config (min/target/compile SDK, dependencies, flavors, signing), and how it connects to Flutter — the foundation for Kotlin code and platform features.

Why this concept exists

Flutter compiles to a native Android app; Android's build system (Gradle) governs SDK levels, dependencies, permissions, signing, and build variants. You must configure it for compatibility (SDK versions), native libraries, flavors (dev/prod), and release signing — Flutter tooling wraps but doesn't replace it.

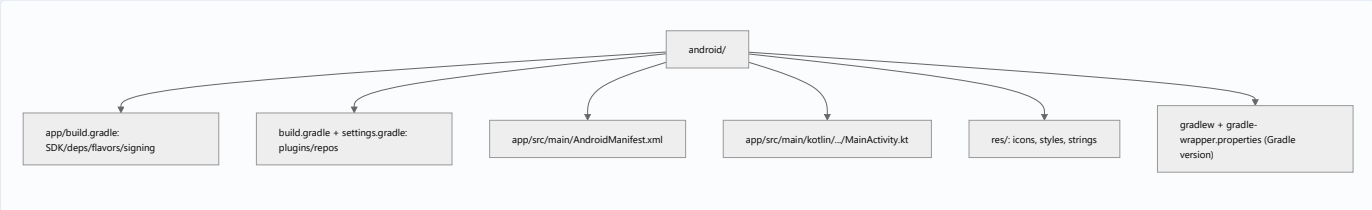
Real-world analogy

The `android/` project is the **chassis and engine mounts** your Flutter "body" bolts onto: Gradle is the **assembly-line spec** (which parts/versions, which trims/flavors, how it's badged/signed). You tune the spec for the market (SDK levels), options (flavors), and certification (signing).

Problem Statement

You must raise `minSdkVersion` for a plugin, add dev/prod flavors with different app ids, add a native dependency, and understand where `MainActivity`/manifest live. You'll map the project + edit Gradle.

Internal Working



- `android/app/build.gradle` (module): `compileSdk`, `defaultConfig` (`applicationId`, `minSdk`, `targetSdk`, `versionCode/Name`), `buildTypes` (debug/release + `minifyEnabled`/`shrinkResources` for R8), `signingConfigs`, `flavorDimensions`/`productFlavors`, and `dependencies { }` (native libs).
- `android/build.gradle` + `settings.gradle`: top-level plugin/repo config (Android Gradle Plugin, Kotlin, Google/Maven repos); newer Flutter uses the declarative plugins block.
- `gradle-wrapper.properties`: pins the **Gradle version** (compatibility matters with AGP/Kotlin).
- `MainActivity` (`kotlin/.../MainActivity.kt`): usually `class MainActivity : FlutterActivity()` — hosts the Flutter engine; where you register channels/plugins (02_kotlin_plugin_code.md).
- `AndroidManifest.xml`: app declaration — package, permissions, activities, intent filters, `application` config (04_permissions_and_manifest.md).
- `res/`: launcher icons, `styles.xml` (theme/splash), `strings.xml`, drawables.
- **Flavors**: define `productFlavors` (e.g., `dev`/`prod`) with distinct `applicationId`/config; build with `flutter build apk --flavor prod` + a matching entrypoint (`main_prod.dart`) — pairs with Firebase dev/prod (18 · firebase_setup_and_core).
- **SDK versions**: `minSdk` (lowest supported), `targetSdk` (tested-against), `compileSdk` (build API) — plugins often dictate a floor.

Memory Representation

Not applicable; build-time config. Release builds enable R8 (shrink/obfuscate) — pairs with `--obfuscate`/`--split-debug-info` (21 · startup_and_app_size).

Compiler Behavior

Gradle + AGP compile Kotlin/resources and package the app; Flutter's AOT Dart is embedded. SDK/AGP/Kotlin/Gradle versions must be compatible.

Runtime Behavior

`MainActivity` (a `FlutterActivity`) starts the engine and runs your Dart `main`. Flavors select config/app id at build.

Flutter Engine Behavior

`FlutterActivity` is the Android **embedder** hosting the engine + registering plugins (10 · embedder_and_startup).

Dart VM Behavior

Not applicable.

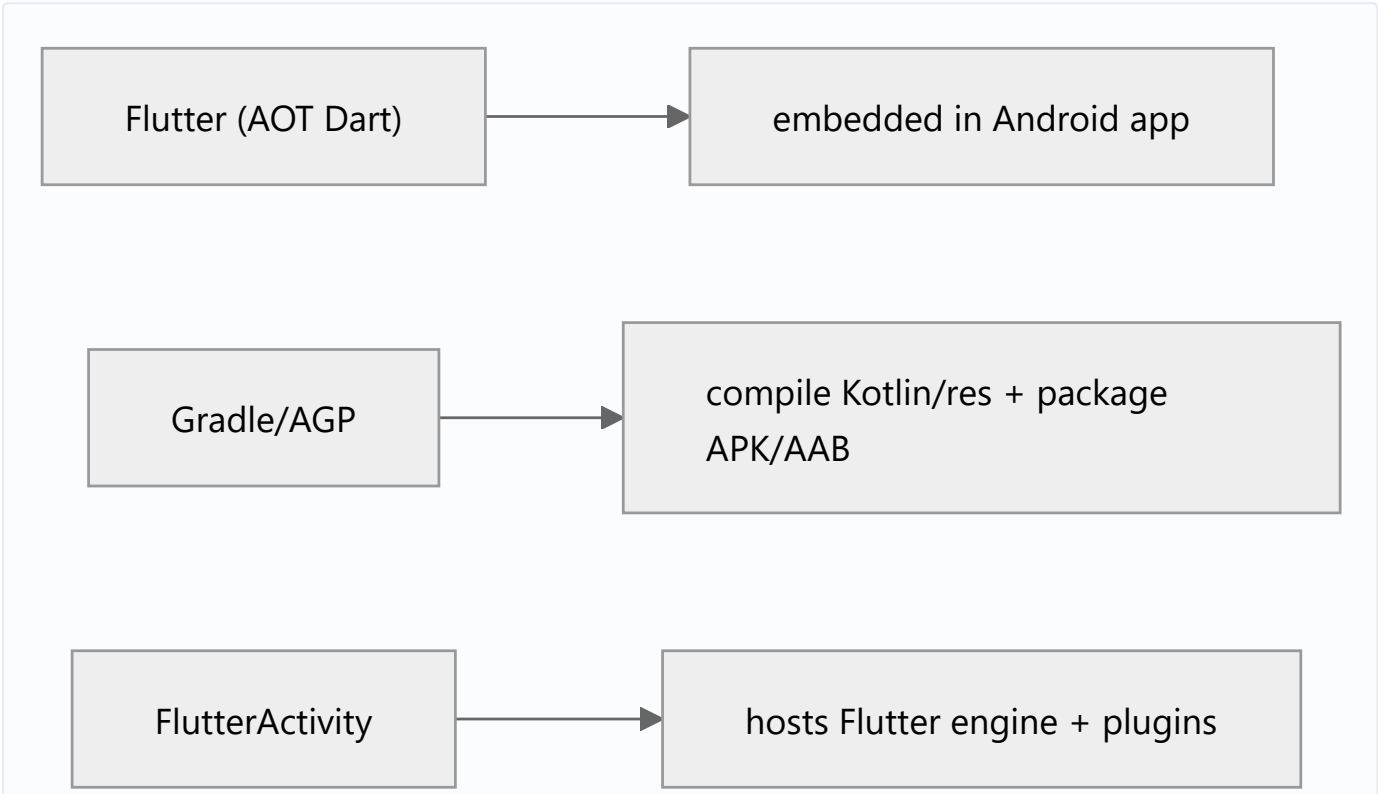
Examples

```
// android/app/build.gradle (Groovy) — key config
android {
    namespace "com.example.app"
    compileSdk 34

    defaultConfig {
        applicationId "com.example.app"
        minSdk 23          // plugins often require a floor (e.g., 21/23)
        targetSdk 34
        versionCode 1
        versionName "1.0.0"
    }

    flavorDimensions "env"
    productFlavors {
        dev  { dimension "env"; applicationIdSuffix ".dev"; versionNameSuffix "-dev" }
        prod { dimension "env" }
    }

    buildTypes {
        release {
            minifyEnabled true       // R8 shrink/obfuscate
            shrinkResources true
            signingConfig signingConfigs.release // configure real signing (Module 51)
        }
    }
    dependencies {
        implementation "androidx.core:core-ktx:1.12.0" // native dependency example
    }
}
```

```
// android/app/src/main/kotlin/.../MainActivity.kt
package com.example.app
import io.flutter.embedding.android.FlutterActivity
class MainActivity : FlutterActivity()  // hosts the engine; register channels in configureFlutterEngine
```

```
Build with a flavor + matching entrypoint:
  flutter build apk --flavor prod -t lib/main_prod.dart
  flutter build appbundle --release   # AAB for Play (Module 51)
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Wrong/low `minSdk` for a plugin	Build/runtime failure	Raise `minSdk` to the plugin's floor
Incompatible Gradle/AGP/Kotlin versions	Build errors	Align versions (wrapper + AGP + Kotlin)
Editing manifest for permissions ad hoc	Missing/duplicated	Declare properly (04_permissions_and_manifest.md)
No flavors for dev/prod	Config bleed	`productFlavors` + entrypoints
Debug signing for release	Rejected/unsigned	Configure release `signingConfig` (Module 51)
Ignoring R8 in release	Larger/less safe	`minifyEnabled` / `shrinkResources` + obfuscation

Best Practices

- Set `minSdk` / `targetSdk` / `compileSdk` deliberately (respect plugin floors; keep `targetSdk` current for Play).

- Keep **Gradle/AGP/Kotlin versions aligned**; use the wrapper.
- Define **dev/prod flavors** (distinct app ids/config) with matching Dart entrypoints; pair with Firebase envs.
- Configure **release signing + R8** (shrink/obfuscate); combine with `--obfuscate` / `--split-debug-info`.
- Register channels/plugins in `MainActivity.configureFlutterEngine` (02_kotlin_plugin_code.md); keep manifest/permissions correct.

Performance

Release R8 shrinking + AAB per-device delivery reduce size; correct SDK/flavor config avoids bloat. Build-time only (21 · startup_and_app_size).

Advantages / Disadvantages

- **+** Full Android build control (versions/deps/flavors/signing), standard Gradle project, native dependency support.
- **−** Gradle/version-compatibility complexity, platform-specific config, signing/flavor setup overhead.

Interview Questions

1. 🟢 **Where is the Android project in a Flutter app?** — In the `android/` folder — a standard Gradle project with `MainActivity`, manifest, and `build.gradle` files.
2. 🟢 **What does `app/build.gradle` configure?** — SDK versions (`min/target/compileSdk`), `applicationId`, versioning, `buildTypes`, `signingConfigs`, flavors, and native dependencies.
3. 🟡 **What is `MainActivity` and its role?** — A `FlutterActivity` that hosts the Flutter engine and where you register channels/plugins (`configureFlutterEngine`).
4. 🟡 **`minSdk` vs `targetSdk` vs `compileSdk`?** — Lowest supported OS, the OS version you've tested/targeted (Play requirements), and the API level you compile against.
5. 🟡 **How do you set up dev/prod builds?** — `productFlavors` (distinct app ids/config) + matching Dart entrypoints; build with `--flavor`.
6. 🔴 **Why do Gradle/AGP/Kotlin versions matter?** — They must be compatible; mismatches cause build failures — align the wrapper, AGP, and Kotlin versions.
7. 🔴 **What release config affects size/security?** — R8 (`minifyEnabled` / `shrinkResources`) + signing + `--obfuscate` / `--split-debug-info`, and AAB per-device delivery.

Senior Engineer Tips

- Pin and align Gradle/AGP/Kotlin versions; upgrade deliberately — version drift is the top Android build headache.
- Set up flavors + entrypoints early (dev/prod), matching Firebase/config envs, to avoid prod data bleed.
- Keep native changes minimal and documented; most native config lives here, so treat `build.gradle`/manifest as reviewed code.

Architect Perspective

The Android project/Gradle layer is where the Flutter app becomes a shippable Android artifact: SDK targeting, dependencies, flavors, signing, and R8. Configuring it correctly (versions, flavors, release optimization) is a delivery-and-compatibility concern that underpins native integration, deployment, and app size (Module 51, 21 · startup_and_app_size).

Summary

- `android/` is a Gradle project: `app/build.gradle` (SDKs/deps/flavors/signing), `MainActivity` (hosts engine), manifest, `res/`.
- Set SDK versions per plugin/Play, align Gradle/AGP/Kotlin, use dev/prod flavors + entrypoints, configure release signing + R8.
- Register channels/plugins in `MainActivity`; foundation for Kotlin/platform features.

Revision Notes

- `android/app/build.gradle`: `compileSdk`, `defaultConfig(applicationId/minSdk/targetSdk/version)`, `buildTypes(release: R8)`, `signingConfigs`, `productFlavors`, `dependencies`.
- `MainActivity : FlutterActivity` hosts engine + registers channels/plugins; manifest declares app/permissions.
- Align Gradle/AGP/Kotlin; flavors + entrypoints for dev/prod; release signing + obfuscation; AAB.
- Respect plugin `minSdk` floors.

Practice Questions

1. Where do you set `minSdk` and why might a plugin force it higher?
2. What's `MainActivity`'s role?
3. How do you configure dev/prod flavors?

Coding Questions

1. Edit `app/build.gradle` to add `dev` / `prod` flavors with distinct app ids.
2. Raise `minSdk` and add a native dependency.
3. Enable R8 + configure a release signing placeholder.

Mini Project

Android build config (Flutter/Gradle): Configure `android/app/build.gradle` with sensible SDK versions, dev/prod flavors (distinct app ids) + matching entrypoints, a native dependency, and release R8 + signing placeholder; confirm `MainActivity` hosts the engine. Build both flavors. Acceptance: flavors build; SDKs/deps correct; release optimized; documented; app runs on device.

Kotlin Plugin Code (Channels, Activity, Context, Lifecycle)

> On Android, native channel handlers are Kotlin: register them in `configureFlutterEngine`, use the right `Context` (application vs activity), get the `Activity` for UI/permission work via `ActivityAware` (in plugins), and offload heavy work off the main thread — returning results on it.

Introduction

This file covers writing the Android/Kotlin side of platform channels/plugins: where to register handlers, `Context` vs `Activity` (and why it matters), the plugin lifecycle (`FlutterPlugin` / `ActivityAware`), threading, and returning results/errors — the practical Kotlin patterns behind Module 26's channels.

Why this concept exists

Channels need a native implementation. On Android, that's Kotlin with Android-specific concerns: which `Context` to use (leaks/capabilities), getting the `Activity` (needed for permissions, `startActivityForResult`, UI), and lifecycle (activity attach/detach). Getting these wrong causes crashes, leaks, or missing capabilities.

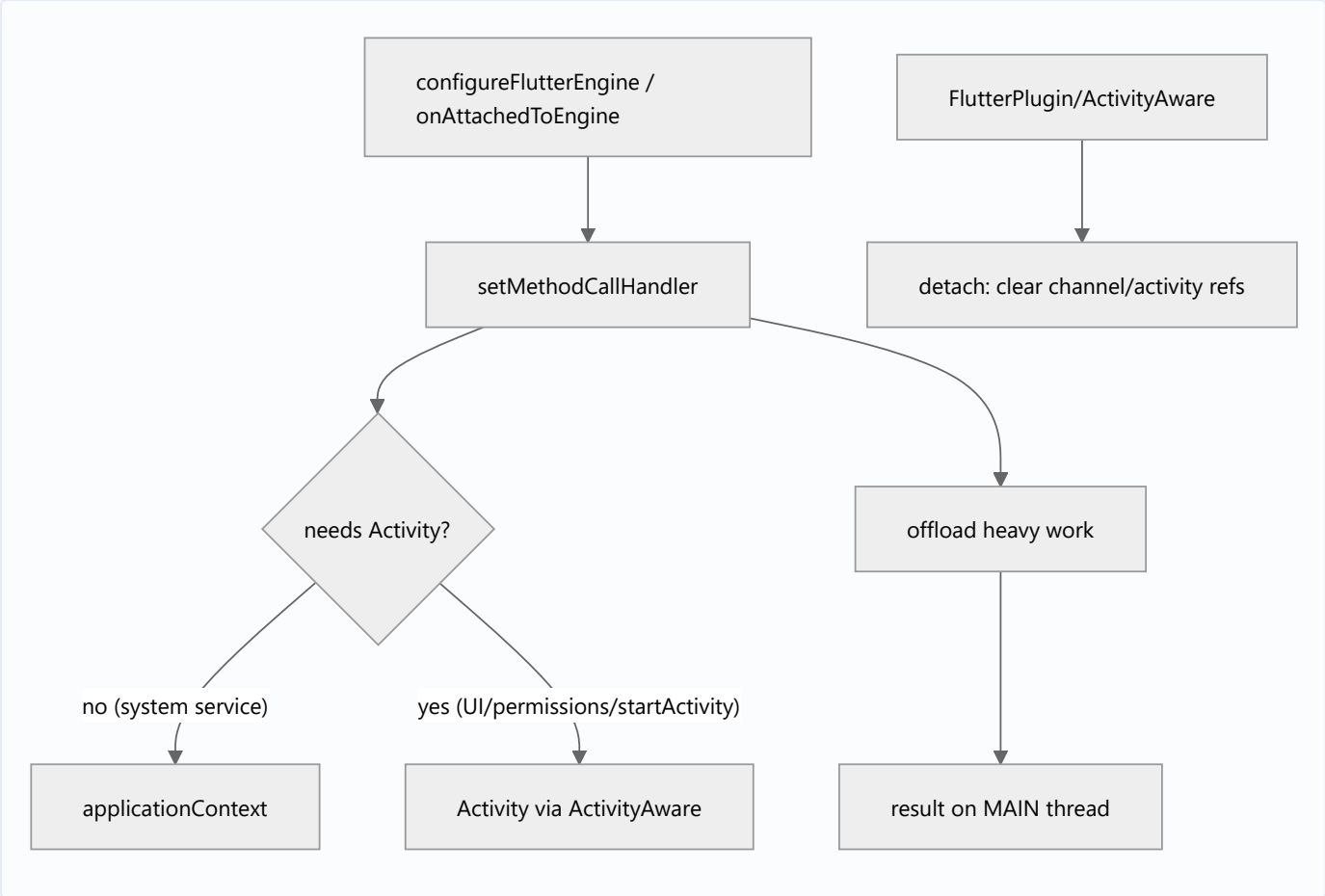
Real-world analogy

Writing Kotlin channel code is being the **local branch office** that fulfills requests from HQ (Dart). You need the right **credentials/keys** (`Context` / `Activity`) to access different resources, must **hand back through the correct window** (reply on main thread), and **check out when the branch closes** (lifecycle cleanup).

Problem Statement

Implement a Kotlin `MethodChannel` that reads a system service (application `Context`) and one that launches an activity/requests a permission (needs the `Activity`), with correct lifecycle and threading. You'll register handlers and use the right context/activity.

Internal Working



- **Registering handlers**:
  - **App-embedded**: override `configureFlutterEngine(flutterEngine)` in `MainActivity` and set up `MethodChannel(binaryMessenger, name).setMethodCallHandler { ... }`.
  - **Plugin**: implement `FlutterPlugin` (`onAttachedToEngine` / `onDetachedFromEngine`) to create/dispose the channel with the plugin's binary messenger.
- `Context` **vs** `Activity`:
  - Use `applicationContext` for system services (BatteryManager, connectivity), app resources — long-lived, **no leak** of an activity.
  - Use the `Activity` for UI (dialogs, `startActivity` / `startActivityForResult`), runtime **permissions**, window operations — required for those APIs.
  - Get the Activity in a **plugin** via `ActivityAware` (`onAttachedToActivity` / `onDetachedFromActivity...`); **don't hold** a stale Activity reference (leak) — clear on detach.
- **Threading**: handlers run on the **platform main thread**; do heavy work on a background thread/coroutine and call `result.success/error` **on the main thread** (post back) to avoid ANRs (26 · platform_channel_fundamentals).

- **Results/errors**: `result.success(value)`, `result.error(code, message, details)`, `result.notImplemented()`; supported types only (codec).
- **Lifecycle/cleanup**: null out channel/activity references and unregister receivers on detach to avoid leaks.

Memory Representation

Holding an `Activity` / `Context` reference beyond its lifecycle **leaks** it; use `applicationContext` where possible and clear activity refs on detach (08 · dispose_and_leaks analog for native).

Compiler Behavior

Kotlin compiled by Gradle/AGP; channel names/types are stringly-typed (Pigeon adds Kotlin type-safety — 26 · plugins_and_pigeon).

Runtime Behavior

Handlers dispatch by method; wrong context for an API throws/misbehaves; heavy main-thread work causes ANRs. Activity may be absent (background) — guard.

Flutter Engine Behavior

`MainActivity`/plugin attaches to the engine's binary messenger; the engine marshals channel messages (10 · engine_internals).

Dart VM Behavior

Not applicable.

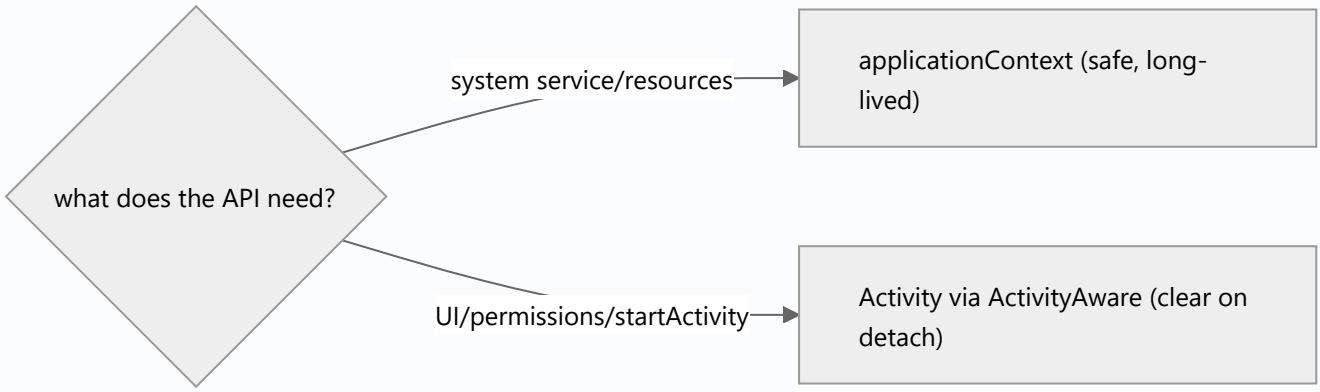
Examples

```kotlin
package com.example.app

import android.content.Context
import android.os.BatteryManager
import androidx.annotation.NonNull
import io.flutter.embedding.android.FlutterActivity
import io.flutter.embedding.engine.FlutterEngine
import io.flutter.plugin.common.MethodChannel
import kotlinx.coroutines.*


class MainActivity : FlutterActivity() {
```

```kotlin
    private val scope = CoroutineScope(Dispatchers.Main)

    override fun configureFlutterEngine(@NonNull flutterEngine: FlutterEngine) {
        super.configureFlutterEngine(flutterEngine)
        MethodChannel(flutterEngine.dartExecutor.binaryMessenger, "app/system")
            .setMethodCallHandler { call, result ->
                when (call.method) {
                    "getBatteryLevel" -> {
                        // system service -> applicationContext (no activity leak)
                        val bm = applicationContext.getSystemService(Context.BATTERY_SERVICE) as
     BatteryManager
                        result.success(bm.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPACITY))
                    }
                    "heavyCompute" -> {
                        // offload heavy work, reply on MAIN thread
                        scope.launch {
                            val value = withContext(Dispatchers.Default) { doHeavyWork() }
                            result.success(value) // back on main
                        }
                    }
                    "openSettings" -> {
                        // needs the Activity (this is FlutterActivity -> is an Activity)
                        startActivity(android.content.Intent(android.provider.Settings.ACTION_SETTINGS))
                        result.success(null)
                    }
                    else -> result.notImplemented()
                }
            }
    }

    private fun doHeavyWork(): Int = (1..1_000_000).sum()

    override fun onDestroy() { scope.cancel(); super.onDestroy() } // cleanup
}
```

```
// Plugin form (FlutterPlugin + ActivityAware) — sketch:
// class MyPlugin : FlutterPlugin, ActivityAware, MethodChannel.MethodCallHandler {
//   private var channel: MethodChannel? = null
//   private var activity: Activity? = null
//   override fun onAttachedToEngine(b: FlutterPlugin.FlutterPluginBinding) {
//     channel = MethodChannel(b.binaryMessenger, "my_plugin").also { it.setMethodCallHandler(this) }
//   }
//   override fun onDetachedFromEngine(b: ...) { channel?.setMethodCallHandler(null); channel = null }
//   override fun onAttachedToActivity(b: ActivityPluginBinding) { activity = b.activity }
//   override fun onDetachedFromActivity() { activity = null } // clear to avoid leak
// }
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Holding a stale `Activity` / `Context`	Memory leak	Use `applicationContext`; clear activity on detach
Using app context for Activity-only APIs	Crash (e.g., can't `startActivityForResult`)	Get Activity via `ActivityAware`
Heavy work on the main thread	ANR/jank	Offload (coroutine/thread); reply on main
Replying off the main thread	Errors	Post `result` to main
Not cleaning up on detach	Leaks (channel/activity/receivers)	Null refs + unregister in detach
Assuming an Activity always exists	Null in background	Guard/handle absence

Best Practices

- Register handlers in `configureFlutterEngine` (app) or `onAttachedToEngine` (plugin, `FlutterPlugin`).
- Use `applicationContext` for system services/resources; get the `Activity` via `ActivityAware` only when needed (UI/permissions/startActivity), and **clear it on detach**.
- **Offload heavy work** off the main thread (coroutines/`Dispatchers.Default`); **reply on the main thread**.
- Return via `result.success/error/notImplemented`; use **codec-supported types**; consider **Pigeon** for type-safety.
- **Clean up** on detach (null channel/activity, unregister receivers); handle a missing Activity (background).

Performance

Main-thread handlers must return quickly — offload heavy work to avoid ANRs; keep messages small (26 · platform_channel_fundamentals, Module 21).

Advantages / Disadvantages

- + Full access to Android APIs/SDKs from Kotlin, proper lifecycle integration, coroutine-friendly offloading.
- − Context/Activity/lifecycle pitfalls (leaks/crashes), main-thread discipline, stringly-typed channels (Pigeon helps), Kotlin/Android knowledge required.

Interview Questions

1. 🟢 **Where do you register Android channel handlers?** — In `MainActivity.configureFlutterEngine` (app) or a plugin's `onAttachedToEngine` (`FlutterPlugin`).
2. 🟢 `applicationContext` **vs** `Activity` **— when each?** — App context for system services/resources (long-lived, no leak); the Activity for UI, permissions, and `startActivity` / `ForResult`.
3. 🟡 **How does a plugin get the Activity?** — Implement `ActivityAware` (`onAttachedToActivity` / `onDetachedFromActivity`) and clear the reference on detach.
4. 🟡 **Why offload heavy work, and how do you reply?** — Main-thread handlers must return fast (ANR risk); offload to a coroutine/thread and call `result` back on the main thread.
5. 🟡 **How do you return errors/values?** — `result.success(v)`, `result.error(code,msg,details)`, `result.notImplemented()`; codec-supported types only.
6. 🔴 **What causes native memory leaks here?** — Holding a stale `Activity` / `Context` (or unregistered receivers) past its lifecycle; use app context and clear refs on detach.

7. 🔴 **How do you make Kotlin channel code type-safe?** — Use Pigeon-generated Kotlin interfaces instead of stringly-typed `setMethodCallHandler`.

Senior Engineer Tips

- Default to `applicationContext`; reach for the `Activity` only for APIs that require it, and always clear it on detach to avoid leaks.
- Wrap heavy native work in a coroutine (`Dispatchers.Default`) and marshal the result to main — a common ANR source otherwise.
- Prefer Pigeon for anything beyond a couple of methods; it removes Kotlin-side name/type mismatches.

Architect Perspective

The Kotlin channel layer is the Android implementation of your native features; correct context/activity/lifecycle/threading handling makes it leak-free, crash-free, and responsive. Packaged as a plugin with `FlutterPlugin` / `ActivityAware` (and Pigeon types), it's reusable and maintainable — the Android side of the repository-fronted native integration (Module 26).

Summary

- Register handlers in `configureFlutterEngine` / `onAttachedToEngine`; use `applicationContext` for services, `Activity` (via `ActivityAware`) for UI/permissions — clear it on detach.
- Offload heavy work; reply on the main thread; return via `result.*`; clean up on detach.
- Use codec types (or Pigeon); this is the Android side of channels/plugins.

Revision Notes

- Register: `configureFlutterEngine` (app) / `FlutterPlugin.onAttachedToEngine` (plugin).
- `applicationContext` (services, no leak) vs `Activity` (UI/permissions/startActivity, via `ActivityAware`, clear on detach).
- Offload heavy work (coroutine/`Dispatchers.Default`), reply on main; `result.success/error/notImplemented`.
- Clean up on detach (channel/activity/receivers); Pigeon for type-safety.

Practice Questions

1. When do you need the `Activity` vs `applicationContext`?
2. Why/how do you offload heavy native work and reply?
3. What causes leaks in Kotlin channel code?

Coding Questions

1. Write a Kotlin `MethodChannel` handler using `applicationContext` for a system service.
2. Add a method needing the `Activity` (via `ActivityAware`) and clear the ref on detach.
3. Offload heavy work with a coroutine and reply on the main thread.

Mini Project

Kotlin channel service (Flutter + Android): Implement a Kotlin `MethodChannel` with `getBatteryLevel` (applicationContext), `heavyCompute` (coroutine offload → main-thread reply), and `openSettings` (Activity), registered in `MainActivity` (or a `FlutterPlugin` + `ActivityAware`), with proper cleanup. Wrap the Dart side in a repository. Acceptance: correct context/activity usage; no leaks; heavy work off main; results/errors returned; runs on device.

Platform Views (Embedding Native Android Views)

> Platform views embed a **native Android** `View` (Google Maps, WebView, camera preview, ad banners) inside the Flutter widget tree via `AndroidView` / `PlatformViewLink` — powerful but comparatively expensive, so use them only when a Flutter widget can't do the job.

Introduction

Sometimes you must show a genuine native view (a `MapView`, `WebView`, camera `SurfaceView`, or a third-party SDK view) inside Flutter. Platform views composite a native `View` into the Flutter surface. This file covers how they work on Android (hybrid composition / virtual display), the widget API, a `PlatformViewFactory`, and the performance tradeoffs.

Why this concept exists

Flutter draws its own UI, so it can't natively render a `WebView` / `MapView`. Platform views bridge that: the native view renders and is composited with Flutter content. It's the mechanism behind plugins like `google_maps_flutter`, `webview_flutter`, and `camera`.

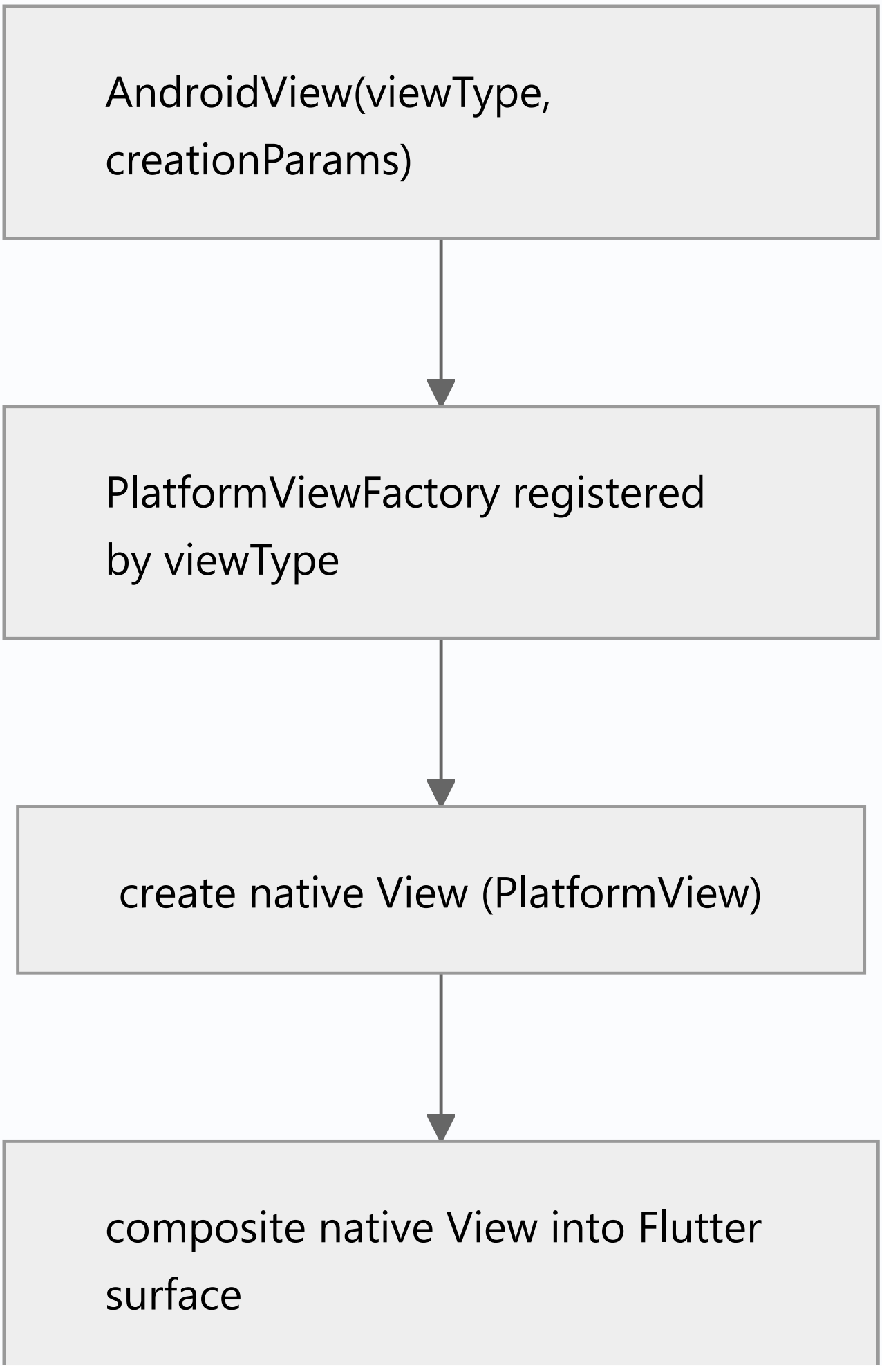
Real-world analogy

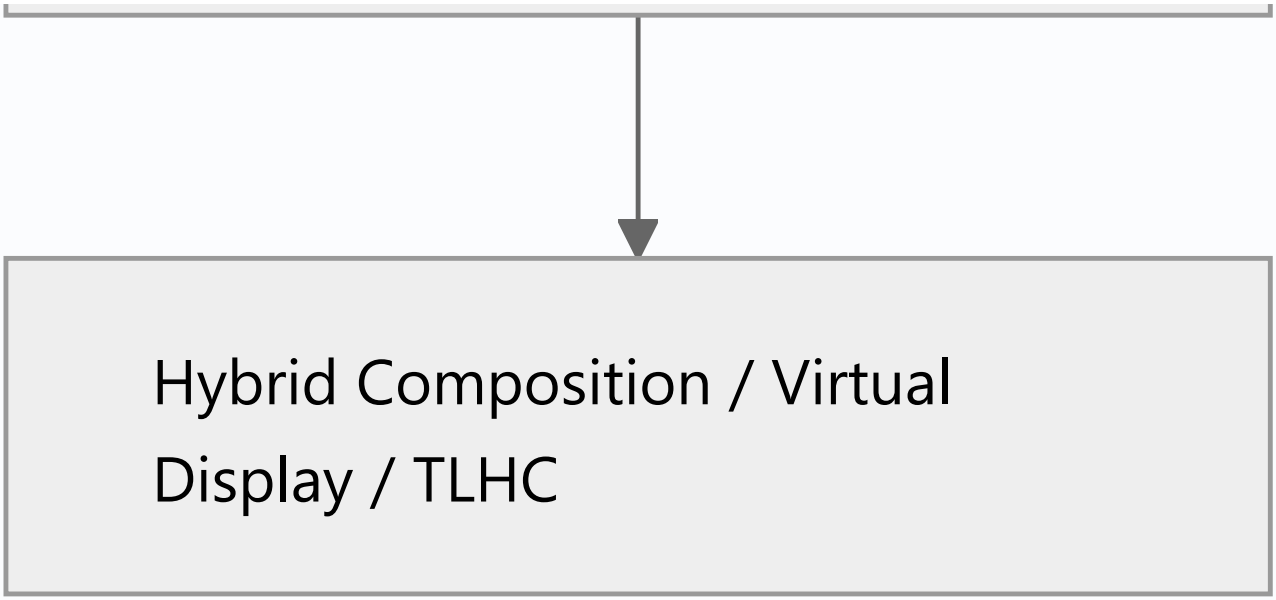
A platform view is a **window cut into your custom-painted wall** through which you see a **real object from the other room** (a native view). It works, but the window/compositing costs more than just painting on your wall — so you add windows only where you truly need to show the real thing.

Problem Statement

Embed a native Android `WebView` (or a custom native `View`) inside a Flutter screen, register its factory, pass creation params, and understand the perf implications. You'll register a `PlatformViewFactory` and use `AndroidView`.

Internal Working


AndroidView(viewType, creationParams)
PlatformViewFactory registered by viewType
create native View (PlatformView)
composite native View into Flutter surface



- **Dart side**: `AndroidView(viewType: 'my-native-view', creationParams: {...}, creationParamsCodec: StandardMessageCodec())` — or `PlatformViewLink` for finer control/gesture handling. `viewType` matches a registered factory.
- **Native side**: implement a `PlatformView` (wraps your `android.view.View`) and a `PlatformViewFactory` (creates it with creation params); register it in your plugin via `registerViewFactory(viewType, factory)`.
- **Composition modes** (Android): **Hybrid Composition** (composites native views with Flutter — better correctness/compat, some overhead), **Virtual Display** (renders the view to a texture — older), and newer **Texture Layer Hybrid Composition (TLHC)**. Flutter picks/uses these under the hood; you mostly choose the widget.
- **Cost**: platform views are **heavier** than Flutter widgets (extra compositing, potential texture copies, gesture/keyboard/z-order complexity). Use **sparingly** and only when necessary.
- **Gestures/keyboard**: input must be routed correctly to the native view (the platform-view system handles most; complex cases need `PlatformViewLink`/gesture recognizers).

Memory Representation

A native `View` + its texture/surface live alongside Flutter's layer tree; compositing/texture buffers cost memory. Dispose the native view when removed (`PlatformView.dispose`).

Compiler Behavior

Not applicable; factory/view are Kotlin registered at plugin/engine setup.

Runtime Behavior

The native view renders and composites each frame with Flutter content; heavy native views (maps/webview) add raster/compositing cost. Removing the widget disposes the native view.

Flutter Engine Behavior

The engine composites the native view into its surface via the platform-view system (hybrid/virtual display/TLHC) — extra work vs pure Flutter layers (09 · compositing).

Dart VM Behavior

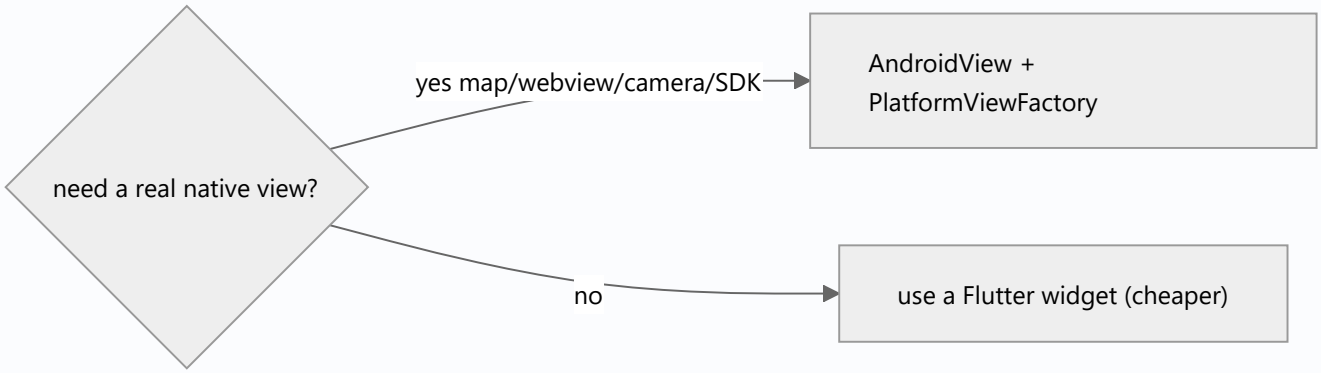
Not applicable.

Examples

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

// Embed a native Android view by its registered viewType
class NativeWebView extends StatelessWidget {
  final String url;
  const NativeWebView({super.key, required this.url});
  @override
  Widget build(BuildContext context) {
    return AndroidView(
      viewType: 'app/native-webview',                // matches the registered factory
      creationParams: {'url': url},
      creationParamsCodec: const StandardMessageCodec(),
      // gestureRecognizers: {...} // if the native view needs specific gestures
    );
  }
}
```

```kotlin
// Android: PlatformView + Factory, registered by viewType
class NativeWebViewFactory : PlatformViewFactory(StandardMessageCodec.INSTANCE) {
  override fun create(context: Context, viewId: Int, args: Any?): PlatformView {
    val params = args as? Map<*, *>
    return NativeWebViewImpl(context, params?.get("url") as? String ?: "")
  }
}
class NativeWebViewImpl(context: Context, url: String) : PlatformView {
  private val webView = android.webkit.WebView(context).apply { loadUrl(url) }
  override fun getView(): android.view.View = webView
  override fun dispose() { webView.destroy() } // release native resource
}
// Register in FlutterPlugin.onAttachedToEngine:
// binding.platformViewRegistry.registerViewFactory("app/native-webview", NativeWebViewFactory())
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Using platform views for things Flutter can do	Unnecessary cost/complexity	Use Flutter widgets; reserve for real native views
Not disposing the native view	Leak (WebView/Map/camera)	Implement `PlatformView.dispose`
Many/large platform views	Compositing/memory cost, jank	Minimize count/size; avoid in scrolling lists
Gesture/keyboard conflicts	Input not reaching native view	Route gestures (`PlatformViewLink`/recognizers)
Wrong `viewType` mismatch	View not created	Match Dart `viewType` to registered factory
Passing params without a codec	Params not received	Use `creationParamsCodec` on both sides

Best Practices

- Use platform views **only when necessary** (maps/webview/camera/native SDK views); prefer Flutter widgets otherwise (cheaper, simpler).
- Implement `PlatformView.dispose` to **release native resources** (destroy WebView, remove observers).
- **Minimize** platform-view count/size; avoid them inside heavy scrolling lists; be mindful of compositing/raster cost.
- Match `viewType` and use a **codec** for creation params; route gestures/keyboard correctly.
- Prefer maintained **plugins** (`webview_flutter`, `google_maps_flutter`, `camera`) over hand-rolling platform views.

Performance

Platform views add compositing/texture overhead vs Flutter layers — heavier and can jank if many/large or in lists. Use sparingly; profile raster/compositing (21 · jank_and_raster).

Advantages / Disadvantages

- **+** Embed real native views (maps/webview/camera/SDKs) inside Flutter; access native rendering.
- **−** More expensive than Flutter widgets (compositing/memory), gesture/keyboard/z-order complexity, per-platform native code, disposal responsibility.

Interview Questions

1. 🟢 **What are platform views for?** — Embedding a native platform `View` (map/webview/camera/SDK) inside the Flutter widget tree.
2. 🟢 **How do you embed one on Android?** — `AndroidView(viewType, creationParams, codec)` on the Dart side, backed by a registered `PlatformViewFactory` creating a `PlatformView`.
3. 🟡 **Why are platform views more expensive than Flutter widgets?** — They add native-view compositing/texture overhead (hybrid composition/virtual display/TLHC) beyond Flutter's own layers.
4. 🟡 **How do you avoid leaking a native view?** — Implement `PlatformView.dispose` to release resources (e.g., `WebView.destroy`).
5. 🟡 **When should you NOT use a platform view?** — When a Flutter widget can do it — reserve platform views for genuinely native views.
6. 🔴 **What are the Android composition modes?** — Hybrid Composition, Virtual Display, and Texture Layer Hybrid Composition (TLHC) — different tradeoffs in correctness/perf; Flutter selects/uses them.
7. 🔴 **What complexities do platform views add?** — Gesture/keyboard routing, z-ordering with Flutter content, disposal, and compositing cost — especially in lists.

Senior Engineer Tips

- Prefer existing plugins (webview/maps/camera) — they handle the platform-view + gesture/lifecycle complexity for you.
- Keep platform views small, few, and out of fast scroll views; profile compositing/raster when you add them.
- Always dispose native views; leaked WebViews/MapViews are memory-heavy and common.

Architect Perspective

Platform views are the "embed a real native view" escape hatch — essential for maps/webview/camera/SDKs but costlier than Flutter's own rendering. Architecturally, minimize and encapsulate them (behind widgets/plugins), dispose native resources, and prefer Flutter widgets — reserving platform views for what truly requires native rendering (09 · compositing, Module 29).

Summary

- Platform views embed native Android `View`s via `AndroidView` + a `PlatformViewFactory` / `PlatformView`.

- Heavier than Flutter widgets (compositing/memory) — use sparingly, dispose native resources, mind gestures/lists.
- Prefer Flutter widgets/maintained plugins; reserve for genuinely native views (map/webview/camera/SDK).

Revision Notes

- Dart: `AndroidView(viewType, creationParams, creationParamsCodec)`; native: `PlatformView` + `PlatformViewFactory` registered by `viewType`.
- Composition: Hybrid/Virtual Display/TLHC (Flutter-managed); heavier than Flutter layers.
- Dispose native view (`PlatformView.dispose`); minimize count/size; route gestures/keyboard.
- Prefer plugins (webview/maps/camera); use only for real native views.

Practice Questions

1. When is a platform view justified vs a Flutter widget?
2. Why are platform views more expensive, and where do you avoid them?
3. How do you prevent leaking an embedded native view?

Coding Questions

1. Embed a native `WebView` via `AndroidView` + a `PlatformViewFactory`.
2. Pass creation params (url) with a codec and read them natively.
3. Implement `dispose` to release the native view.

Mini Project

Native view embed (Flutter + Android): Embed a native Android view (e.g., a simple custom `View` or `WebView`) via `AndroidView` + a registered `PlatformViewFactory` / `PlatformView`, pass creation params with a codec, and dispose it correctly. Keep it single/small and out of a list. Acceptance: native view renders inside Flutter; params passed; disposed (no leak); minimal/encapsulated; runs on device.

Permissions & the Manifest

> Android permissions have two parts: **declare** them in `AndroidManifest.xml`, and (for dangerous permissions) **request them at runtime** — plus handle denial/"don't ask again" gracefully; use `permission_handler` to request/check from Dart, and keep the manifest minimal.

Introduction

`AndroidManifest.xml` declares the app (package, activities, intent filters, permissions); dangerous permissions (camera, location, storage, contacts) also require a **runtime request** since Android 6 (API 23). This file covers manifest structure, permission types, the runtime-permission flow (via `permission_handler`), and best practices.

Why this concept exists

Users must consent to sensitive capabilities. Android splits permissions into **normal** (granted at install by declaration) and **dangerous** (granted at runtime by the user). Apps must declare, request at the right time, handle denial/rationale, and degrade gracefully — a correctness and Play-policy requirement.

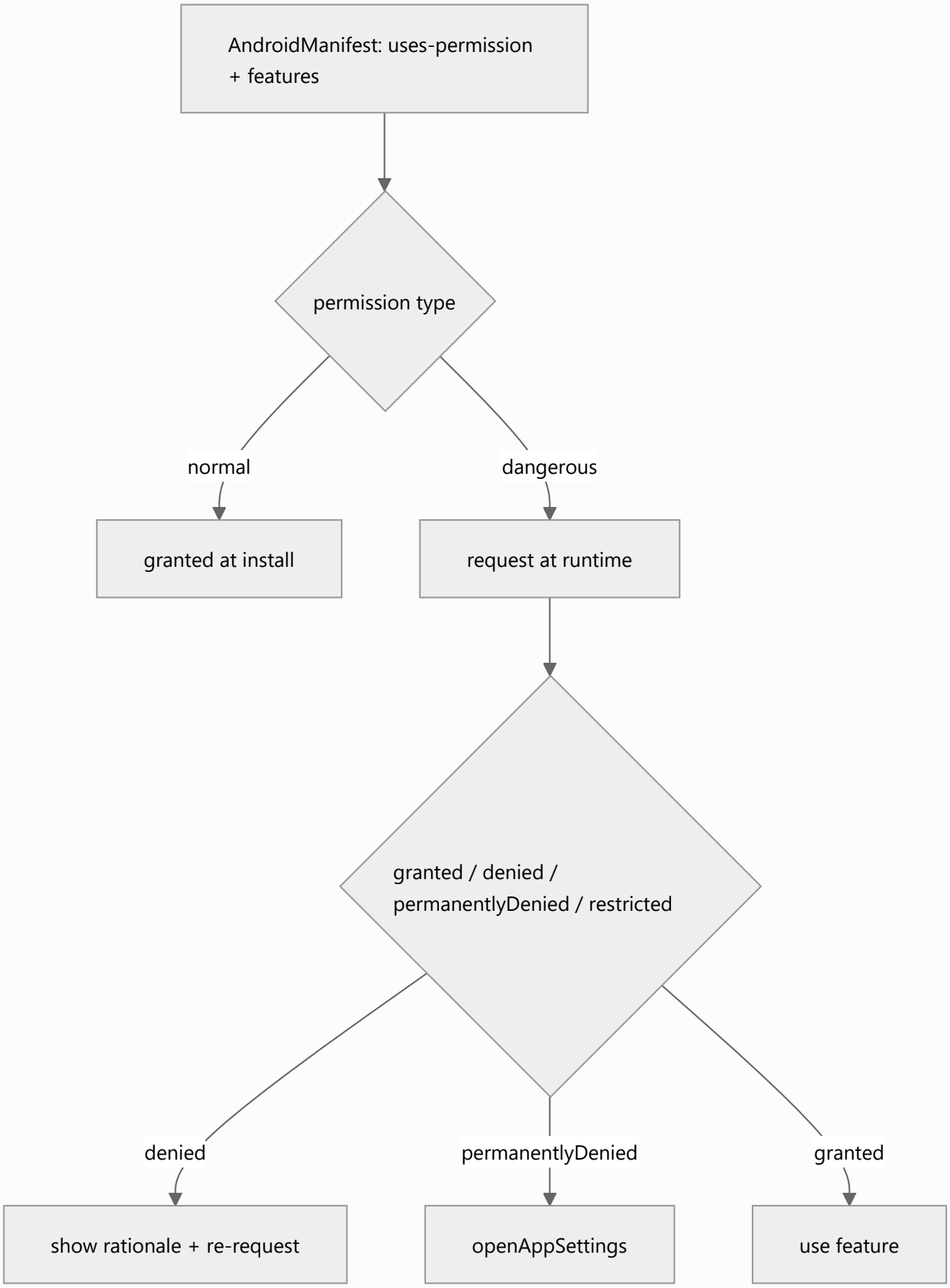
Real-world analogy

The manifest is your **building permit application** listing what you intend to do; runtime permissions are **asking the resident for the key** each time you need to enter a private room (camera/location). They can say no, or "stop asking" — and you must still function without that room.

Problem Statement

Add camera + location to your app: declare them, request at runtime when the feature is used, handle granted/denied/permanently-denied, and open app settings if needed. You'll edit the manifest and use `permission_handler` behind a repository.

Internal Working



- **Manifest** (`android/app/src/main/AndroidManifest.xml`): declare `<uses-permission android:name="android.permission.CAMERA"/>` , `ACCESS_FINE_LOCATION` , etc.; optionally `<uses-feature>` (e.g., camera) — and `<application>` / `<activity>` config, intent filters (deep links — 13 · deep_linking).
- **Permission types**: **normal** (INTERNET, network state) granted at install by declaration; **dangerous** (camera, location, mic, contacts, storage) require **runtime** consent; **special** (e.g., overlay, all-files-access) need settings-based grants.
- **Runtime flow** (via `permission_handler`): `Permission.camera.status` → if not granted, `.request()` ; handle `granted` / `denied` / `permanentlyDenied` / `restricted` ; show a **rationale** on denial; `openAppSettings()` when permanently denied.
- **Timing**: request **in context**, when the user triggers the feature (not all upfront) — better grant rates + Play guidance.
- **Scoped storage / newer APIs**: media/storage permissions changed (Android 13+ granular media perms; scoped storage) — declare the right ones per `targetSdk` .
- **Wrap in a repository/service**: expose `ensureCameraPermission()` returning a result; keep permission logic centralized and testable.

Memory Representation

Not applicable; permission state is OS-managed. `permission_handler` queries/requests via platform APIs.

Compiler Behavior

Manifest merged at build (library manifests merge in); missing declarations cause SecurityException at runtime, not compile time.

Runtime Behavior

`.request()` shows the OS dialog (dangerous perms); denial/"don't ask again" changes status to `denied` / `permanentlyDenied` . Using a capability without permission throws `SecurityException` — always check/request first.

Flutter Engine Behavior

Permission dialogs/results go through the Activity (needs `ActivityAware` in plugins — 02_kotlin_plugin_code.md); `permission_handler` manages this.

Dart VM Behavior

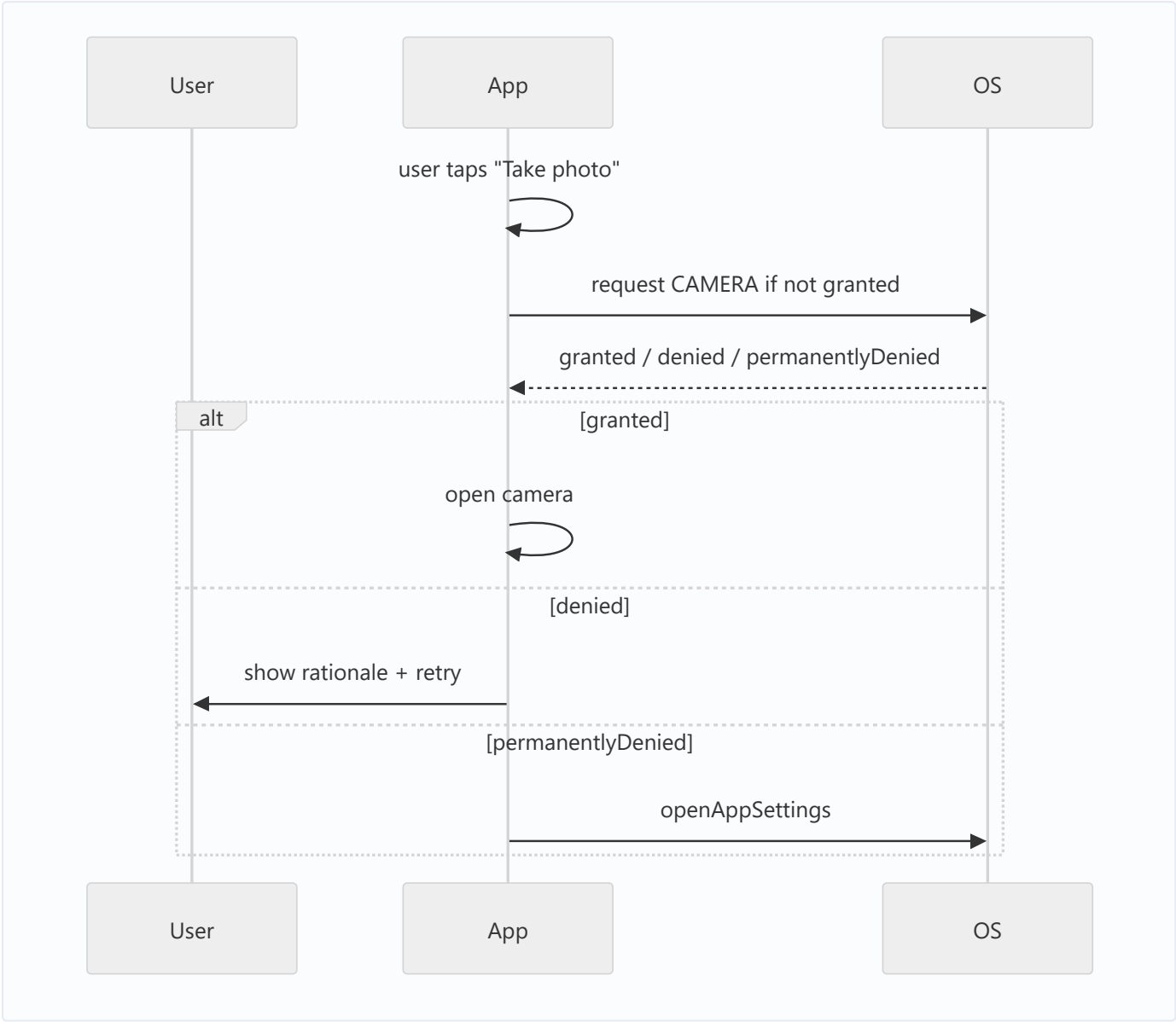
Not applicable.

Examples

```xml
<!-- android/app/src/main/AndroidManifest.xml -->
<manifest ...>
  <uses-permission android:name="android.permission.INTERNET"/>       <!-- normal -->
  <uses-permission android:name="android.permission.CAMERA"/>         <!-- dangerous -->
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
  <uses-feature android:name="android.hardware.camera" android:required="false"/>
  <application ...>
    <activity android:name=".MainActivity" ...> ... </activity>
  </application>
</manifest>
```

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

// Repository centralizing permission logic (request in context, handle all states)
class PermissionService {
  Future<bool> ensureCamera() async {
    var status = await Permission.camera.status;
    if (status.isGranted) return true;
    if (status.isPermanentlyDenied) {
      await openAppSettings();          // user must enable in settings
      return false;
    }
    status = await Permission.camera.request(); // shows OS dialog
    // (show a rationale before requesting again on plain denial)
    return status.isGranted;
  }
}
// Usage: if (await permissions.ensureCamera()) { openCamera(); } else { showWhyWeNeedIt(); }
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Using a capability without runtime request	`SecurityException` crash	Check + request dangerous perms first
Declaring but not requesting (dangerous)	Install-grant only covers normal	Request at runtime
Requesting all permissions upfront	Low grant rates, Play issues	Request in context, when needed
Not handling `permanentlyDenied`	User stuck	`openAppSettings()` + explain
Over-declaring permissions	Privacy/Play scrutiny, user distrust	Declare only what you use
Wrong storage/media perms for targetSdk	Broken media access	Use granular media perms (Android 13+)/scoped storage

Best Practices

- **Declare only needed** permissions in the manifest; use `<uses-feature required="false">` where optional.
- **Request dangerous permissions at runtime, in context** (on feature use), via `permission_handler`; handle **all states** (granted/denied/permanentlyDenied/restricted).
- Show a **rationale** on denial; route to `openAppSettings()` when permanently denied; **degrade gracefully** without the permission.
- Use the correct **media/storage** permissions for your `targetSdk` (Android 13+ granular; scoped storage).
- **Centralize** permission logic in a repository/service (testable, consistent); mirror on iOS (Info.plist usage strings — Module 28).

Performance

Negligible; permission checks/requests are occasional. Requesting in context improves grant rates (a UX/business metric more than perf).

Advantages / Disadvantages

- **+** User consent/privacy, Play compliance, capability access with graceful degradation.
- **−** Multi-state flow + settings handling, changing storage/media rules per SDK, must degrade without permission.

Interview Questions

1. 🟢 **What are the two parts of Android permissions?** — Declare in `AndroidManifest.xml` and (for dangerous permissions) request at runtime with user consent.
2. 🟢 **Normal vs dangerous permissions?** — Normal (e.g., INTERNET) are granted at install by declaration; dangerous (camera/location/mic/contacts/storage) require a runtime request.
3. 🟡 **How do you request permissions from Dart?** — Via `permission_handler`: check `Permission.x.status`, `.request()`, and handle `granted/denied/permanentlyDenied/restricted`.
4. 🟡 **What do you do on `permanentlyDenied`?** — Explain and call `openAppSettings()`; you can't re-prompt directly.
5. 🟡 **When should you request a permission?** — In context, when the user triggers the feature — not all upfront (better grant rates + Play guidance).
6. 🔴 **What changed for storage/media permissions recently?** — Android 13+ introduced granular media permissions and scoped storage; declare the right ones per `targetSdk`.
7. 🔴 **What happens if you use a dangerous capability without permission?** — A `SecurityException` at runtime — always check/request first and degrade gracefully.

Senior Engineer Tips

- Centralize permission flows in a `PermissionService` returning clear results; screens call `ensureX()` and handle true/false — consistent UX and testable.
- Request in context with a pre-prompt rationale; handle "don't ask again" by guiding to settings — and always have a no-permission fallback.
- Keep the manifest minimal; over-declaring hurts trust and Play review; align media/storage perms with `targetSdk`.

Architect Perspective

Permissions are a privacy, UX, and compliance concern. A centralized, in-context, all-states-handled permission strategy (declare minimal, request when needed, degrade gracefully, settings fallback) — mirrored across Android/iOS — is essential for real apps and store approval, and integrates with device features and background work (Module 29, Module 33).

Summary

- Declare permissions in the manifest; request **dangerous** ones at runtime (in context) via `permission_handler`; handle all states + settings fallback.
- Declare only what you use; use correct media/storage perms per `targetSdk`; degrade gracefully.
- Centralize in a service; mirror iOS usage strings.

Revision Notes

- Manifest: `<uses-permission>` / `<uses-feature>`; normal (install) vs dangerous (runtime) vs special (settings).
- `permission_handler`: `status` / `request()`; handle granted/denied/permanentlyDenied(→ `openAppSettings`)/restricted; show rationale.
- Request in context; declare minimal; Android 13+ granular media/scoped storage per targetSdk.
- Centralize in `PermissionService`; degrade gracefully; mirror iOS Info.plist.

Practice Questions

1. Which permissions need a runtime request, and how do you handle denial?
2. What do you do when a permission is permanently denied?
3. Why request permissions in context rather than upfront?

Coding Questions

1. Declare camera + location in the manifest and request camera at runtime via `permission_handler`.
2. Handle `permanentlyDenied` by opening app settings.
3. Build a `PermissionService.ensureCamera()` with rationale + fallback.

Mini Project

Permission-gated camera (Flutter + Android): Declare camera/location in the manifest, build a `PermissionService` that requests camera in-context, handles granted/denied/permanentlyDenied (rationale + `openAppSettings`), and gates a camera feature with graceful fallback. Acceptance: minimal declarations; runtime request in context; all states handled; degrades without permission; centralized/testable; runs on device.

Android Integration (Services, Intents, Broadcast Receivers)

> Deep Android integration means using native OS constructs: **Intents** (launch activities/share/deep links), **Services** (background work, incl. foreground services with notifications), and **BroadcastReceivers** (react to system/app events) — surfaced to Flutter via channels/plugins, respecting modern background-execution limits.

Introduction

Beyond reading values, apps integrate with Android's app model: starting other apps/activities (Intents), running background work (Services/WorkManager), and reacting to system events (BroadcastReceivers). This file surveys these, how they reach Flutter, and the strict modern background limits.

Why this concept exists

Android features (share, open URLs, foreground services for ongoing tasks, reacting to connectivity/boot/battery events) live in the native app model. Flutter integrates via native Kotlin bridged with channels/plugins. Modern Android heavily restricts background work, so integration must respect those rules.

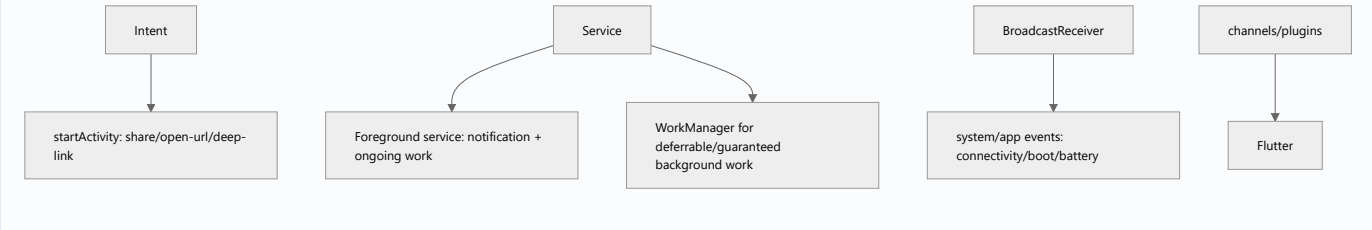
Real-world analogy

Intents are **sending a courier with a labeled parcel** ("open this URL", "share this text") to whichever app handles it. Services are **hiring staff to keep working** after you leave the office (with a visible badge for foreground services). BroadcastReceivers are **subscribing to building-wide announcements** (power, network) and reacting.

Problem Statement

Your app must open a URL/share text (Intent), run an ongoing location task (foreground service with a notification), and react to connectivity changes (receiver). You'll use Intents, a foreground service, and a receiver — bridged to Flutter and within background limits.

Internal Working



- **Intents**: launch activities/apps: `startActivity(Intent(ACTION_VIEW, uri))` (open URL), `ACTION_SEND` (share), or explicit intents to your activities. Incoming intents (deep links) route to Flutter (13 · deep_linking). Often handled by plugins (`url_launcher` , `share_plus`).
- **Services**:
  - **Foreground service**: for user-visible ongoing work (music, navigation) — **requires a persistent notification** and (Android 14+) a **service type** + permission; won't be killed like background work.
  - **Background limits**: since Android 8+ background execution is heavily restricted — use **WorkManager** for deferrable/guaranteed background tasks, not raw background services (Module 33).
- **BroadcastReceivers**: register (statically in manifest — limited since Android 8 — or dynamically at runtime) to react to system/app events (connectivity, boot, battery, custom). Stream events to Flutter via `EventChannel` (26 · event_channel); many are covered by plugins (`connectivity_plus`).
- **Bridging to Flutter**: expose these via `MethodChannel` (fire an intent), `EventChannel` (stream receiver events), or plugins; background isolates need setup (02 · isolates).
- **Prefer plugins**: `url_launcher` / `share_plus` / `connectivity_plus` / `workmanager` / `flutter_local_notifications` cover most cases — hand-roll only for custom native behavior.

Memory Representation

Services/receivers hold native resources — stop services and unregister receivers when done to avoid leaks/battery drain (08 · dispose_and_leaks analog; 26 · event_channel).

Compiler Behavior

Manifest declarations (services/receivers/permissions) merged at build; missing declarations/permissions fail at runtime.

Runtime Behavior

Intents launch async; foreground services run with a notification (killed if misconfigured on modern Android); receivers fire on events; background limits may defer/deny raw background work (use WorkManager).

Flutter Engine Behavior

Background work may run in a **background Flutter isolate** (e.g., WorkManager/headless) needing engine/messenger setup; intents/receivers bridge via the embedder (10 · embedder_and_startup).

Dart VM Behavior

Background isolates for background tasks need `RootIsolateToken`/background messenger to use channels (02 · isolates).

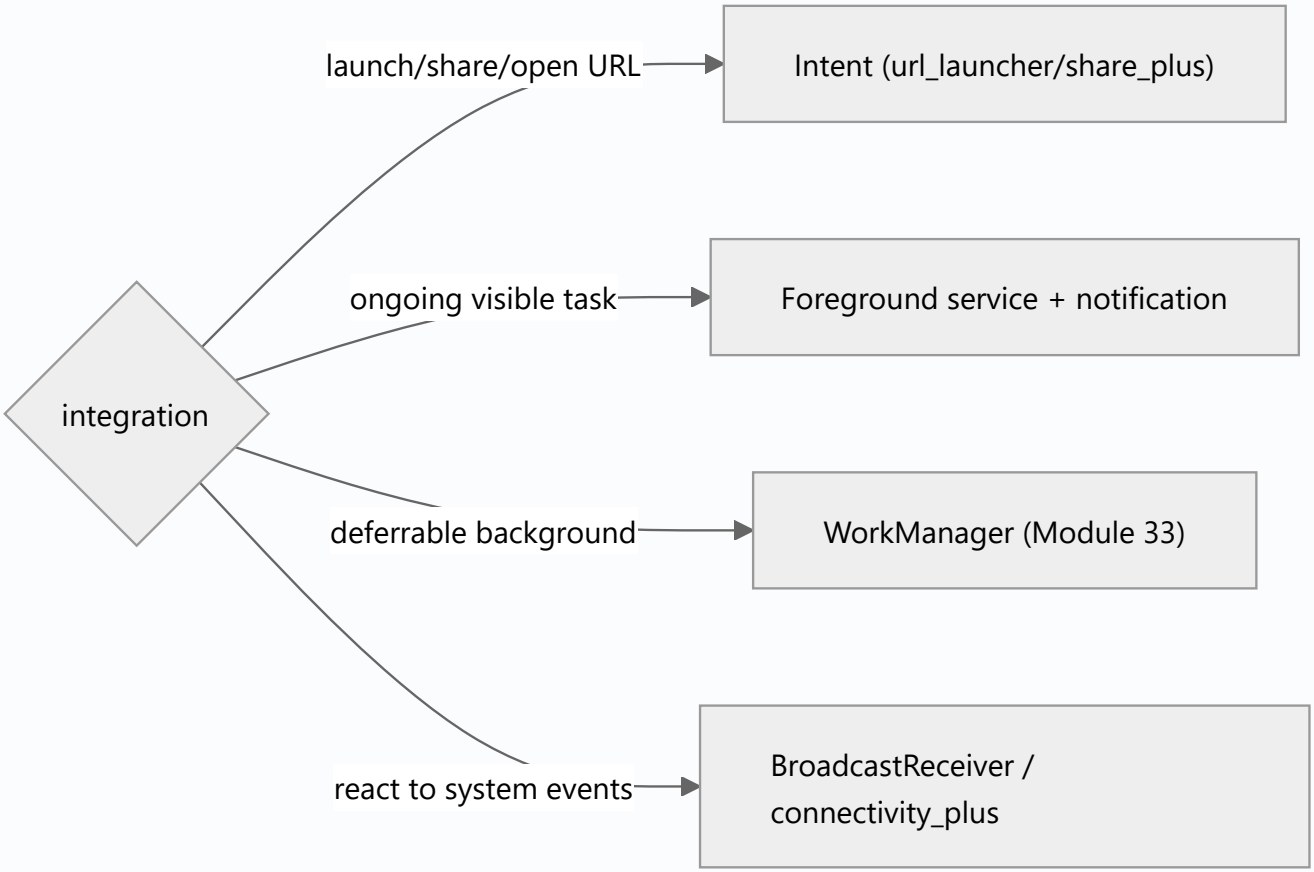
Examples

```
// Prefer plugins for common integrations:
import 'package:url_launcher/url_launcher.dart';   // open URL / dial / email (Intents)
import 'package:share_plus/share_plus.dart';        // share sheet (ACTION_SEND)
import 'package:connectivity_plus/connectivity_plus.dart'; // connectivity (receiver)

Future<void> openAndShare(String url) async {
  await launchUrl(Uri.parse(url));                   // ACTION_VIEW intent
  await Share.share('Check this out: $url');          // share intent
}

Stream<List<ConnectivityResult>> connectivity() =>
    Connectivity().onConnectivityChanged;             // wraps a BroadcastReceiver
```

```
// Custom Intent from a MethodChannel (when no plugin fits):
"openSettings" -> {
  startActivity(Intent(Settings.ACTION_APPLICATION_DETAILS_SETTINGS)
    .setData(Uri.fromParts("package", packageName, null)))
  result.success(null)
}

// Foreground service (sketch): must post a notification + declare in manifest + type/permission
// class TrackingService : Service() {
//   override fun onStartCommand(i: Intent?, f: Int, id: Int): Int {
//     startForeground(NOTIF_ID, buildNotification()) // required
//     // do ongoing work; stopSelf() when done
//     return START_STICKY
//   }
// }
```

```
<!-- Manifest declarations for a foreground service (Android 14+ needs a type) -->
<uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE_LOCATION"/>
<service android:name=".TrackingService" android:foregroundServiceType="location"/>
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Raw background service for ongoing bg work	Killed by background limits (Android 8+)	Foreground service (visible) or WorkManager
Foreground service without notification/type	Crash/kill on modern Android	Post notification + declare type/permission
Static receivers for restricted implicit broadcasts	Ignored since Android 8	Register dynamically / use JobScheduler/WorkManager
Not stopping services / unregistering receivers	Leaks/battery drain	`stopSelf` /unregister when done
Hand-rolling instead of plugins	Reinvent + bugs	Use `url_launcher` / `share_plus` / `connectivity_plus` / `workmanager`
Channels from bg isolate without setup	Not wired	`RootIsolateToken` + background messenger

Best Practices

- **Prefer maintained plugins** (`url_launcher`, `share_plus`, `connectivity_plus`, `workmanager`, `flutter_local_notifications`); hand-roll native only for custom behavior.
- For **ongoing user-visible work** use a **foreground service** (notification + service type/permission on Android 14+); for **deferrable/guaranteed** background use **WorkManager** — never rely on raw background services (Module 33).
- Register **receivers** appropriately (dynamic for most; respect Android 8+ implicit-broadcast limits); stream to Flutter via `EventChannel`; **unregister/stop** when done.
- **Bridge via channels/plugins**; set up background-isolate messengers where needed.
- Declare services/receivers/permissions in the **manifest**; respect modern background/battery restrictions.

Performance / Battery

Background limits exist to protect battery — foreground services and WorkManager are the sanctioned paths; unstopped services/receivers drain battery. Pause/stop work when not needed (08 · app_lifecycle).

Advantages / Disadvantages

- **+** Full Android integration (launch/share/deep links, ongoing tasks, event reactions), mostly via plugins.
- **−** Strict background limits + service-type/notification requirements, manifest/permission setup, background-isolate complexity, per-Android-version changes.

Interview Questions

1. 🟢 **What are Intents/Services/BroadcastReceivers?** — Intents launch activities/apps (share/open URL/deep links); Services run background/ongoing work; BroadcastReceivers react to system/app events.
2. 🟢 **How do you open a URL or share from Flutter?** — Via plugins (`url_launcher`, `share_plus`) that fire Android Intents.
3. 🟡 **Foreground service vs WorkManager?** — Foreground service for ongoing user-visible work (needs a notification + type); WorkManager for deferrable/guaranteed background tasks under background limits.
4. 🟡 **Why can't you rely on raw background services?** — Android 8+ restricts background execution; use foreground services or WorkManager.
5. 🟡 **How do receiver events reach Flutter?** — Stream them via an `EventChannel` (or use a plugin like `connectivity_plus`).

6. 🔴 **What's required for a foreground service on modern Android?** — A persistent notification, plus (Android 14+) a declared `foregroundServiceType` and matching permission.
7. 🔴 **How do background tasks use channels/isolates?** — They run in a background Flutter isolate needing `RootIsolateToken`/background messenger to access channels.

Senior Engineer Tips

- Default to plugins for integration; drop to custom Kotlin only for behavior no plugin provides.
- Choose the sanctioned background path (foreground service for visible/ongoing, WorkManager for deferrable) — raw background services will be killed and drain battery.
- Always stop services/unregister receivers; leaked native resources hurt battery and stability.

Architect Perspective

Android integration (intents/services/receivers) connects the Flutter app to the OS app model within strict modern constraints. Using plugins + the correct background mechanism (foreground service/WorkManager) behind repositories keeps the app compliant, battery-friendly, and portable — and ties into background services, notifications, and device features (Modules 33, 32, 29).

Summary

- Integrate via Intents (launch/share/deep links), Services (foreground for ongoing; WorkManager for deferrable background), and BroadcastReceivers (event reactions) — mostly through plugins.
- Respect background limits (Android 8+): foreground services need notifications/types; use WorkManager, not raw background services.
- Declare in manifest, stop/unregister resources, bridge via channels/plugins (+ bg-isolate setup).

Revision Notes

- Intents (url_launcher/share_plus/deep links), Services (foreground = notification + type/perm; deferrable = WorkManager), Receivers (dynamic; connectivity_plus) → bridge via channels/plugins.
- Android 8+ background limits: no raw background services; Android 14+ foreground service types.
- Stop services/unregister receivers (battery/leaks); bg isolate needs token/messenger.
- Prefer plugins; declare in manifest; respect battery/lifecycle.

Practice Questions

1. When use a foreground service vs WorkManager?
2. Why do raw background services fail on modern Android?
3. How do BroadcastReceiver events reach Flutter?

Coding Questions

1. Open a URL and share text using plugins (Intents).
2. Sketch a foreground service (notification + manifest type) for ongoing work.
3. Stream connectivity changes to Flutter (plugin or `EventChannel`).

Mini Project — Module capstone

Android integration slice (Flutter + Android): Combine the module — open-URL/share via `url_launcher` / `share_plus` (Intents), a permission-gated camera (runtime permission), a native view embed (platform view), and connectivity reactions (`connectivity_plus` / `EventChannel`), plus a foreground-service sketch for ongoing work — all bridged to Flutter behind repositories, within background limits. Acceptance: intents/share work; permissions handled; native view embedded; connectivity streamed; foreground service correctly declared; resources stopped/unregistered; runs on device.