28 · Native iOS

Flutter Practice Notes

Contents

28 · Native iOS

Introduction

This module covers the **iOS platform side** of a Flutter app: the iOS/Xcode project and CocoaPods, writing Swift (channel handlers, `AppDelegate` / `FlutterViewController`, lifecycle), embedding native iOS views (`UiKitView`), `Info.plist` and permissions (usage strings), and iOS integration (capabilities, background modes, universal links). It mirrors Native Android (Module 27).

Why this module exists

Flutter apps ship as iOS apps: you configure Xcode/CocoaPods, add capabilities and Info.plist usage strings, write Swift for platform channels/plugins, embed native views, and integrate with iOS lifecycle/background. Correct iOS setup is required to build, run, and pass App Store review.

Module map

#	File	Topic	Level
1	01_ios_project_and_cocoapods.md	Xcode project, CocoaPods, build config, schemes/flavors	🔵
2	02_swift_plugin_code.md	Swift channels, `AppDelegate` / `FlutterViewController`, lifecycle	🔴
3	03_platform_views_ios.md	Embedding native iOS views (`UiKitView`)	🔴
4	04_infoplist_and_permissions.md	`Info.plist`, permission usage strings, runtime flow	🔵
5	05_ios_integration.md	Capabilities, background modes, universal links, App Store	🔴

> **Cross-references:** Platform channels/plugins: Module 26. Android counterpart: Module 27. Embedder/startup: 10 · embedder_and_startup. Device features: Module 29. Deep links/universal links: 13 · deep_linking. Deployment/signing: Module 51.

Prerequisites

26 Platform Channels, 27 Native Android (many concepts mirror). Basic Swift + a Mac/Xcode help.

What you'll be able to do after this module

- Navigate the iOS project + manage CocoaPods and Xcode config (versions, schemes, signing basics).
- Write Swift channel handlers with correct `AppDelegate` / `FlutterViewController` /lifecycle usage.

- Embed native iOS views via `UiKitView`.
- Add `Info.plist` usage strings and handle iOS permissions.
- Configure capabilities, background modes, and universal links.

Capstone

iOS integration slice: A Swift `MethodChannel` / `EventChannel` reading a system value, a permission-gated feature (Info.plist usage string), an embedded native iOS view, and a capability/background-mode + universal-link integration — bridged to Flutter behind a repository.

Summary

The iOS side is Xcode/CocoaPods config + Swift (channels/AppDelegate/lifecycle) + Info.plist/permissions + platform views + capabilities/background/universal links. Master it to ship iOS apps and implement native features — mirroring Android with iOS-specific idioms.

iOS Project Structure, CocoaPods & Xcode

> A Flutter app's `ios/` folder is a real Xcode project: the `Runner` app (with `AppDelegate`, `Info.plist`, assets), a `Podfile` managing native dependencies via **CocoaPods**, and Xcode **schemes/build configs** for signing, deployment target, and flavors — you configure these to build, sign, and add native libraries.

Introduction

`flutter create` generates a standard iOS project under `ios/` (opened in Xcode via `Runner.xcworkspace`). This file maps its structure (`Runner`, `AppDelegate`, `Info.plist`, `Assets.xcassets`, `Podfile`), CocoaPods dependency management, and key Xcode config (deployment target, signing, schemes/flavors) — the foundation for Swift code and iOS features.

Why this concept exists

Flutter compiles to a native iOS app; iOS uses Xcode + CocoaPods for building, dependencies, signing, and capabilities. You must set the deployment target, manage pods (installed by Flutter tooling), configure signing/team, and (optionally) schemes for flavors. Flutter wraps but doesn't replace Xcode/CocoaPods.

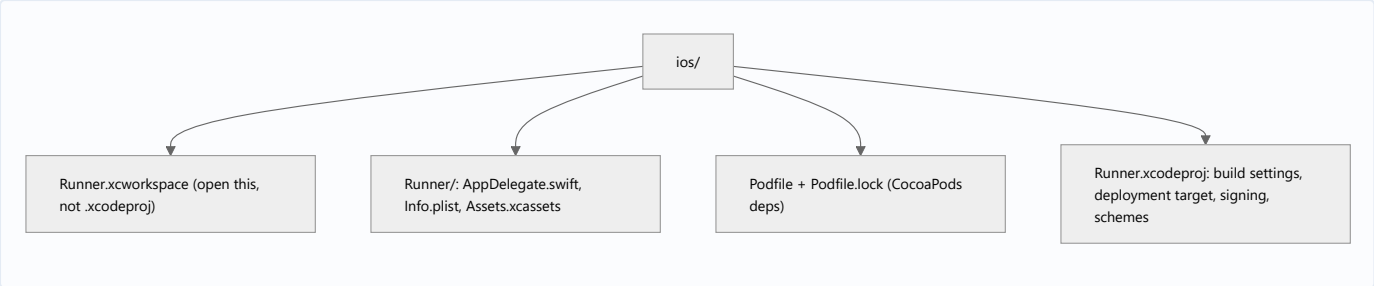
Real-world analogy

The `ios/` project is the **iOS chassis** your Flutter body mounts onto; **CocoaPods** is the **parts supplier** delivering native dependencies; **Xcode schemes/configs** are the **build/trim/certification specs** (deployment target, signing team, flavor). You tune these for the platform and App Store.

Problem Statement

You must raise the iOS deployment target for a plugin, understand where `AppDelegate` / `Info.plist` live, add a native pod, set the signing team, and add dev/prod schemes. You'll map the project + configure Xcode/CocoaPods.

Internal Working



- `Runner.xcworkspace`: open this in Xcode (CocoaPods requires the workspace, not the `.xcodeproj`).
- `Runner/`: `AppDelegate.swift` (hosts the Flutter engine / registers plugins — 02_swift_plugin_code.md), `Info.plist` (app config, permissions/usage strings — 04_infoplist_and_permissions.md), `Assets.xcassets` (icons/launch), `Base.lproj` (launch storyboard).
- **CocoaPods**: `Podfile` declares native dependencies; `flutter pub get` + build runs `pod install` (creating `Podfile.lock`). Plugins add their pods automatically. Set the **platform version** in the `Podfile` (e.g., `platform :ios, '13.0'`).
- **Xcode build config**: **iOS Deployment Target** (min iOS version — plugins often dictate a floor), **Signing & Capabilities** (team, bundle id, provisioning — Module 51), **build configurations** (Debug/Release/Profile), and **schemes**.
- **Flavors/schemes**: iOS uses **schemes + `xcconfig`/build configs** (and sometimes multiple targets) for dev/prod (distinct bundle ids/config); build with `flutter build ios --flavor prod -t lib/main_prod.dart` (requires matching schemes/configs). More manual than Android flavors.
- **Deployment target**: raise it if a plugin/pod requires a newer minimum iOS.

Memory Representation

Not applicable; build-time config. Release builds AOT-compile Dart + strip symbols (pair with `--obfuscate` / `--split-debug-info` — 21 · startup_and_app_size).

Compiler Behavior

Xcode (clang/Swift) compiles native code + links pods; Flutter's AOT Dart is embedded. Deployment target/Swift/pod versions must be compatible.

Runtime Behavior

`AppDelegate` (a `FlutterAppDelegate`) starts the engine and runs your Dart `main`. Schemes select config/bundle id at build.

Flutter Engine Behavior

`FlutterAppDelegate` / `FlutterViewController` is the iOS **embedder** hosting the engine + registering plugins (10 · embedder_and_startup).

Dart VM Behavior

Not applicable.

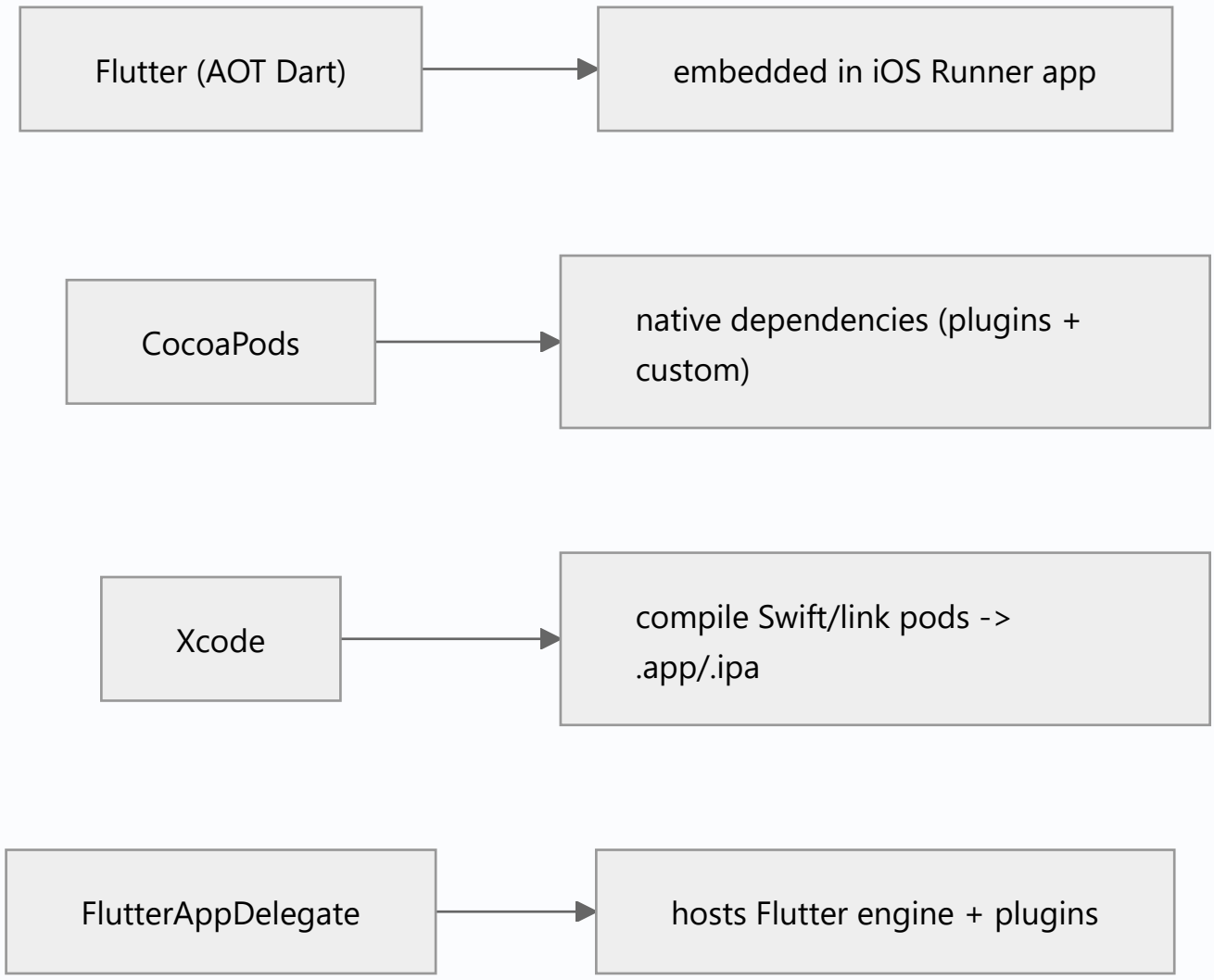
Examples

```
# ios/Podfile — set platform version; plugins add their pods automatically
platform :ios, '13.0'   # raise for plugins that require a newer minimum

target 'Runner' do
  use_frameworks!
  # Flutter installs plugin pods here; add custom pods if needed:
  # pod 'SomeNativeSDK', '~> 2.0'
end
```

```
// ios/Runner/AppDelegate.swift — hosts the Flutter engine; register channels/plugins here
import Flutter
import UIKit

@main
@objc class AppDelegate: FlutterAppDelegate {
  override func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?
  ) -> Bool {
    GeneratedPluginRegistrant.register(with: self) // registers plugins
    // register custom channels here (see swift_plugin_code.md)
    return super.application(application, didFinishLaunchingWithOptions: launchOptions)
  }
}
```

```
Build with a flavor/scheme + entrypoint (requires matching Xcode schemes/configs):
  flutter build ios --flavor prod -t lib/main_prod.dart
  flutter build ipa --release   # for App Store (Module 51)
CocoaPods (run automatically by Flutter build, or manually):
  cd ios && pod install
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Opening `.xcodeproj` not `.xcworkspace`	Pods not linked	Open `Runner.xcworkspace`
Deployment target too low for a plugin	Build/pod failure	Raise iOS deployment target + Podfile platform
Editing pods by hand	Drift/lock conflicts	Let `pod install` manage; commit `Podfile.lock`
No signing team/provisioning	Can't run on device/ship	Configure Signing & Capabilities (Module 51)
Expecting Android-style flavors	iOS uses schemes/configs	Set up schemes + build configs for dev/prod
Skipping Info.plist usage strings	Permission crashes/rejection	Add usage descriptions (04_infoplist_and_permissions.md)

Best Practices

- Open the `.xcworkspace`; let Flutter/CocoaPods manage pods; **commit** `Podfile.lock` for reproducible builds.
- Set the **iOS deployment target** (and `Podfile platform`) to satisfy plugins/pods; keep Xcode/Swift/CocoaPods compatible.
- Configure **Signing & Capabilities** (team, bundle id) early; enable capabilities as needed (05_ios_integration.md).
- Set up **schemes + build configs** for dev/prod (distinct bundle ids); build with `--flavor` + entrypoint.
- Register channels/plugins in `AppDelegate` (02_swift_plugin_code.md); add **Info.plist usage strings** for permissions.

Performance

Release AOT + symbol stripping + app thinning (per-device via App Store) reduce size; correct target/config avoids issues. Build-time only (21 · startup_and_app_size).

Advantages / Disadvantages

- + Full iOS build control (target/deps/signing/capabilities/schemes), standard Xcode/CocoaPods, native dependency support.
- − Xcode/CocoaPods/signing complexity, Mac required, flavors more manual than Android, provisioning/capabilities overhead.

Interview Questions

1. 🟢 **Where is the iOS project in a Flutter app, and what do you open?** — In `ios/`; open `Runner.xcworkspace` (not the `.xcodeproj`) so CocoaPods deps are linked.
2. 🟢 **What manages native iOS dependencies?** — CocoaPods (`Podfile` / `Podfile.lock`); Flutter runs `pod install`, and plugins add pods automatically.
3. 🟡 **What is `AppDelegate`'s role?** — A `FlutterAppDelegate` that hosts the engine and registers plugins/channels (in `didFinishLaunchingWithOptions`).
4. 🟡 **What's the iOS deployment target and why raise it?** — The minimum iOS version supported; plugins/pods may require a newer floor.
5. 🟡 **How do iOS flavors differ from Android?** — iOS uses schemes + build configs/`xcconfig` (and sometimes targets), which is more manual than Android's `productFlavors`.
6. 🔴 **Why must `Podfile.lock` be committed?** — For reproducible builds (pins pod versions across the team/CI).
7. 🔴 **What's required to run on a device / ship?** — Signing & Capabilities (team, bundle id, provisioning profile) — Module 51.

Senior Engineer Tips

- Always open the **workspace**, commit `Podfile.lock`, and let Flutter manage pods; hand-editing causes lock/version pain.
- Align iOS deployment target with your plugins; set signing/team early to avoid device/CI blockers.
- Set up dev/prod schemes + configs (matching Firebase/entrypoints) to prevent config bleed — accept it's more manual than Android.

Architect Perspective

The iOS project/Xcode/CocoaPods layer turns the Flutter app into a shippable iOS artifact: deployment target, dependencies, signing, capabilities, and schemes. Configuring it correctly (versions, signing, flavors, capabilities) is a delivery/compatibility concern underpinning native integration and App Store deployment (Module 51).

Summary

- `ios/` is an Xcode project: `Runner` (`AppDelegate` / `Info.plist`/assets), `Podfile` (CocoaPods), build configs/schemes; open the `.xcworkspace`.
- Set deployment target/pods, signing/team, schemes for dev/prod; register plugins in `AppDelegate`; add Info.plist usage strings.
- More manual (schemes/signing) than Android; commit `Podfile.lock`.

Revision Notes

- Open `Runner.xcworkspace`; `Runner/` = `AppDelegate.swift` / `Info.plist` /assets; CocoaPods `Podfile` / `Podfile.lock` (commit it).
- Deployment target (+ `Podfile platform`) per plugins; Signing & Capabilities (team/bundle id); schemes+configs for dev/prod flavors.
- Register channels/plugins in `AppDelegate.didFinishLaunchingWithOptions`; Info.plist usage strings for permissions.
- Release AOT + thinning; more manual than Android.

Practice Questions

1. Why open `.xcworkspace` and commit `Podfile.lock`?
2. What is `AppDelegate`'s role and where do you register plugins?
3. How do iOS flavors differ from Android's?

Coding Questions

1. Set the iOS deployment target + Podfile platform and add a custom pod.
2. Register a plugin/channel in `AppDelegate`.
3. Configure a dev/prod scheme + build config (outline).

Mini Project

iOS build config (Flutter/Xcode): Configure the iOS project — set deployment target + Podfile platform, add a native pod, configure signing/team, and set up dev/prod schemes + entrypoints; confirm `AppDelegate` hosts the engine. Build both schemes. Acceptance: workspace opens with pods; target/deps correct; schemes build; signing configured; runs on device.

Swift Plugin Code (Channels, `AppDelegate` / `FlutterViewController`, Lifecycle)

> On iOS, native channel handlers are Swift: register them in `AppDelegate` (via the `FlutterViewController`'s binary messenger) or in a plugin (`FlutterPlugin`), return results with the `FlutterResult` callback (`value` / `FlutterError` / `FlutterMethodNotImplemented`), and offload heavy work off the main thread — replying on it.

Introduction

This file covers the iOS/Swift side of channels/plugins: where to register handlers (`AppDelegate` / `FlutterViewController` / `FlutterPlugin`), the `FlutterResult` reply pattern, `EventChannel` stream handlers, threading (main thread/GCD), and lifecycle — the iOS mirror of the Kotlin file (27 · kotlin_plugin_code).

Why this concept exists

Channels need a native implementation; on iOS that's Swift with iOS-specific concerns: getting the `FlutterViewController`/binary messenger, replying via `FlutterResult`, threading with GCD (`DispatchQueue`), and integrating with UIKit lifecycle. Getting these right avoids crashes/hangs and enables native capabilities.

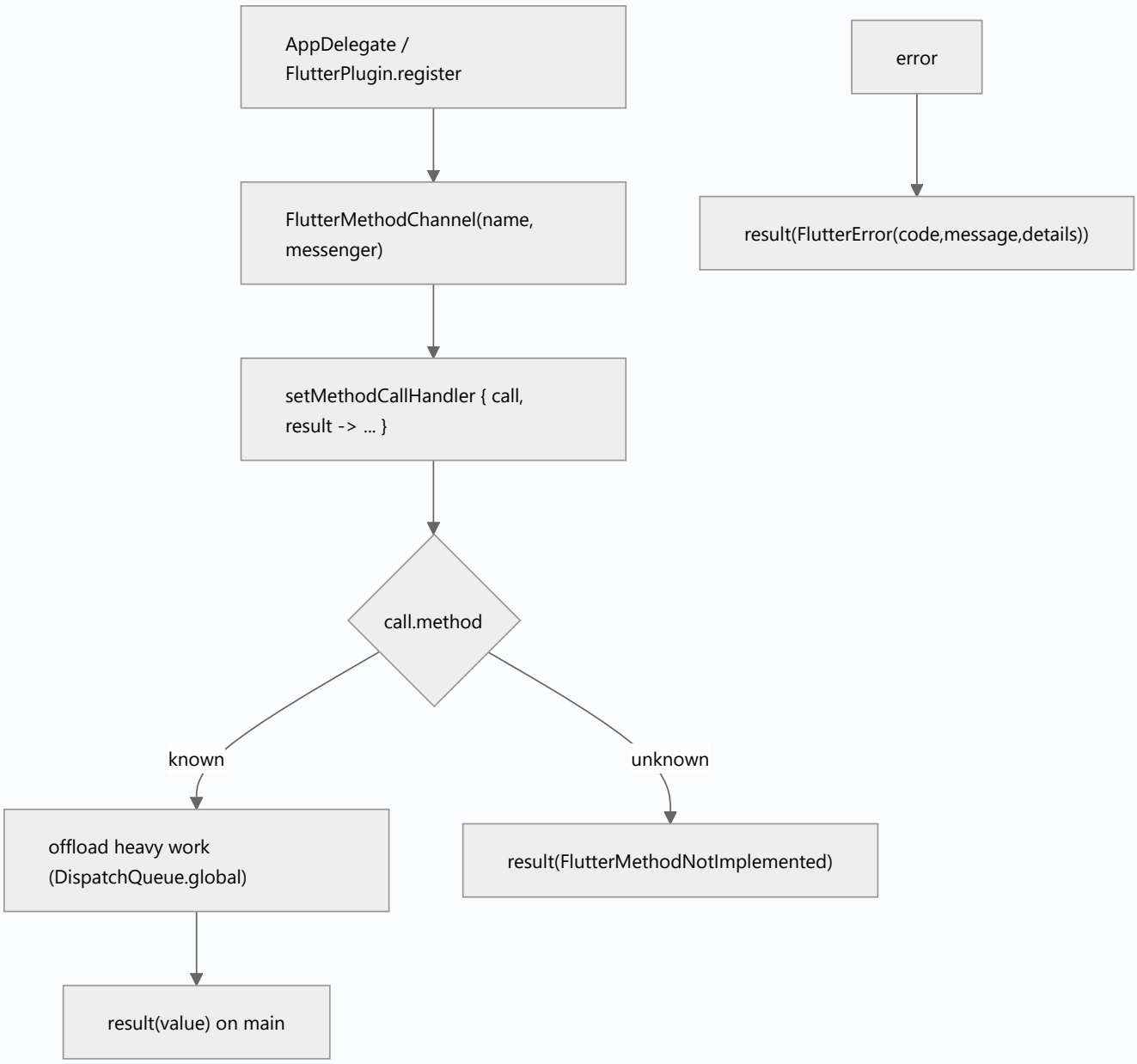
Real-world analogy

Writing Swift channel code is the **iOS branch office** fulfilling HQ (Dart) requests: it uses iOS credentials (view controller/APIs), does work possibly on a background queue, and **hands the reply back through the correct window** (`result` on the main thread).

Problem Statement

Implement a Swift `MethodChannel` that reads a system value and one that does heavy work off the main thread, plus an `EventChannel` stream — registered in `AppDelegate` (or a `FlutterPlugin`), with correct threading and reply. You'll register handlers and reply via `FlutterResult`.

Internal Working



- **Registering handlers**:
  - **App-embedded**: in `AppDelegate.didFinishLaunchingWithOptions`, get the `FlutterViewController` (`window?.rootViewController as! FlutterViewController`) and create `FlutterMethodChannel(name:, binaryMessenger: controller.binaryMessenger)` + `setMethodCallHandler`.
  - **Plugin**: implement `FlutterPlugin` (`register(with registrar:)`) and use the registrar's messenger; register via `GeneratedPluginRegistrant`.
- **Replying (`FlutterResult`)**: call `result(value)` for success, `result(FlutterError(code:message:details:))` for errors, `result(FlutterMethodNotImplemented)` for unknown methods. Reply **exactly once**, on the **main thread**.

- **Args**: `call.arguments` (cast from `Any?`, usually a `[String: Any]` dict); codec-supported types only.
- **Threading**: handlers run on the **main thread** (main run loop). Do heavy work on a **background queue** (`DispatchQueue.global().async`) and call `result` back on the **main queue** (`DispatchQueue.main.async`) to avoid blocking UI (26 · platform_channel_fundamentals).
- `EventChannel`: implement `FlutterStreamHandler` (`onListen(withArguments:eventSink:)` / `onCancel`); emit via the `FlutterEventSink` on the main thread; clean up in `onCancel` (26 · event_channel).
- **Lifecycle**: `AppDelegate` hooks (background/foreground, URL handling) integrate with app lifecycle (05_ios_integration.md).

Memory Representation

Retaining strong references to controllers/observers past their lifetime can leak; use appropriate ownership and remove observers on `onCancel`/deinit (26 · event_channel).

Compiler Behavior

Swift compiled by Xcode; channels are stringly-typed (Pigeon generates Swift type-safety — 26 · plugins_and_pigeon).

Runtime Behavior

Handlers dispatch by `call.method`; unknown → `FlutterMethodNotImplemented` → Dart `MissingPluginException`; `FlutterError` → Dart `PlatformException`. Heavy main-thread work blocks UI.

Flutter Engine Behavior

`FlutterAppDelegate` / `FlutterViewController` embeds the engine + registers plugins; channel messages marshal via the engine (10 · engine_internals).

Dart VM Behavior

Not applicable.

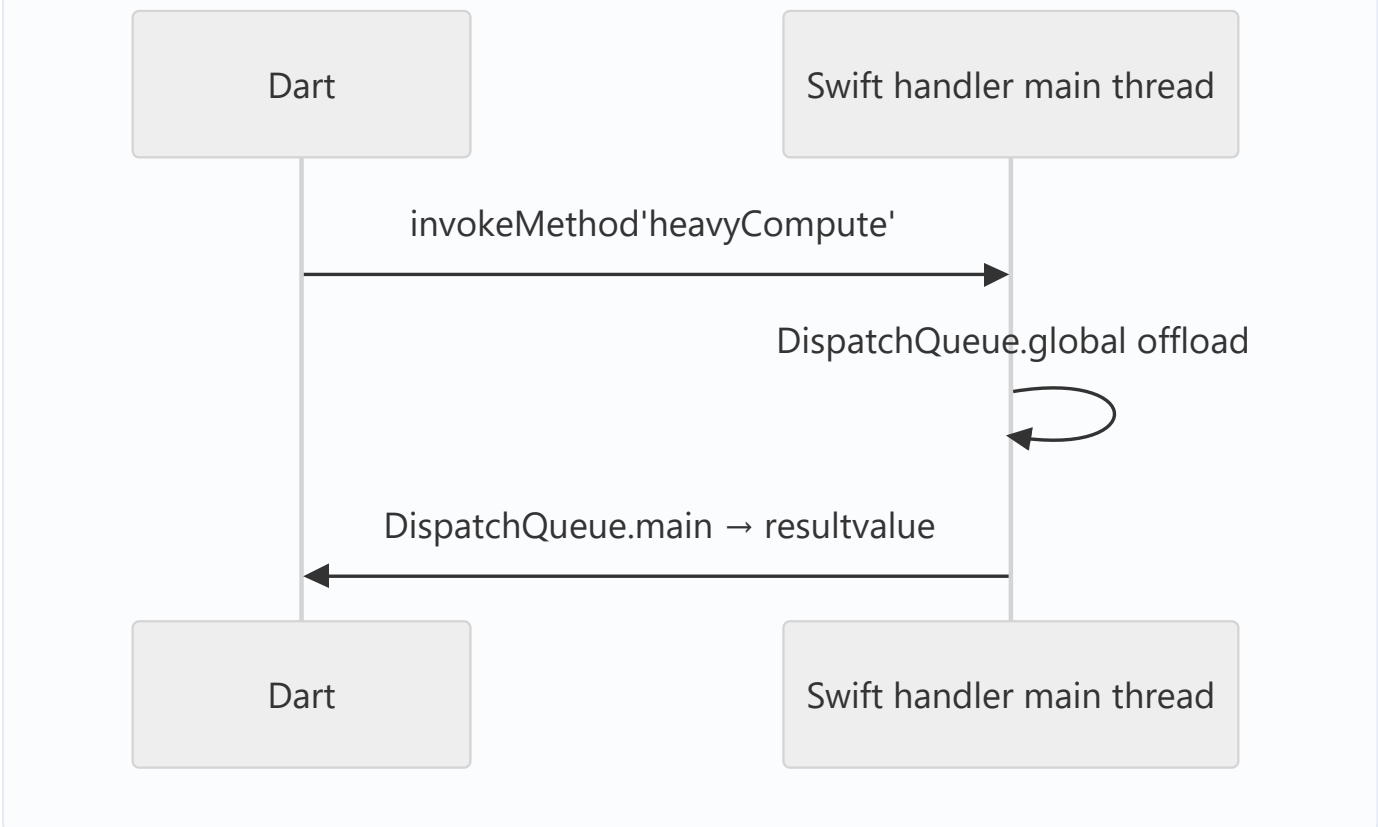
Examples

```
import Flutter
import UIKit


@main
@objc class AppDelegate: FlutterAppDelegate {
```

```
  override func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?
  ) -> Bool {
    let controller = window?.rootViewController as! FlutterViewController
    let channel = FlutterMethodChannel(name: "app/system", binaryMessenger: controller.binaryMessenger)

    channel.setMethodCallHandler { call, result in
      switch call.method {
      case "getBatteryLevel":
        UIDevice.current.isBatteryMonitoringEnabled = true
        let level = UIDevice.current.batteryLevel
        if level < 0 { result(FlutterError(code: "UNAVAILABLE", message: "No battery info", details: nil))
}
        else { result(Int(level * 100)) }                // reply on main (already on main)

      case "heavyCompute":
        DispatchQueue.global().async {                    // offload heavy work
          let value = (1...1_000_000).reduce(0, +)
          DispatchQueue.main.async { result(value) }      // reply on MAIN queue
        }

      default:
        result(FlutterMethodNotImplemented)               // unknown method
      }
    }

    GeneratedPluginRegistrant.register(with: self)
    return super.application(application, didFinishLaunchingWithOptions: launchOptions)
  }
}
```

```
// EventChannel stream handler (native -> Dart)
class BatteryStreamHandler: NSObject, FlutterStreamHandler {
  var sink: FlutterEventSink?
  func onListen(withArguments args: Any?, eventSink: @escaping FlutterEventSink) -> FlutterError? {
    sink = eventSink
    NotificationCenter.default.addObserver(self, selector: #selector(changed),
      name: UIDevice.batteryStateDidChangeNotification, object: nil)
    return nil
  }
  @objc func changed() { DispatchQueue.main.async { self.sink?("charging") } } // emit on main
  func onCancel(withArguments args: Any?) -> FlutterError? {
    NotificationCenter.default.removeObserver(self); sink = nil; return nil     // cleanup
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Heavy work on the main thread	Blocks UI/hangs	Offload to `DispatchQueue.global`; reply on main
Replying off the main thread	Errors/crashes	`DispatchQueue.main.async { result(...) }`
Calling `result` more than once/never	Crash/hang	Reply exactly once per call
Not handling unknown methods	Missing handler	`result(FlutterMethodNotImplemented)`
Not removing observers (EventChannel)	Leak	Remove in `onCancel`/deinit
Passing custom objects	Codec unsupported	Send dicts/primitives (or Pigeon)

Best Practices

- Register handlers in `AppDelegate` (app) or a `FlutterPlugin` (reusable), using the correct **binary messenger**.
- Reply via `FlutterResult` exactly once, on the **main thread**; use `FlutterError` / `FlutterMethodNotImplemented` appropriately.
- **Offload heavy work** to `DispatchQueue.global()`; return results on `DispatchQueue.main`.
- For streams, implement `FlutterStreamHandler` and **remove observers in** `onCancel`; emit on main.
- Use **codec-supported types**; adopt **Pigeon** for Swift type-safety; wrap the Dart side in a repository.

Performance

Main-thread handlers must return quickly — offload heavy work; keep messages small. Threading discipline avoids UI hangs (26 · platform_channel_fundamentals, Module 21).

Advantages / Disadvantages

- + Full iOS/Swift API access, GCD-based offloading, proper lifecycle integration.
- − Threading/reply-once discipline, stringly-typed channels (Pigeon helps), observer cleanup, Swift/iOS knowledge + Mac required.

Interview Questions

1. 🟢 **Where do you register iOS channel handlers?** — In `AppDelegate.didFinishLaunchingWithOptions` (via `FlutterViewController.binaryMessenger`) or a `FlutterPlugin`.

2. 🟢 **How do you reply from Swift?** — Call the `FlutterResult` callback once: `result(value)`, `result(FlutterError(...))`, or `result(FlutterMethodNotImplemented)` — on the main thread.
3. 🟡 **How do you offload heavy work?** — `DispatchQueue.global().async { ... DispatchQueue.main.async { result(value) } }` — compute off-main, reply on main.
4. 🟡 **How do you stream events (EventChannel) on iOS?** — Implement `FlutterStreamHandler` (`onListen` with a `FlutterEventSink`, `onCancel` to clean up); emit on main.
5. 🟡 **How do you read arguments?** — From `call.arguments` (cast, usually `[String: Any]`), codec-supported types only.
6. 🔴 **What happens if you call `result` twice or never?** — Crash/hang — reply exactly once per call.
7. 🔴 **How do you make Swift channel code type-safe?** — Use Pigeon-generated Swift interfaces instead of stringly-typed `setMethodCallHandler`.

Senior Engineer Tips

- Mirror the Android discipline: offload heavy native work, reply on the main thread exactly once, and clean up observers — the top iOS channel bugs.
- Prefer a `FlutterPlugin` (reusable, `register(with:)`) over ad-hoc `AppDelegate` code as the native surface grows.
- Use Pigeon for anything beyond a couple methods; it eliminates Swift-side name/type mismatches.

Architect Perspective

The Swift channel layer is the iOS implementation of native features; correct threading (GCD/main), single-reply, and observer cleanup make it responsive and leak-free. Packaged as a plugin (+ Pigeon types) and wrapped behind repositories, it mirrors Android for a portable, testable native-integration story (Module 26, 27 · kotlin_plugin_code).

Summary

- Register handlers in `AppDelegate` / `FlutterPlugin`; reply via `FlutterResult` once, on the main thread.
- Offload heavy work (GCD) and reply on main; stream via `FlutterStreamHandler` (clean up in `onCancel`).
- Codec types (or Pigeon); wrap Dart side in a repository — the iOS mirror of Kotlin channel code.

Revision Notes

- Register: `AppDelegate` (via `FlutterViewController.binaryMessenger`) or `FlutterPlugin.register(with:)`.
- Reply: `result(value)` / `FlutterError` / `FlutterMethodNotImplemented` — once, on main.
- Offload: `DispatchQueue.global` → `DispatchQueue.main` for reply; EventChannel = `FlutterStreamHandler` (remove observers in `onCancel`).
- Codec args (`[String:Any]`); Pigeon for type-safety; repository-wrap Dart side.

Practice Questions

1. How do you offload heavy work and reply correctly on iOS?
2. What are the three `FlutterResult` outcomes?
3. How do you implement and clean up an `EventChannel` on iOS?

Coding Questions

1. Write a Swift `MethodChannel` handler for `getBatteryLevel` + `heavyCompute` (offloaded).
2. Implement a `FlutterStreamHandler` for battery events (cleanup in `onCancel`).
3. Return a `FlutterError` for a failure case and handle unknown methods.

Mini Project

Swift channel service (Flutter + iOS): Implement a Swift `MethodChannel` (`getBatteryLevel`, `heavyCompute` offloaded → main-thread reply) and an `EventChannel` (battery events via `FlutterStreamHandler`, cleaned up in `onCancel`), registered in `AppDelegate` (or a `FlutterPlugin`), and wrap the Dart side in a repository. Acceptance: correct threading (offload + reply on main); reply-once; stream cleanup; results/errors returned; runs on device/simulator.

Platform Views (Embedding Native iOS Views — `UiKitView`)

> Platform views embed a native iOS `UIView` (MapKit, WKWebView, AVFoundation camera preview, SDK views) inside the Flutter tree via `UiKitView` + a native `FlutterPlatformViewFactory` — powerful but heavier than Flutter widgets (compositing overhead), so use them only when a Flutter widget can't do the job.

Introduction

To show a genuine native iOS view (a `WKWebView`, `MKMapView`, camera preview, or third-party SDK view) inside Flutter, use a platform view. This file covers the iOS mechanism (`UiKitView` + `FlutterPlatformViewFactory` / `FlutterPlatformView`), creation params, and the performance tradeoffs — the iOS mirror of Android platform views (27 · platform_views_android).

Why this concept exists

Flutter renders its own UI and can't natively draw a `WKWebView` / `MKMapView`. Platform views composite a native `UIView` into the Flutter surface — the mechanism behind plugins like `webview_flutter`, `google_maps_flutter`, and `camera` on iOS.

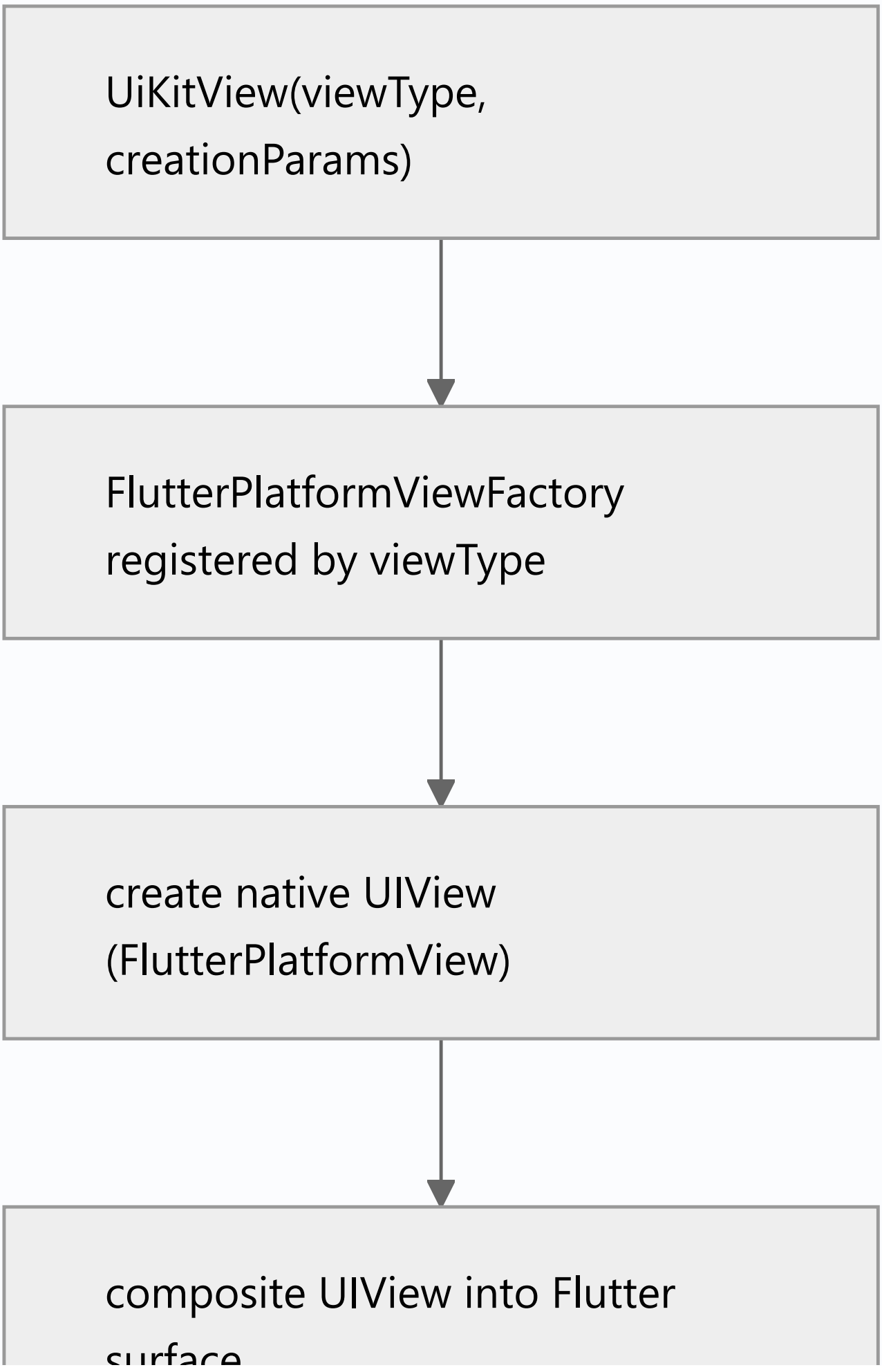
Real-world analogy

A platform view is a **window into the native room** cut through your Flutter-painted wall — you see a real `UIView` through it. The window/compositing costs more than painting the wall, so add them only where you truly need the real native view.

Problem Statement

Embed a native iOS `WKWebView` (or custom `UIView`) inside a Flutter screen, register its factory, pass creation params, and understand the cost. You'll register a `FlutterPlatformViewFactory` and use `UiKitView`.

Internal Working


UiKitView(viewType, creationParams)
FlutterPlatformViewFactory registered by viewType
create native UIView (FlutterPlatformView)
composite UIView into Flutter surface

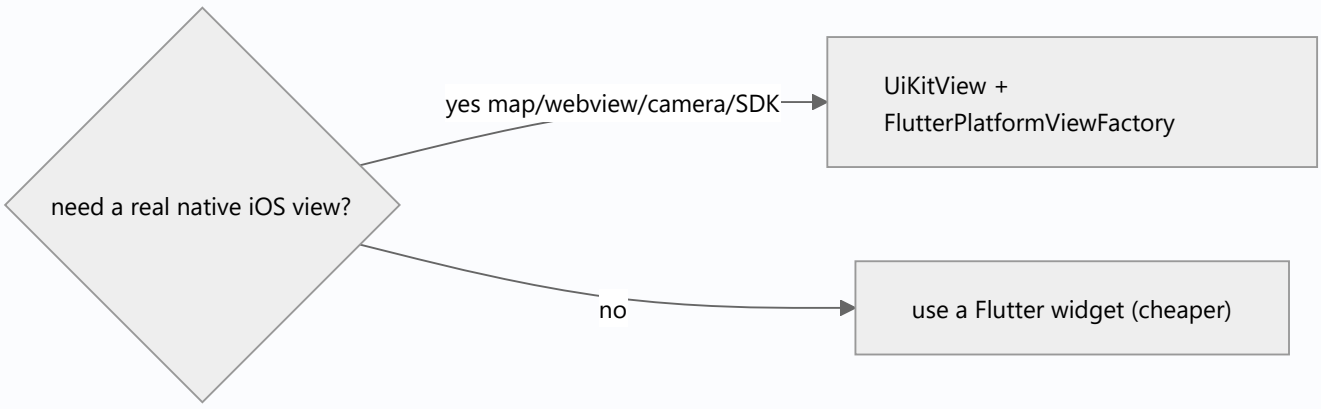
surface
```

- **Dart side**: `UiKitView(viewType: 'app/native-webview', creationParams: {...}, creationParamsCodec: StandardMessageCodec())` — or `PlatformViewLink` for gesture control. `viewType` matches a registered factory.
- **Native side (Swift)**: implement a `FlutterPlatformView` (returns your `UIView`) and a `FlutterPlatformViewFactory` (creates it with args); register via `registrar.register(factory, withId: viewType)` in your plugin.
- **Composition**: iOS composites the native `UIView` with Flutter content (hybrid composition); heavier than pure Flutter layers.
- **Cost**: platform views add compositing/texture overhead and gesture/keyboard/z-order complexity — use **sparingly**, avoid in fast-scrolling lists.
- **Gestures/keyboard**: input routing to the native view is mostly handled by the platform-view system; complex cases use `PlatformViewLink`/gesture recognizers.
- **Cleanup**: release native resources (e.g., `WKWebView`) when the view is removed.

## Memory Representation

A native `UIView` + compositing/texture buffers live alongside Flutter's layer tree — memory cost; release the native view when disposed.

## Compiler Behavior

Not applicable; factory/view are Swift, registered at plugin setup.

## Runtime Behavior

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

## Flutter Engine Behavior

The engine composites the native view into its surface (hybrid composition) — 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 iOS view by its registered viewType
class NativeWebViewIOS extends StatelessWidget {
  final String url;
  const NativeWebViewIOS({super.key, required this.url});
  @override
  Widget build(BuildContext context) {
    return UiKitView(
      viewType: 'app/native-webview',                // matches the registered factory
      creationParams: {'url': url},
      creationParamsCodec: const StandardMessageCodec(),
    );
  }
}
```

```
// iOS: FlutterPlatformView + Factory, registered by viewType
import Flutter
import WebKit

class NativeWebView: NSObject, FlutterPlatformView {
  private let webView: WKWebView
  init(frame: CGRect, viewId: Int64, args: Any?) {
    webView = WKWebView(frame: frame)
    super.init()
    if let params = args as? [String: Any], let url = params["url"] as? String,
       let u = URL(string: url) { webView.load(URLRequest(url: u)) }
  }
  func view() -> UIView { webView }   // return the native UIView
}

class NativeWebViewFactory: NSObject, FlutterPlatformViewFactory {
  func create(withFrame frame: CGRect, viewIdentifier viewId: Int64, arguments args: Any?) ->
FlutterPlatformView {
    NativeWebView(frame: frame, viewId: viewId, args: args)
  }
  func createArgsCodec() -> FlutterMessageCodec & NSObjectProtocol {
FlutterStandardMessageCodec.sharedInstance() }
}
// Register in the plugin/AppDelegate:
// registrar.register(NativeWebViewFactory(), withId: "app/native-webview")
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Platform views for things Flutter can do	Needless cost/complexity	Use Flutter widgets
Not releasing the native view	Leak (WKWebView/MapView)	Release on removal
Many/large platform views (esp. in lists)	Compositing/memory cost, jank	Minimize count/size; avoid in scroll
Gesture/keyboard conflicts	Input not reaching native view	Route gestures (`PlatformViewLink`)
`viewType` mismatch	View not created	Match Dart `viewType` to registered factory
Params without codec	Params not received	Use `creationParamsCodec` both sides

Best Practices

- Use platform views **only when necessary** (map/webview/camera/SDK views); prefer Flutter widgets otherwise.
- **Release native resources** when the view is disposed; keep platform views **few/small** and out of fast lists.
- Match `viewType` and use a **codec** for creation params; route gestures/keyboard for complex views.
- Prefer maintained **plugins** (`webview_flutter` , `google_maps_flutter` , `camera`) over hand-rolling; they handle the platform-view/gesture/lifecycle complexity.
- Profile compositing/raster when adding them (21 · jank_and_raster).

Performance

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

Advantages / Disadvantages

- **+** Embed real native iOS views (map/webview/camera/SDKs) inside Flutter.
- **−** More expensive than Flutter widgets (compositing/memory), gesture/keyboard/z-order complexity, Swift code, disposal responsibility.

Interview Questions

1. 🟢 **What embeds a native iOS view in Flutter?** — `UiKitView` on the Dart side, backed by a registered `FlutterPlatformViewFactory` creating a `FlutterPlatformView` .
2. 🟢 **When should you use a platform view?** — Only for genuinely native views (map/webview/camera/SDK); use Flutter widgets otherwise.

3. 🟡 **Why are platform views more expensive than Flutter widgets?** — They add native-view compositing/texture overhead beyond Flutter's own layers (hybrid composition).
4. 🟡 **How do you pass creation params?** — Via `creationParams` + `creationParamsCodec` on Dart, read from `args` in the factory/view on Swift.
5. 🟡 **How do you avoid leaking the native view?** — Release native resources (e.g., the `WKWebView`) when the platform view is removed.
6. 🔴 **What complexities do platform views add?** — Gesture/keyboard routing, z-ordering with Flutter content, compositing cost — especially in scrolling lists.
7. 🔴 **iOS vs Android platform views — same idea?** — Yes: `UiKitView` / `FlutterPlatformViewFactory` (iOS) mirrors `AndroidView` / `PlatformViewFactory` (Android); both composite native views at extra cost.

Senior Engineer Tips

- Prefer existing plugins (webview/maps/camera) — they encapsulate the platform-view + gesture/lifecycle work on both platforms.
- Keep platform views small/few and out of fast scroll views; always release the native view on disposal.
- Profile compositing/raster when embedding; native views are the usual suspects for iOS/Android jank in mixed UIs.

Architect Perspective

`UiKitView` is the iOS "embed a real native view" escape hatch — essential for maps/webview/camera/SDKs but costlier than Flutter rendering. Minimize and encapsulate (behind widgets/plugins), release resources, and prefer Flutter widgets — mirroring the Android approach for a consistent cross-platform strategy (27 · platform_views_android, 09 · compositing).

Summary

- iOS platform views: `UiKitView` + `FlutterPlatformViewFactory` / `FlutterPlatformView` embed a native `UIView`.
- Heavier than Flutter widgets (compositing/memory) — use sparingly, release resources, mind gestures/lists.
- Prefer Flutter widgets/maintained plugins; mirrors Android platform views.

Revision Notes

- Dart: `UiKitView(viewType, creationParams, creationParamsCodec)`; Swift: `FlutterPlatformView` + `FlutterPlatformViewFactory` registered by `viewType`.

- Hybrid composition; heavier than Flutter layers; release native view on disposal; minimize/avoid in lists.
- Match `viewType`, use codec for params, route gestures; prefer plugins (webview/maps/camera).

Practice Questions

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

Coding Questions

1. Embed a `WKWebView` via `UiKitView` + a `FlutterPlatformViewFactory`.
2. Pass creation params (url) with a codec and read them in Swift.
3. Release the native view on disposal.

Mini Project

Native iOS view embed (Flutter + iOS): Embed a native `UIView` (e.g., `WKWebView` or a simple custom view) via `UiKitView` + a registered `FlutterPlatformViewFactory` / `FlutterPlatformView`, pass creation params with a codec, and release it correctly. Keep it single/small, out of a list. Acceptance: native view renders inside Flutter; params passed; released (no leak); minimal/encapsulated; runs on device/simulator.

`Info.plist` & Permissions (Usage Strings)

> iOS requires a **usage-description string in `Info.plist`** for every sensitive capability (camera, location, mic, photos, contacts) — missing one crashes the app when requested; then the OS shows a **one-time runtime prompt**, and you handle authorized/denied/restricted (with a settings fallback), typically via `permission_handler`.

Introduction

`Info.plist` is the iOS app's configuration file (bundle id, display name, capabilities, and **privacy usage strings**). iOS shows a system permission prompt on first use of a sensitive API — but **only if you've declared a usage string**; otherwise it crashes. This file covers usage strings, the permission flow, and best practices — the iOS mirror of Android's manifest/permissions (27 · permissions_and_manifest).

Why this concept exists

Apple mandates transparency: apps must state *why* they need sensitive data (the usage string is shown in the prompt). This protects users and is an App Store requirement — missing/vague strings cause crashes or rejection.

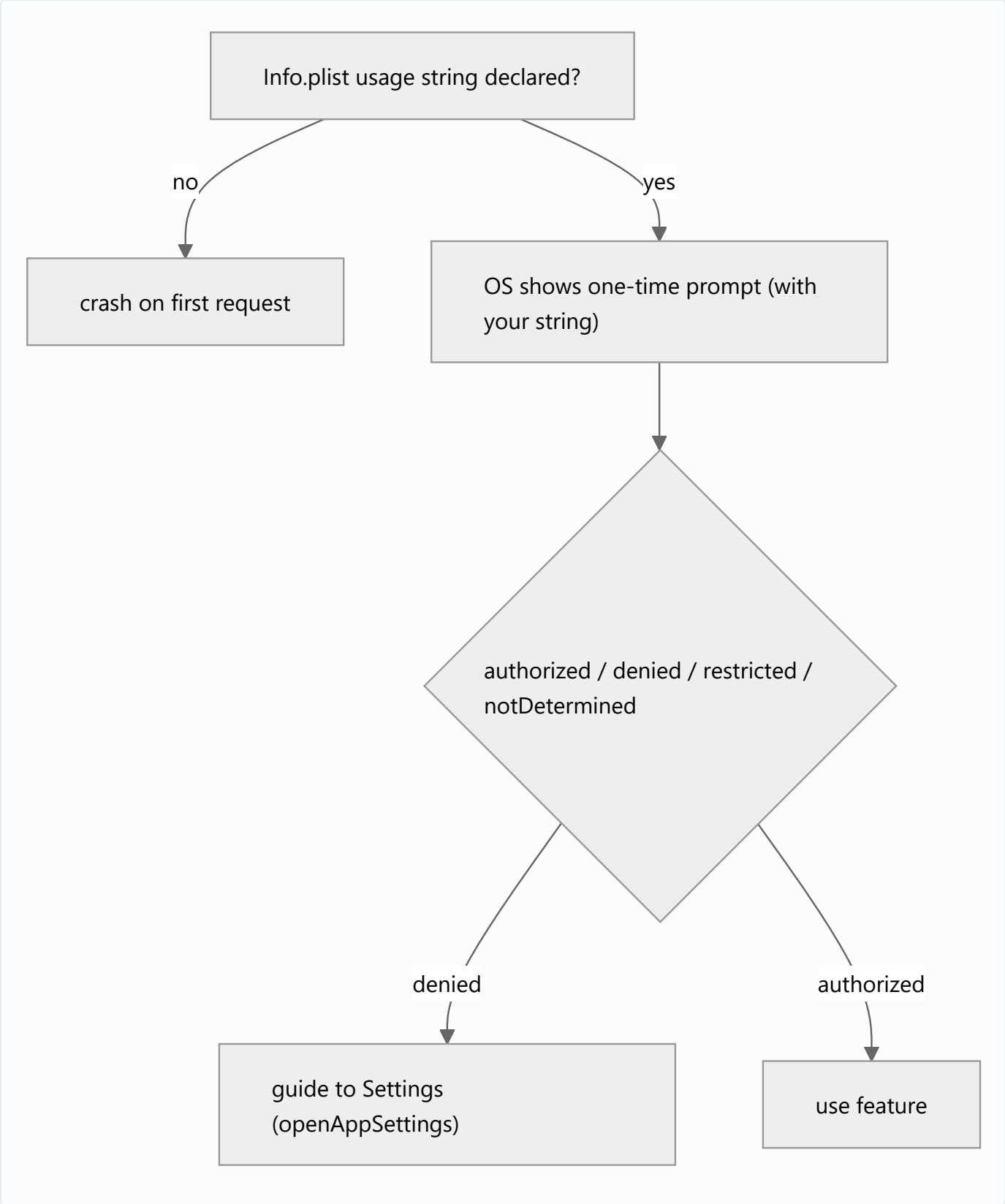
Real-world analogy

The usage string is your **written justification on the access-request form** ("we need the camera to scan receipts"). Without the form, security **refuses you at the door immediately** (crash); with it, they **ask the resident once** (system prompt), who can allow, deny, or say "settings only."

Problem Statement

Add camera + location to iOS: declare usage strings in `Info.plist`, request at runtime when the feature is used, handle authorized/denied/restricted, and open Settings when denied — via `permission_handler` behind a repository. You'll edit `Info.plist` + handle the flow.

Internal Working



- `Info.plist` **usage strings**: add a key + human-readable reason per capability, e.g.:
  - `NSCameraUsageDescription` — camera
  - `NSLocationWhenInUseUsageDescription` / `NSLocationAlwaysAndWhenInUseUsageDescription` — location
  - `NSMicrophoneUsageDescription`, `NSPhotoLibraryUsageDescription`, `NSContactsUsageDescription`, etc.

    - **Missing string** → **immediate crash** when the API is first used.
- **Runtime flow**: iOS prompts **once** on first request; after that the status is fixed until the user changes it in **Settings**. States: `authorized` / `denied` / `restricted` / `notDetermined` (+ iOS 14+ `provisional` / `limited` for some). Via `permission_handler`: check `status`, `.request()`, and on denial guide the user to `openAppSettings()` (you can't re-prompt).
- **Timing**: request **in context** (when the feature is used) with a **pre-prompt rationale** — better acceptance and Apple guidance.
- **Background/always location**: requires additional keys/justification and is scrutinized in review.
- **App Tracking Transparency (ATT)**: tracking/IDFA needs `NSUserTrackingUsageDescription` + the ATT prompt.
- **Wrap in a repository/service**: expose `ensureCamera()` returning a result; centralize + mirror Android.

Memory Representation

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

Compiler Behavior

`Info.plist` is bundled at build; missing usage strings aren't compile errors — they crash at runtime on first request.

Runtime Behavior

First `.request()` shows the OS prompt (with your string); subsequent requests return the fixed status (no re-prompt) until changed in Settings. Using a capability without a declared string → crash.

Flutter Engine Behavior

Permission prompts/results go through iOS via the `FlutterViewController`/plugins (02_swift_plugin_code.md); `permission_handler` manages this.

Dart VM Behavior

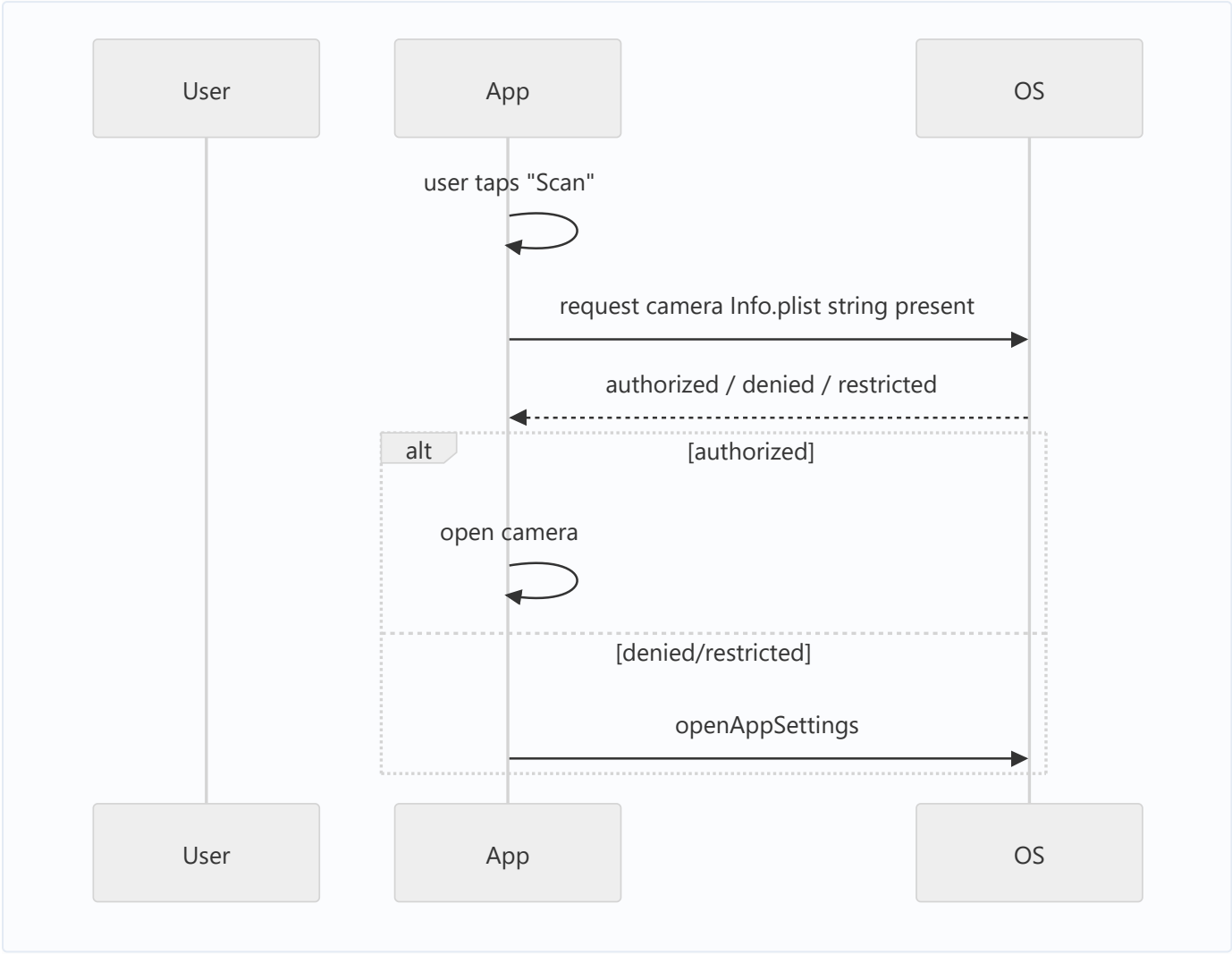
Not applicable.

Examples

```
<!-- ios/Runner/Info.plist — usage strings (REQUIRED per capability) -->
<key>NSCameraUsageDescription</key>
<string>We use the camera to scan receipts.</string>
<key>NSLocationWhenInUseUsageDescription</key>
<string>We use your location to show nearby stores.</string>
<key>NSPhotoLibraryUsageDescription</key>
<string>We access photos so you can attach images.</string>
```

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

// Repository centralizing permission logic (same API as Android — mirror)
class PermissionService {
  Future<bool> ensureCamera() async {
    var status = await Permission.camera.status;
    if (status.isGranted) return true;
    if (status.isPermanentlyDenied || status.isRestricted) {
      await openAppSettings();          // iOS: user must enable in Settings
      return false;
    }
    status = await Permission.camera.request(); // shows the one-time OS prompt
    return status.isGranted;
  }
}
// Usage: if (await permissions.ensureCamera()) openCamera(); else showRationale();
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Missing usage string	Immediate crash on first request	Add the `NS...UsageDescription` key
Vague/generic usage strings	App Store rejection	Specific, honest justifications
Requesting all permissions upfront	Low acceptance / Apple guidance	Request in context with rationale
Not handling denied/restricted	User stuck	Guide to Settings (`openAppSettings`)
Expecting to re-prompt after denial	iOS prompts once	Route to Settings instead
Missing background/ATT keys	Feature broken / rejection	Add required extra keys + prompts

Best Practices

- **Declare a specific usage string** for every sensitive capability (missing → crash; vague → rejection).

- **Request in context** with a pre-prompt rationale; handle **authorized/denied/restricted/notDetermined**; route to `openAppSettings()` on denial (iOS can't re-prompt).
- Add extra keys for **always-location/background/ATT**; justify carefully (review scrutiny).
- **Degrade gracefully** without the permission; **centralize** in a repository/service; **mirror Android** (27 · permissions_and_manifest).

Performance

Negligible; occasional checks/requests. In-context requests improve acceptance (UX/business, not perf).

Advantages / Disadvantages

- + User transparency/consent, App Store compliance, capability access with graceful degradation.
- − Crash-on-missing-string gotcha, one-shot prompt + Settings fallback, extra keys for background/ATT, review scrutiny.

Interview Questions

1. 🟢 **Why do iOS permissions need `Info.plist` usage strings?** — iOS requires a stated reason (shown in the prompt) for sensitive capabilities; missing it crashes the app on first request.
2. 🟢 **What happens if a usage string is missing?** — The app crashes immediately when the capability is first requested.
3. 🟡 **How many times does iOS prompt for a permission?** — Once; afterward the status is fixed until the user changes it in Settings — so guide to `openAppSettings()` on denial.
4. 🟡 **How do you request permissions from Dart?** — Via `permission_handler`: check `status`, `.request()`, handle authorized/denied/restricted, route to Settings on denial.
5. 🟡 **When should you request?** — In context (on feature use) with a pre-prompt rationale — better acceptance and Apple guidance.
6. 🔴 **What extra requirements apply to always-location/tracking?** — Additional Info.plist keys (background/always usage, `NSUserTrackingUsageDescription`) + the ATT prompt, with careful justification for review.
7. 🔴 **How does iOS permission handling compare to Android?** — Both declare + request at runtime; iOS crashes on missing usage strings and prompts once (Settings fallback), while Android has per-request dialogs and granular media perms — centralize both behind one service.

Senior Engineer Tips

- Add usage strings **before** shipping any sensitive feature — a missing one is a guaranteed crash caught late.
- Write **specific, honest** strings (Apple rejects vague ones); pre-prompt with your own rationale before triggering the OS prompt.
- Centralize permissions in a `PermissionService` shared with Android; always provide a no-permission fallback + Settings route.

Architect Perspective

iOS permissions are a privacy/compliance/UX concern: usage strings (mandatory), in-context requests, Settings fallback, and graceful degradation. A centralized, cross-platform `PermissionService` (mirroring Android) with correct Info.plist declarations ensures features work and pass App Store review, integrating with device features and background/ATT requirements (Module 29, 05_ios_integration.md).

Summary

- Declare a specific `NS...UsageDescription` per sensitive capability (missing → crash); iOS prompts once, with a Settings fallback.
- Request in context (rationale), handle all states, route to `openAppSettings()` on denial, degrade gracefully.
- Add background/ATT keys where needed; centralize in a service mirroring Android.

Revision Notes

- `Info.plist`: `NSCameraUsageDescription` / `NSLocationWhenInUse...` / `NSMicrophone...` / `NSPhotoLibrary...` (missing → crash; vague → rejection).
- One-time prompt; states authorized/denied/restricted/notDetermined; denial → `openAppSettings()` (no re-prompt).
- Request in context + rationale; background/always-location/ATT need extra keys/prompts.
- Centralize in `PermissionService`; mirror Android; degrade gracefully.

Practice Questions

1. What happens if you request the camera without a usage string?
2. How do you handle a denied permission on iOS?
3. Why request permissions in context?

Coding Questions

1. Add camera/location usage strings to `Info.plist` and request camera via `permission_handler`.
2. Handle denied/restricted by opening app settings.
3. Build a cross-platform `PermissionService.ensureCamera()`.

Mini Project

iOS permission-gated feature (Flutter + iOS): Add camera + location usage strings to `Info.plist`, build a `PermissionService` requesting camera in-context (rationale), handling authorized/denied/restricted (+ `openAppSettings`), and gating a camera feature with graceful fallback — shared with Android. Acceptance: usage strings present (no crash); one-time prompt handled; Settings fallback; degrades without permission; cross-platform service; runs on device.

iOS Integration (Capabilities, Background Modes, Universal Links, App Store)

> Beyond channels and permissions, real iOS apps enable **capabilities** (push, background modes, associated domains, keychain sharing) in Xcode, run limited work in **background modes**, open via **universal links** (associated domains + AASA file), and satisfy **App Store** requirements — the iOS integration surface that turns a Flutter build into a shippable, deep-linkable, background-capable app.

Introduction

This module capstone covers the iOS-specific integration layer: **capabilities** (entitlements enabled in Xcode's Signing & Capabilities), **background modes** (fetch, processing, remote notifications, audio/location), **universal links** (HTTPS deep links via associated domains + Apple App Site Association), and **App Store** review essentials. It ties together channels (02_swift_plugin_code.md), permissions (04_infoplist_and_permissions.md), and deployment (Module 51).

Why this concept exists

iOS gates powerful features (push, background execution, deep links, keychain) behind **entitlements/capabilities** for security and battery/privacy control. Background execution is deliberately limited (battery); universal links require server-verified association (security); the App Store enforces privacy/quality. You must configure these to ship a full-featured app.

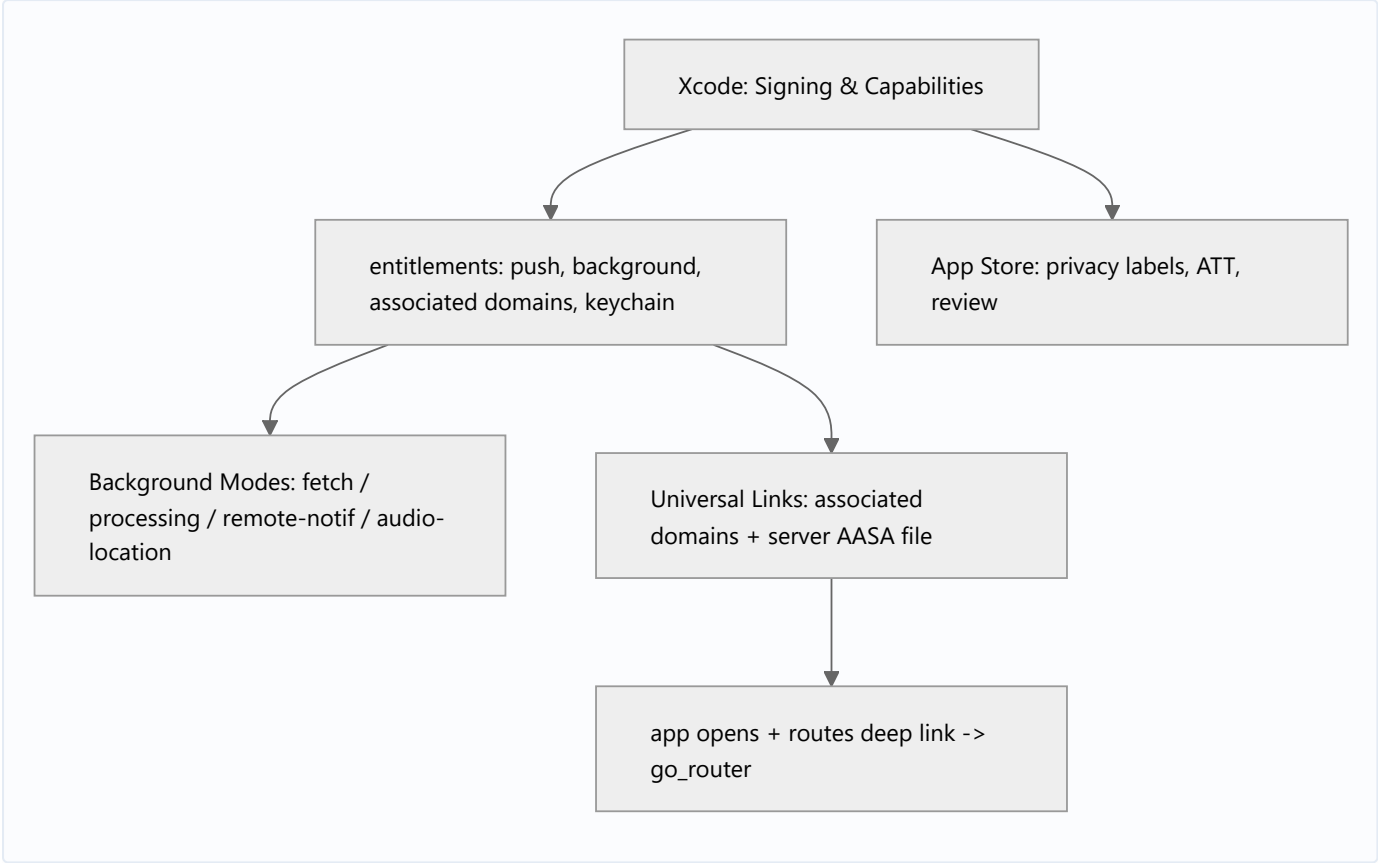
Real-world analogy

Capabilities are **security clearances** you request for the app (each unlocks a room: push, background, keychain). Background modes are **limited after-hours passes** (short, supervised tasks — not free rein). Universal links are a **verified forwarding address** (the domain must vouch for the app via the AASA file). The App Store is **customs/inspection** before release.

Problem Statement

Ship an iOS app that receives push notifications, refreshes data in the background, opens `https://app.example.com/...` links directly into the app (universal links), and passes App Store review. You'll enable capabilities, add background modes, host an AASA file + associated domains, and meet review requirements.

Internal Working



- **Capabilities (entitlements)**: enabled in Xcode **Signing & Capabilities** (writes the `.entitlements` file + updates the provisioning profile): Push Notifications, Background Modes, Associated Domains, Keychain Sharing, Sign in with Apple, App Groups, etc. Each must be in your provisioning profile.
- **Background modes**: iOS strictly limits background work. Modes include **Background fetch** & **Background processing** (`BGTaskScheduler` — opportunistic, OS-scheduled), **Remote notifications** (silent push wakes the app briefly), and continuous **audio/location/VoIP** (only if genuinely used — misuse → rejection). Not general-purpose threads; see Module 33.
- **Universal links**: HTTPS links that open the app directly (no browser bounce). Require: **Associated Domains** capability (`applinks:app.example.com`) + an **`apple-app-site-association` (AASA)** JSON file served at `https://app.example.com/.well-known/apple-app-site-association` (no extension, `application/json` , listing your App ID + paths). iOS verifies it; then links route into the app → handled by `go_router` /deep-link handling (13 · deep_linking). Falls back to Safari if unverified.
- **Push notifications**: APNs (via capability + entitlement); typically through FCM (Module 32).
- **App Store**: **privacy nutrition labels**, **ATT** prompt for tracking (04_infoplist_and_permissions.md), correct usage strings, no private APIs, screenshots/metadata — reviewed before release (Module 51).

Memory Representation

Not applicable; entitlements/config are build-time. Background tasks get short, OS-metered execution windows (memory/time limited).

Compiler Behavior

Capabilities add entitlements embedded/signed into the app; must match the provisioning profile or signing fails.

Runtime Behavior

Background modes give brief OS-scheduled execution; silent push wakes the app momentarily; verified universal links launch/route the app; unverified links open Safari.

Flutter Engine Behavior

The iOS embedder (10 · embedder_and_startup) receives lifecycle/link/notification callbacks in `AppDelegate` and forwards them to Flutter (plugins/channels — 02_swift_plugin_code.md).

Dart VM Behavior

Background execution may run Dart in a background isolate/engine (plugin-dependent) with limited time (Module 33).

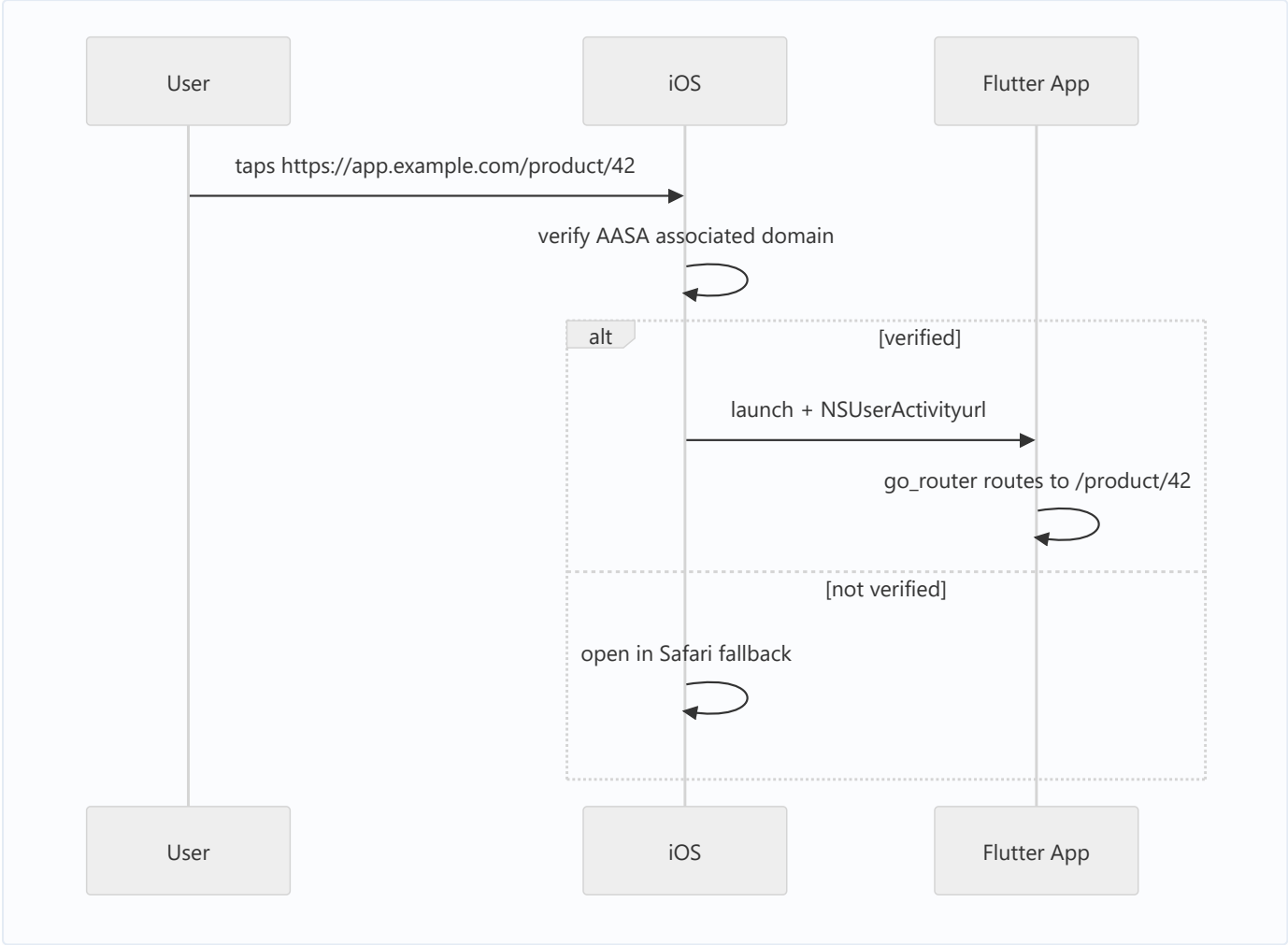
Examples

```
Enable in Xcode → Runner → Signing & Capabilities → "+ Capability":
  • Push Notifications
  • Background Modes → check: Background fetch, Remote notifications, Background processing
  • Associated Domains → add: applinks:app.example.com
  • Keychain Sharing / Sign in with Apple / App Groups (as needed)
```

```
// Served at: https://app.example.com/.well-known/apple-app-site-association
// (no file extension, Content-Type: application/json, HTTPS, no redirects)
{
  "applinks": {
    "apps": [],
    "details": [
      { "appID": "TEAMID.com.example.app", "paths": ["/product/*", "/invite/*"] }
    ]
  }
}
```

```
// AppDelegate: receive a universal link, forward to Flutter (plugins usually handle this)
override func application(
  _ application: UIApplication,
  continue userActivity: NSUserActivity,
  restorationHandler: @escaping ([UIUserActivityRestoring]?) -> Void
) -> Bool {
  if userActivity.activityType == NSUserActivityTypeBrowsingWeb,
     let url = userActivity.webpageURL {
    // forward url to Flutter (e.g., via a plugin / method channel) -> go_router
  }
  return super.application(application, continue: userActivity, restorationHandler: restorationHandler)
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Capability not in provisioning profile	Signing/entitlement failure	Enable in Xcode + regenerate profile
AASA served with wrong type/redirect/extension	Universal link falls back to Safari	Serve JSON at `.well-known`, HTTPS, no redirect, no extension
Claiming background modes you don't use	App Store rejection	Only enable modes you genuinely use
Expecting unlimited background execution	iOS meters it strictly	Use `BGTaskScheduler`/silent push; keep short
Missing privacy labels/ATT/usage strings	Rejection	Complete privacy labels + ATT + usage strings
Wrong App ID/paths in AASA	Links don't route	Match `TEAMID.bundleId` + correct paths

Best Practices

- Enable only **capabilities you use** (each adds entitlement/profile surface); keep provisioning profiles in sync.
- Treat **background execution as scarce**: `BGTaskScheduler`/silent push, short work; only enable continuous modes (audio/location) if truly needed (Module 33).
- Host a **correct AASA** (HTTPS, `.well-known`, right type/App ID/paths) and add **Associated Domains**; route links via `go_router` (13 · deep_linking).
- Complete **privacy labels, ATT, usage strings**; avoid private APIs; test on device before submission (Module 51).
- Handle links/notifications in `AppDelegate` and forward to Flutter behind a repository (02_swift_plugin_code.md).

Performance

Background modes are OS-metered (battery); overusing them is rejected/throttled. Universal links avoid a browser bounce (faster UX). Build-time capabilities have no runtime cost.

Advantages / Disadvantages

- **+** Push, background refresh, seamless deep links, keychain/App Groups; full native iOS integration; better UX + shippability.
- **−** Xcode/entitlement/provisioning complexity, strict background limits, server-side AASA hosting, App Store review scrutiny, Mac required.

Interview Questions

1. 🟢 **What are iOS capabilities?** — Entitlements enabled in Xcode Signing & Capabilities (push, background modes, associated domains, keychain) that unlock gated features and must be in the provisioning profile.
2. 🟢 **What is a universal link?** — An HTTPS link that opens the app directly (no browser bounce), via the Associated Domains capability + a server-hosted AASA file.
3. 🟡 **What does the AASA file do and where does it live?** — It associates the domain with the app (App ID + paths); served at `https://domain/.well-known/apple-app-site-association` as JSON over HTTPS (no redirect/extension).
4. 🟡 **How limited is iOS background execution?** — Very: short OS-scheduled windows via `BGTaskScheduler`/silent push; only enable continuous modes (audio/location) if genuinely used, or risk rejection.
5. 🟡 **How do push notifications work on iOS?** — APNs via the Push Notifications capability/entitlement, commonly through FCM (Module 32).

6. 🔴 **Why might a universal link fall back to Safari?** — Unverified AASA (wrong Content-Type, redirect, path, App ID, or not reachable) → iOS can't confirm the association.
7. 🔴 **What are key App Store review pitfalls?** — Missing privacy labels/usage strings/ATT, unused background modes, private APIs, or mismatched entitlements.

Senior Engineer Tips

- Enable capabilities deliberately and keep provisioning profiles regenerated; entitlement/profile mismatches are a common CI/device blocker.
- Validate the AASA with Apple's tooling and real devices — most universal-link bugs are server-side (type/redirect/paths).
- Never claim background modes you don't use; it's a frequent rejection cause. Forward links/notifications through a single repository shared with Android.

Architect Perspective

iOS integration is where the app meets the platform's security/privacy/battery model: capabilities (least-privilege entitlements), metered background work, verified universal links, and App Store compliance. Designing these cleanly — minimal capabilities, short background tasks, correct AASA, links/notifications behind a cross-platform repository — yields a shippable, deep-linkable, background-capable app and completes the native integration story alongside Android (Module 27, Module 51).

Summary

- Capabilities = Xcode entitlements (push/background/associated domains/keychain), must be in the provisioning profile.
- Background execution is strictly metered (`BGTaskScheduler` /silent push); enable continuous modes only if truly used.
- Universal links = Associated Domains + server AASA (HTTPS `.well-known`, right App ID/paths) → route via `go_router`, else Safari fallback.
- App Store: privacy labels, ATT, usage strings, no private APIs; handle links/notifications in `AppDelegate` behind a repository.

Revision Notes

- Signing & Capabilities → entitlements (push, Background Modes, Associated Domains, Keychain); must match provisioning profile.
- Background: `BGTaskScheduler` /silent push, short/metered; audio/location only if genuinely used (else rejection).

- Universal links: `applinks:domain` + AASA at `https://domain/.well-known/apple-app-site-association` (JSON, HTTPS, no redirect/extension, `TEAMID.bundleId` + paths); unverified → Safari.
- App Store: privacy labels, ATT, usage strings, no private APIs; route links/notifications via `AppDelegate` → Flutter repository.

Practice Questions

1. What must be true for a universal link to open the app instead of Safari?
2. How limited is background execution on iOS, and how do you schedule work?
3. What is an entitlement and how does it relate to the provisioning profile?

Coding Questions

1. Add the Associated Domains capability + host an AASA file for `/product/*`.
2. Handle an incoming universal link in `AppDelegate` and route it in Flutter via `go_router`.
3. Schedule a background refresh with `BGTaskScheduler` (outline) and forward the result to Flutter.

Mini Project

iOS integration slice (module capstone — Flutter + iOS): Ship an iOS integration: enable **Push Notifications + Background Modes + Associated Domains** capabilities; host an **AASA** file and route `https://app.example.com/product/*` universal links into the app via `go_router`; schedule a **background refresh** (`BGTaskScheduler` /silent push); handle links/notifications in `AppDelegate` and forward to Flutter behind a repository (shared with Android); and satisfy **App Store** basics (privacy labels, usage strings, ATT if tracking). Combine with a Swift channel (02_swift_plugin_code.md), a permission-gated feature (04_infoplist_and_permissions.md), and an embedded native view (03_platform_views_ios.md). Acceptance: capabilities enabled + signed; universal link opens + routes in-app (Safari fallback if unverified); background refresh runs (metered); links/notifications forwarded via a cross-platform repository; App Store requirements met; runs on device.