54 · Flutter Desktop

Flutter Practice Notes

Contents

54 · Flutter Desktop

Introduction

This module covers running Flutter on the **desktop** — **Windows, macOS, and Linux**: the **fundamentals** (how desktop targets work, and **when desktop fits**), **desktop UX & windowing** (window management, menu bars, keyboard shortcuts, mouse conventions, multi-window), **native integration** (FFI, platform channels on desktop, full filesystem/OS access), and **packaging & distribution** (MSIX/DMG/AppImage, code signing, stores vs direct download, updates) — tied together in a capstone. It's the sibling target to Flutter Web (Module 53), building on responsive/adaptive UI (Module 24/Module 25), platform channels (Module 26), and deployment (Module 51).

Why this module exists

Flutter Desktop turns one codebase into native Windows/macOS/Linux apps — genuinely useful for **internal tools, cross-platform productivity/creative apps, and companion desktop apps** — but desktop has its **own UX conventions** (menu bars, keyboard-first workflows, resizable windows, right-click), **full OS access** (real filesystem, FFI to native libs), and **fragmented distribution** (three platforms, different packaging/signing/stores). Unlike mobile, there's no single store flow. Knowing the targets, desktop UX, native integration, and per-platform packaging is what makes a real desktop app rather than a phone UI stretched to a window.

Module map

#	File	Topic	Level
1	01_desktop_fundamentals.md	Windows/macOS/Linux targets, how it works, when desktop fits	🔵
2	02_desktop_ux_and_windowing.md	Window management, menus, keyboard shortcuts, mouse, multi-window	🟡
3	03_native_integration_desktop.md	FFI, platform channels, filesystem/OS access on desktop	🔴
4	04_packaging_and_distribution.md	MSIX/DMG/AppImage, signing, stores vs direct, updates	🔴
5	05_desktop_integration.md	Capstone: a production desktop app	🔴

> **Cross-references:** Web (sibling target): Module 53. Responsive/adaptive UI: Module 24/Module 25. Platform channels/native: Module 26/Module 27/Module 28. File handling: Module 34. Deployment/signing: Module 51/50 · build_signing.

Prerequisites

24 Responsive UI, 25 Adaptive UI, 26 Platform Channels, 34 File Handling, 51 Deployment. A machine of the target OS to build/test each platform.

What you'll be able to do after this module

- Explain the desktop targets, how they build, and when desktop is the right choice.
- Implement desktop UX: window management, menu bars, keyboard shortcuts, mouse conventions, multi-window.
- Integrate natively: FFI to native libraries, platform channels, full filesystem/OS access.
- Package + sign + distribute per platform (MSIX/DMG/AppImage; stores vs direct; updates).
- Ship a production desktop app that feels native on Windows/macOS/Linux.

Capstone

Production desktop app: A cross-platform desktop app (e.g., a productivity/tool app) with proper window management + a native menu bar + keyboard shortcuts, mouse/right-click conventions, full filesystem access (native file dialogs), one FFI or platform-channel native integration, and per-platform packaging + signing + a distribution/update plan — with a documented "why desktop fits + platform differences" analysis.

Summary

Flutter Desktop builds native Windows/macOS/Linux apps from one codebase — strong for tools/productivity/companion apps. Success means desktop-idiomatic UX (windowing/menus/keyboard/mouse), full OS/native integration (FFI/channels/filesystem), and per-platform packaging + signing + distribution (no single store) — applied where desktop genuinely fits.

Desktop Fundamentals

> Flutter Desktop compiles your app to a **native executable** for **Windows, macOS, and Linux** — each with a **platform-specific embedder + runner** (Win32/C++, Cocoa/Swift, GTK/C++) hosting the same Flutter engine + your AOT-compiled Dart, rendered via **native Skia/Impeller** (real desktop rendering, not a browser canvas like web). You **must build each platform on its own OS** (Windows for `.exe`, Mac for `.app`, Linux for the binary). Unlike mobile/web, desktop apps get **full OS access** (real filesystem, native libs via FFI, multiple windows) — and desktop **fits** tools/productivity/creative/companion apps, less so where a web app or a truly OS-native app is clearly better.

Introduction

This file establishes the desktop landscape: the three targets, how each builds (embedder/runner + engine), the difference from web/mobile, and **when desktop is the right choice** — the frame for the UX, native-integration, and packaging files.

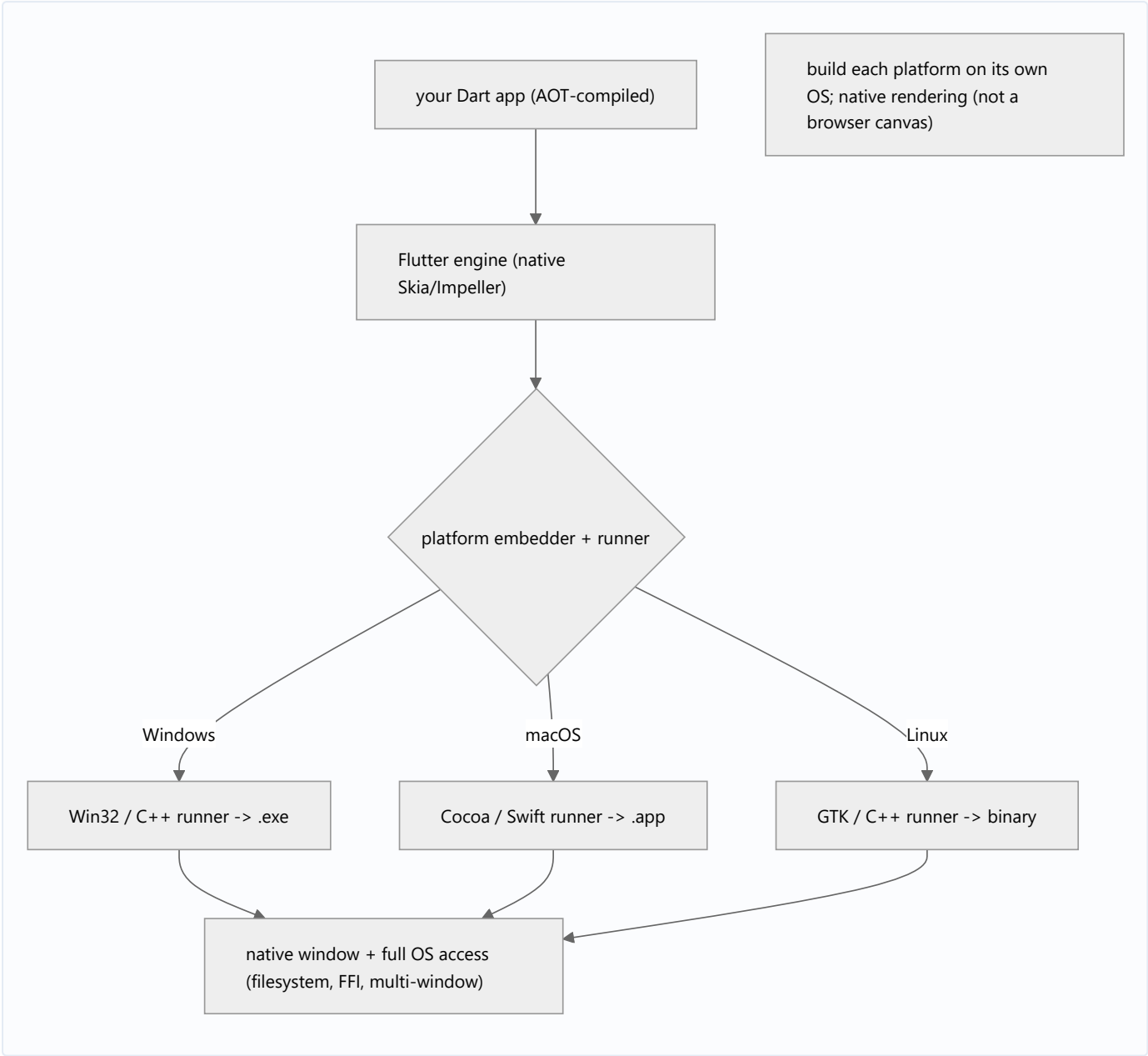
Why this concept exists

Desktop is a first-class Flutter target, but it's **three OSes** with different embedders, build hosts, UX conventions, and distribution — and it differs from mobile (windows/menus/keyboard, full OS access) and web (native rendering, no browser sandbox). Understanding the targets + fit prevents treating desktop like "mobile in a big window" or reaching for it when web/native suits better.

Real-world analogy

Flutter Desktop is like **manufacturing your product in three regional factories** (Windows/macOS/Linux): the **product design is shared** (your Dart app), but each factory has its **own assembly line + local fittings** (embedder/runner) and you must **run production in each region** (build on each OS). Unlike a mall kiosk (mobile app store) or a web pop-up (browser), a desktop app is a **standalone appliance in the user's home** — it can plug into everything (full OS access) but you distribute + service it yourself per region.

Internal Working



- **Three targets + embedders**: Flutter Desktop supports **Windows** (Win32, C++ runner → `.exe`), **macOS** (Cocoa, Swift/Obj-C runner → `.app`), and **Linux** (GTK, C++ runner → native binary). Each has a **platform runner** (in `windows/`, `macos/`, `linux/`) that hosts the **Flutter engine** + your **AOT-compiled Dart**. Enable with `flutter config --enable-<platform>-desktop`; build with `flutter build windows/macos/linux`.
- **Native rendering (unlike web)**: desktop renders via the **real Flutter engine** (Skia/**Impeller**) to a native window — **not a browser canvas** (Module 53). High fidelity/performance like mobile; no download-size/SEO issues.
- **Build-on-target rule**: you can only build a platform on **that OS** — **Windows for `.exe`, macOS for `.app`, Linux for the binary** (like iOS needing a Mac). CI needs **per-OS runners** (04_packaging_and_distribution.md/Module 50).

- **Full OS access (unlike mobile/web)**: desktop apps run **outside a sandbox** (or a looser one — macOS App Store apps are sandboxed) with **full filesystem access** (real paths, not scoped), **native library access via FFI** (03_native_integration_desktop.md), **multiple windows**, system tray, native menus, and OS APIs. Far more capable than a mobile sandbox or browser.
- **What differs from mobile** (design accordingly — 02_desktop_ux_and_windowing.md): **resizable windows** (responsive layout, not fixed phone screen), **menu bars**, **keyboard-first** workflows/shortcuts, **mouse + right-click**, **multi-window**, larger screens/denser UIs, and no touch-first assumptions.
- **When desktop fits (the decision)**:
  - **Great fit**: **cross-platform productivity/creative tools** (editors, IDEs-like tools, design/media apps), **internal/enterprise tools**, **companion desktop apps** to a mobile product, apps needing **full OS/hardware access + performance** on desktop — where one codebase across 3 OSes is a big win.
  - **Consider alternatives**: if it's **web-deliverable** (no install, SEO/reach) → **web** or **PWA** may fit better; if it needs **deep OS-native look/feel or platform-specific APIs** beyond Flutter's reach → a **native** app (WinUI/SwiftUI/GTK) or Electron-style might suit. Flutter Desktop shines for **shared, custom-UI, cross-platform** apps, not for perfectly OS-native chrome.
  - **Trade-off**: **massive code reuse + consistent custom UI across 3 OSes** vs **not pixel-native to each OS's HIG** + **per-platform packaging/signing/distribution overhead**.
- **Maturity note**: desktop is stable/production-capable, but the **plugin ecosystem** is thinner than mobile — some plugins lack desktop support (check, or use FFI/channels). Factor this in.

Memory Representation

Not app state — a **native executable per platform** (runner + engine + AOT Dart + assets) plus platform runner projects (`windows/` / `macos/` / `linux/`). At runtime it's a native process with a native window + full OS resources.

Compiler Behavior

`flutter build <platform>` AOT-compiles Dart + builds the native runner (MSVC/Xcode/GCC-clang) into a platform executable/bundle; you must build on the matching OS. Release is AOT (fast, no VM).

Runtime Behavior

Runs as a **native desktop process** with a native window; full OS access (filesystem/FFI/multi-window); native rendering (Skia/Impeller) at desktop refresh rates. No sandbox restrictions (except macOS App Store sandbox).

Flutter Engine Behavior

The **real engine** renders to a native window via the platform embedder (Win32/Cocoa/GTK) — same widget/render trees as mobile (Module 09); Impeller/Skia backend, not a browser canvas.

Dart VM Behavior

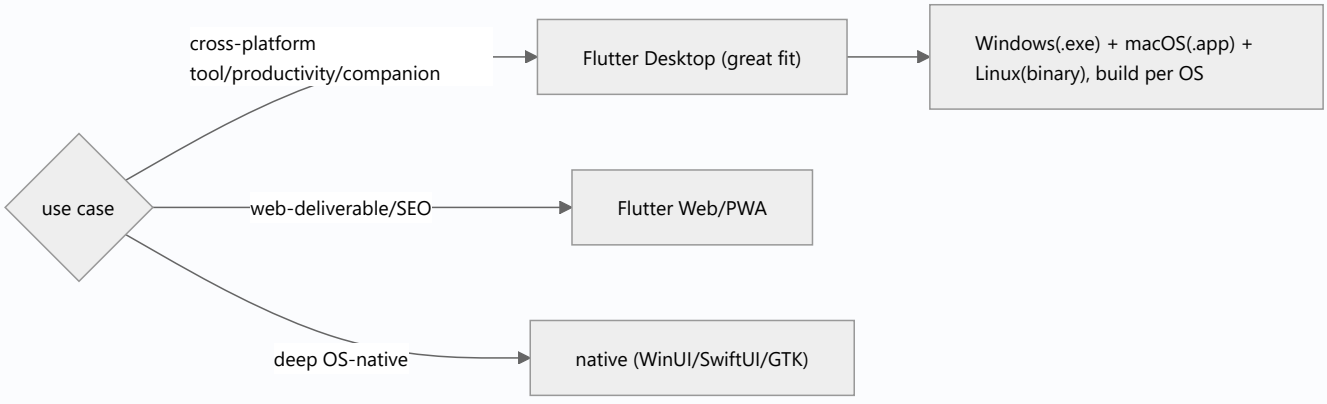
AOT-compiled (like mobile release) — full **isolates** available (unlike web's limited workers), full `dart:io` /filesystem, FFI to native code.

Examples

```
Enable + build (on the matching OS):
  flutter config --enable-windows-desktop   # then: flutter build windows  -> .exe  (on Windows)
  flutter config --enable-macos-desktop     # then: flutter build macos    -> .app  (on macOS)
  flutter config --enable-linux-desktop     # then: flutter build linux    -> binary (on Linux)
  # runners live in windows/ macos/ linux/ ; build each on its own OS (CI needs per-OS runners)
```

```
When Flutter Desktop FITS:
  cross-platform tools/productivity/creative apps, internal/enterprise tools, companion desktop apps,
  apps needing full OS/hardware access + performance, custom shared UI across 3 OSes
When to consider alternatives:
  web-deliverable/no-install/SEO  -> Flutter Web / PWA
  deep OS-native chrome/APIs       -> native (WinUI/SwiftUI/GTK)
  Trade-off: huge code reuse + consistent custom UI  vs  not pixel-native to each HIG + per-platform
packaging/signing
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
Treating desktop like "mobile in a window"	Ignores desktop UX (menus/keyboard/resize)	Design desktop-idiomatic UX (Module 2 of this)
Trying to build all platforms on one OS	You must build on each target OS	Per-OS build hosts / CI runners
Assuming all plugins support desktop	Thinner ecosystem	Check support; FFI/channels for gaps
Using Flutter Desktop for a web-deliverable app	Install overhead vs web reach	Consider Flutter Web/PWA
Expecting pixel-native OS chrome	Flutter draws its own UI	Fits custom-UI apps; adapt conventions
Ignoring per-platform packaging/signing	No single store flow	Plan MSIX/DMG/AppImage + signing (Module 4)
Fixed phone-size layout	Breaks on resizable windows	Responsive layout (Module 24/25)

Best Practices

- **Choose desktop by use case**: great for **cross-platform tools/productivity/creative/companion** apps with shared custom UI; consider **web/PWA** for web-deliverable, **native** for deep OS-native needs.
- Remember desktop is **native-rendered** (fidelity/performance, no web download/SEO issues) but must be **built on each target OS** (per-OS runners/CI); check **plugin desktop support** (use **FFI/channels** for gaps).
- Design **desktop-idiomatic UX** (resizable windows, menus, keyboard/mouse — 02_desktop_ux_and_windowing.md) and leverage **full OS access** (filesystem/FFI/multi-window), not a stretched phone UI.
- Plan **per-platform packaging + signing + distribution** early (no single store — 04_packaging_and_distribution.md); accept the **reuse-vs-per-OS-overhead** trade-off deliberately.

Performance

Native rendering (Skia/Impeller) + AOT + full isolates give **strong desktop performance** (no web download/SEO limits, no mobile sandbox). The concerns are the usual Flutter perf (scoped rebuilds, virtualization — Module 21) plus desktop-scale UIs (large windows/lists). Startup is native-fast.

Advantages / Disadvantages

- **+** One codebase → native Windows/macOS/Linux; native rendering/perf; full OS access (filesystem/FFI/multi-window); consistent custom UI; great for tools/productivity.
- **−** Build-per-OS, thinner plugin ecosystem, not pixel-native to each HIG, per-platform packaging/signing/distribution (no single store), desktop-UX work needed.

Interview Questions

1. 🟢 **What platforms does Flutter Desktop target, and how does each build?** — Windows (Win32/C++ → `.exe`), macOS (Cocoa/Swift → `.app`), Linux (GTK/C++ → binary); each has a platform runner hosting the engine + AOT Dart, and you build on the matching OS.
2. 🟢 **How does desktop rendering differ from web?** — Desktop uses the real Flutter engine (Skia/Impeller) to a native window (high fidelity/perf, no download/SEO); web paints to a browser canvas with a large download.
3. 🟡 **What full-OS capabilities does desktop gain over mobile/web?** — Real filesystem access (not scoped), native libraries via FFI, multiple windows, system tray, native menus, full isolates — outside the mobile/browser sandbox.
4. 🟡 **When is Flutter Desktop the right choice — and when not?** — Right for cross-platform tools/productivity/creative/companion apps with shared custom UI; consider web/PWA for web-deliverable, native for deep OS-native chrome/APIs.
5. 🟡 **Why must you build each platform on its own OS?** — Each uses the platform's native toolchain (MSVC/Xcode/GCC) — like iOS needing a Mac; CI needs per-OS runners.
6. 🔴 **What's the plugin-ecosystem caveat?** — It's thinner than mobile; some plugins lack desktop support — check, and fill gaps with FFI/platform channels.
7. 🔴 **What's the core trade-off of Flutter Desktop?** — Huge code reuse + consistent custom UI across 3 OSes vs not being pixel-native to each HIG + per-platform packaging/signing/distribution overhead.

Senior Engineer Tips

- Decide desktop vs web/native by the use case (shared custom-UI tool vs web reach vs deep OS-native), not by "we have Flutter"; the wrong target ships an install-heavy app where web fit, or a non-native-feeling one where users expected native.
- Set up per-OS CI build hosts and check plugin desktop support early; "we'll build all three on one machine" and "this plugin has no desktop impl" are the two first surprises.
- Design for resizable windows + desktop UX from the start, and plan per-platform packaging/signing — retrofitting menus/keyboard and figuring out MSIX/DMG/AppImage at the end is painful.

Architect Perspective

Flutter Desktop extends the single-codebase promise to native Windows/macOS/Linux with full OS access and native rendering — a strong fit for cross-platform tools/productivity/companion apps. The architect's job is the **fit decision** (desktop vs web/native), designing for **desktop UX + full OS integration**, and planning the **fragmented per-platform build/package/sign/distribute** reality (no single store). Get those right and desktop is a big reuse win; treat it as "mobile in a window" or ignore packaging and it disappoints (02_desktop_ux_and_windowing.md, 04_packaging_and_distribution.md, Module 53).

Summary

- Flutter Desktop builds native Windows (`.exe`)/macOS (`.app`)/Linux (binary) apps via per-OS embedders/runners + the real engine (Skia/Impeller) — native rendering, not a browser canvas; build each on its own OS.
- Gains full OS access (filesystem/FFI/multi-window/isolates) beyond mobile/web sandboxes; requires desktop-idiomatic UX + per-platform packaging/signing/distribution.
- Fits cross-platform tools/productivity/creative/companion apps; consider web/PWA (web-deliverable) or native (deep OS-native); mind thinner plugin support.

Revision Notes

- Targets: Windows (Win32/C++, `.exe`), macOS (Cocoa/Swift, `.app`), Linux (GTK/C++, binary); runners in `windows/` / `macos/` / `linux/` host engine + AOT Dart; `flutter config --enable-<platform>-desktop` + `flutter build <platform>` ; build on matching OS (CI per-OS runners).
- Native rendering (Skia/Impeller, not browser canvas); full OS access (real filesystem, FFI, multi-window, system tray, native menus, full isolates) vs mobile/web sandbox (macOS App Store sandboxed).
- Fits cross-platform tools/productivity/creative/companion/enterprise; web/PWA for web-deliverable, native for deep OS-native. Trade-off: reuse + consistent custom UI vs not-pixel-native HIG + per-platform packaging/signing. Thinner plugin ecosystem (FFI/channels for gaps).

Practice Questions

1. How do the three desktop targets build, and why per-OS?
2. What OS capabilities does desktop gain over mobile/web?
3. When would you choose Flutter Desktop vs web vs native?

Coding Questions

1. Enable + build a desktop target on its OS and run it.
2. Identify a plugin lacking desktop support and the FFI/channel fallback.
3. Decide + justify desktop vs web/native for two scenarios.

Mini Project

Desktop target + fit analysis (Flutter Desktop): Enable and build one desktop target (on its OS), confirm native rendering + full OS access (e.g., real filesystem), and write a fit-analysis doc: which desktop targets, why Flutter Desktop fits (or web/native would be better) for two scenarios, the build-per-OS + plugin-ecosystem caveats, and the reuse-vs-per-OS-overhead trade-off. Acceptance: a desktop target builds/runs (native, full OS access); fit decision justified per scenario (vs web/native); build-per-OS + plugin caveats noted; trade-off documented; desktop framed as native (not web canvas) and not "mobile in a window."

Desktop UX & Windowing

> Desktop users expect **desktop conventions**, not a phone UI in a window: a **resizable window** (responsive layout + a sensible **min size** + remembered size/position), a **native menu bar** (`PlatformMenuBar` — File/Edit/View… on macOS the system bar), **keyboard-first workflows** (shortcuts via `Shortcuts` / `Actions` , full focus traversal, Tab/Enter/Esc), **mouse conventions** (hover states, cursors, **right-click context menus**, scroll wheel, visible scrollbars), and often **multi-window/system-tray** behavior (via `window_manager` / `tray_manager`). Getting these right — plus each OS's HIG differences (macOS traffic lights + top menu vs Windows/Linux) — is what makes the app feel native rather than ported.

Introduction

This file covers desktop-idiomatic UX: window management (size/position/min/multi-window), native menus, keyboard shortcuts + focus, mouse/right-click conventions, and per-OS HIG differences. It applies responsive/adaptive UI (Module 24/Module 25) and web-UX overlap (53 · responsive_and_web_ux) to the desktop context.

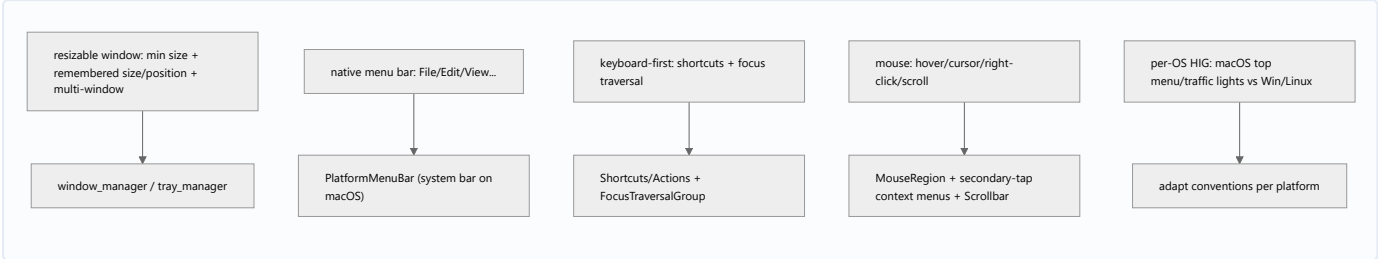
Why this concept exists

Desktop has decades of UX conventions (menu bars, keyboard shortcuts, resizable/multi windows, right-click) users expect and rely on — and mobile-first Flutter code provides none of them. Without desktop UX, the app feels foreign: a fixed phone-sized window, no menus, no shortcuts, no right-click. These patterns bridge the app to the desktop platform.

Real-world analogy

It's **furnishing an office vs a phone booth**: a desk you can **rearrange to fit the room** (resizable window + min size), a **filing/menu system on the wall** (menu bar), **keyboard shortcuts like a power user's hotkeys**, a **mouse with a right-click menu**, and maybe **multiple desks/monitors** (multi-window). A phone-booth setup (tiny fixed screen, touch-only, no menus/shortcuts) dropped into an office is obviously wrong to anyone who works at a desk.

Internal Working



- **Window management**:
  - **Resizable** windows → **responsive layout** (breakpoints + max content width + desktop patterns like side rails/master-detail — Module 24); set a **minimum window size** so the UI never breaks.
  - **Remember + restore** window **size/position** (and maximized state) across launches; set **initial size/title**. Use `window_manager` for size/position/min/max/fullscreen/always-on-top/frameless/custom title bar.
  - **Multi-window** (opening additional windows) — supported via plugins/newer Flutter multi-window APIs; common for tools (inspectors, detached panels). **System tray** via `tray_manager` (background apps, quick actions).
- **Native menu bar**: desktop apps expect a **menu bar** (File/Edit/View/Help...). Flutter's `PlatformMenuBar` renders a **native menu bar** — on **macOS the system top menu bar** (required convention), on Windows/Linux an in-window menu. Wire menu items to actions/shortcuts. (Context menus + app menu also matter.)
- **Keyboard-first workflows** (desktop essential):
  - **Shortcuts** via `Shortcuts` + `Actions` (or `CallbackShortcuts`): Ctrl/**Cmd**+S/C/V/Z, Esc, F-keys, arrows — and **map them in the menu bar** (users discover shortcuts via menus). Respect **platform modifier** (Cmd on macOS, Ctrl on Windows/Linux).
  - **Focus traversal**: full **Tab/Shift+Tab** order (`FocusTraversalGroup`, correct focus), Enter/Space activation, Esc to cancel — desktop users navigate by keyboard.
- **Mouse conventions**:
  - **Hover states** + **cursors** (`MouseRegion` — pointer over clickables, text cursor over text) — desktop expects hover feedback.
  - **Right-click context menus** (`GestureDetector.onSecondaryTap` / `ContextMenuController` / menu builders) — a core desktop interaction.
  - **Scroll wheel/trackpad** + **visible scrollbars** (`Scrollbar`), **double-click**, drag-and-drop where relevant.
- **Text + selection**: selectable text (`SelectionArea` / `SelectableText`), standard copy/paste/cut, and native-feeling text fields.
- **Per-OS HIG differences (adapt)**:

    - **macOS**: system **top menu bar**, **traffic-light** window controls, Cmd-based shortcuts, distinct look/feel; consider `macos_ui` for native-styled widgets.
    - **Windows**: in-window menu, Ctrl shortcuts, title bar controls; `fluent_ui` for Fluent-styled widgets.
    - **Linux**: varies by desktop environment (GTK); in-window menu, Ctrl shortcuts; `yaru` (GNOME/Ubuntu) styling.
    - **Adapt conventions per platform** (shortcut modifiers, menu placement, window controls) — one codebase, platform-aware behavior/styling.
- **Density + sizing**: desktop UIs are often **denser** (more info, smaller touch targets acceptable) and **mouse-precise** — adapt from mobile's touch-sized targets (Module 25).

Memory Representation

Not app state beyond UI: window size/position (persisted to storage/prefs), focus/hover state, shortcut→action mappings, menu structure, multi-window handles. `window_manager` / `tray_manager` hold native window/tray state.

Compiler Behavior

Not applicable (standard widgets/plugins). `PlatformMenuBar` / `Shortcuts` / `Actions` / `MouseRegion` are cross-platform widgets; platform-conditional code (`Platform.isMacOS`) branches conventions.

Runtime Behavior

Window resizes → responsive rebuilds (min-size enforced); size/position restored on launch; native menu + shortcuts fire actions; hover/cursor/right-click respond to mouse; multi-window/tray managed via plugins.

Flutter Engine Behavior

The desktop embedder reports mouse (hover/wheel/secondary-click) + keyboard events and hosts the native window; `PlatformMenuBar` bridges to the OS menu (native macOS menu bar).

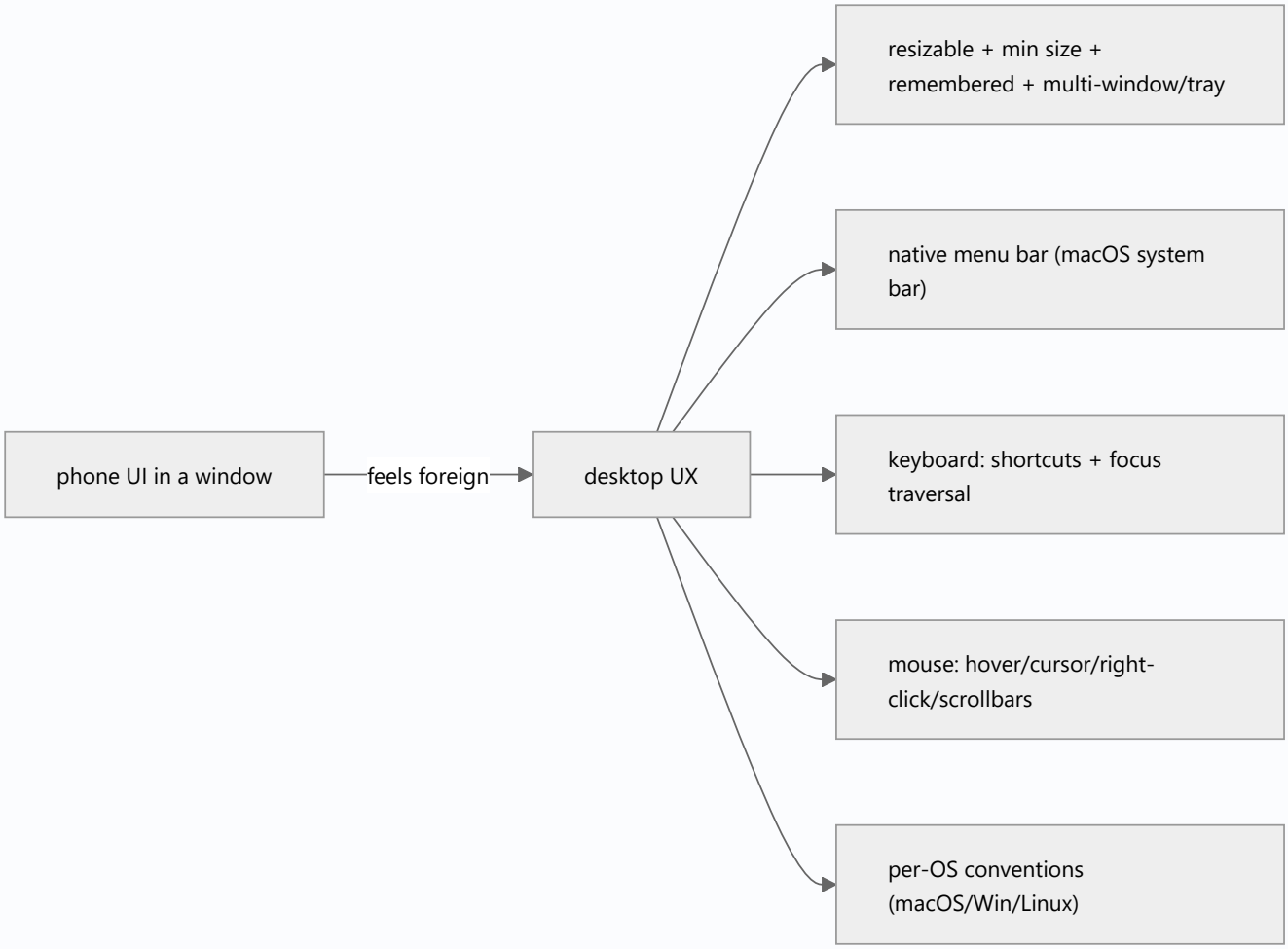
Dart VM Behavior

Not applicable (AOT native); window/input handled by framework + plugins + embedder.

Examples

```
// Window: min size + remembered size/position (window_manager)
Future<void> setupWindow() async {
  await windowManager.ensureInitialized();
  await windowManager.setMinimumSize(const Size(800, 600));   // don't break below this
  await windowManager.setTitle('My Tool');
  // restore saved size/position from prefs; save on resize/move
}

// Native menu bar + keyboard shortcut (platform-aware modifier)
PlatformMenuBar(menus: [
  PlatformMenu(label: 'File', menus: [
    PlatformMenuItem(label: 'Save',
      shortcut: SingleActivator(LogicalKeyboardKey.keyS,
        meta: Platform.isMacOS, control: !Platform.isMacOS),  // Cmd on mac, Ctrl elsewhere
      onSelected: save),
  ]),
], child: appBody);

// Mouse: hover cursor + right-click context menu + visible scrollbar
MouseRegion(cursor: SystemMouseCursors.click, child:
  GestureDetector(onSecondaryTapDown: (d) => showContextMenu(d.globalPosition), child: item));
Scrollbar(child: ListView.builder(/* ... */));

// Keyboard focus traversal + Esc/Enter (Shortcuts/Actions)
Shortcuts(shortcuts: {const SingleActivator(LogicalKeyboardKey.escape): const CancelIntent()},
  child: Actions(actions: {CancelIntent: CallbackAction(onInvoke: (_) => cancel())}, child: form));
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
Fixed phone-size window	Breaks on resize; not desktop	Responsive layout + min size + remembered size
No menu bar	Un-desktop, no shortcut discovery	`PlatformMenuBar` (system bar on macOS)
No keyboard shortcuts/focus	Power users blocked	`Shortcuts` / `Actions` + focus traversal
Same modifier on all OSes	Wrong (Cmd vs Ctrl)	Platform-aware modifiers
No right-click context menu	Missing core desktop interaction	Secondary-tap context menus
No hover/cursor/scrollbars	Feels un-native	`MouseRegion` + `Scrollbar`
Unselectable text	Basic expectation broken	`SelectionArea` / `SelectableText`
Ignoring per-OS HIG	Feels foreign on each	Adapt conventions (menus/controls/styling)

Best Practices

- Support **resizable windows** (responsive layout + **min size** + **remembered size/position**, `window_manager`) and **multi-window/system tray** where the app needs it; never a fixed phone-sized window.
- Provide a **native menu bar** (`PlatformMenuBar` — system bar on macOS) with menu items **mapped to keyboard shortcuts** (platform-aware modifiers: **Cmd**/Ctrl), and full **focus traversal** (Tab/Enter/Esc).
- Implement **mouse conventions** (hover states/cursors, **right-click context menus**, scroll wheel, visible **scrollbars**) and **selectable text**.
- **Adapt per-OS HIG** (menu placement/window controls/shortcuts/styling — `macos_ui` / `fluent_ui` / `yaru`) and **density** for mouse-precise desktop UIs.

Performance

Not primarily perf; responsive rebuilds on resize should be scoped/efficient, and large lists virtualized (Module 21). The concern is **UX fidelity** to desktop conventions. Menus/shortcuts/hover add negligible cost.

Advantages / Disadvantages

- **+** Native desktop feel (windows/menus/keyboard/mouse), power-user efficiency (shortcuts/keyboard), platform-appropriate conventions, one codebase adapting per OS.
- **−** Real work beyond mobile (windowing/menus/shortcuts/right-click/per-OS HIG), plugin reliance (`window_manager` / `tray_manager`), per-OS adaptation + testing.

Interview Questions

1. 🟢 **Why doesn't a mobile UI work on desktop?** — Desktop has resizable windows, menu bars, keyboard-first workflows, mouse/right-click, and multi-window — a fixed phone UI provides none and feels foreign.
2. 🟢 **How do you add a native menu bar in Flutter?** — `PlatformMenuBar` with `PlatformMenu` / `PlatformMenuItem` — rendering the system top menu bar on macOS and an in-window menu on Windows/Linux, wired to actions/shortcuts.
3. 🟡 **How do keyboard shortcuts work, and what's the platform gotcha?** — Via `Shortcuts` / `Actions` (or `CallbackShortcuts`), mapped in the menu bar; use platform-aware modifiers (Cmd on macOS, Ctrl on Windows/Linux).
4. 🟡 **What window-management behaviors do desktop apps need?** — Resizable + min size + remembered size/position (via `window_manager`), possibly multi-window + system tray — with responsive layout inside.

5. 🟡 **What mouse conventions must you support?** — Hover states + cursors, right-click context menus, scroll wheel, visible scrollbars, double-click/drag where relevant.
6. 🔴 **How do you adapt to per-OS HIG?** — Branch conventions (menu placement/window controls/shortcut modifiers) and optionally use OS-styled widget packages (`macos_ui` / `fluent_ui` / `yaru`) — one codebase, platform-aware.
7. 🔴 **Why enforce a minimum window size + remember position?** — To keep the responsive UI from breaking when resized small, and to respect the desktop expectation that windows restore where/how you left them.

Senior Engineer Tips

- Add a native menu bar with shortcut-annotated items + full keyboard traversal early; desktop power users live in menus and shortcuts, and it's the clearest "this is a real desktop app" signal.
- Enforce a min window size, remember size/position, and design responsively; a fixed phone-sized or breaking-on-resize window is the instant "ported mobile app" tell.
- Use platform-aware modifiers (Cmd vs Ctrl) and adapt per-OS conventions/styling; the same Ctrl+S on macOS or a Windows-style menu on macOS feels immediately wrong to users of that OS.

Architect Perspective

Desktop UX & windowing is what makes a Flutter Desktop app belong on the desktop: resizable windows + responsive layout, native menus, keyboard-first workflows, mouse/right-click conventions, and per-OS HIG adaptation. It's the responsive/adaptive discipline extended to the desktop's window+menu+keyboard+mouse context — and combined with fit/native-integration/packaging, it separates a genuine desktop product from a phone UI stretched to a window (Module 24, Module 25, 03_native_integration_desktop.md).

Summary

- Desktop UX = resizable windows (responsive + min size + remembered position + multi-window/tray via `window_manager` / `tray_manager`), a native menu bar (`PlatformMenuBar`, system bar on macOS), keyboard-first workflows (`Shortcuts` / `Actions` + focus traversal), and mouse conventions (hover/cursor/right-click/scrollbars).
- Adapt per-OS HIG (Cmd vs Ctrl, menu placement/controls, styling via `macos_ui` / `fluent_ui` / `yaru`) + desktop density; selectable text.
- Not a phone UI in a window — bridge to desktop conventions.

Revision Notes

- Window: responsive layout + min size + remembered size/position (+ maximized) via `window_manager`; multi-window + system tray (`tray_manager`); initial size/title.
- Menu bar: `PlatformMenuBar` / `PlatformMenu` / `PlatformMenuItem` (native; system top bar on macOS) mapped to shortcuts. Keyboard: `Shortcuts` / `Actions` / `CallbackShortcuts` (platform modifier: Cmd macOS / Ctrl Win-Linux), focus traversal (`FocusTraversalGroup`, Tab/Enter/Esc).
- Mouse: hover/cursor (`MouseRegion`), right-click context menus (`onSecondaryTap` / `ContextMenuController`), scroll wheel, `Scrollbar`, double-click/drag. Text: `SelectionArea` / `SelectableText`. Per-OS HIG: adapt menus/controls/shortcuts/styling (`macos_ui` / `fluent_ui` / `yaru`) + desktop density.

Practice Questions

1. What desktop conventions must you add beyond a mobile UI?
2. How do menus + keyboard shortcuts + focus work, and the platform-modifier gotcha?
3. How do you adapt UX per OS (macOS vs Windows vs Linux)?

Coding Questions

1. Set a min window size + remember size/position with `window_manager`.
2. Build a `PlatformMenuBar` with shortcut-annotated items (platform-aware modifiers).
3. Add hover cursor + a right-click context menu + a visible scrollbar.

Mini Project

Desktop-idiomatic UX (Flutter Desktop): Make an app feel native on desktop: window management (responsive layout + min size + remembered size/position via `window_manager`), a native menu bar (`PlatformMenuBar` File/Edit/View with shortcut-annotated items, platform-aware Cmd/Ctrl), keyboard focus traversal + Esc/Enter, mouse conventions (hover/cursor + right-click context menu + visible scrollbars), selectable text, and per-OS adaptation (modifiers/menu placement/optional OS-styled widgets). Acceptance: resizable window with min size + remembered position; native menu bar + shortcuts (correct modifiers) discoverable via menus; keyboard traversal + right-click menu + hover/cursor/scrollbars; selectable text; per-OS conventions adapted; not a fixed phone UI.

Native Integration on Desktop

> Desktop gives your Flutter app **full OS power** and three ways to reach it: `dart:io` **/ plugins** for ordinary OS access (**real, unrestricted filesystem** — actual paths, not scoped storage — plus native file dialogs via `file_selector` / `file_picker`), **platform channels** to call **native code in the platform runner** (Win32/C++, Cocoa/Swift, GTK/C++) when a plugin doesn't exist, and **FFI (`dart:ffi`)** to call **C/C++ native libraries directly** (no channel round-trip) — ideal for reusing existing native libs or high-performance native work. Desktop's native surface is far larger than mobile's; use the lightest tool that fits (io/plugin → FFI → channel).

Introduction

This file covers desktop native integration: unrestricted filesystem + native dialogs, platform channels on desktop, and FFI to native libraries — and how to choose among them. It applies platform-channel concepts (Module 26) and file handling (Module 34) to the desktop's full-OS context.

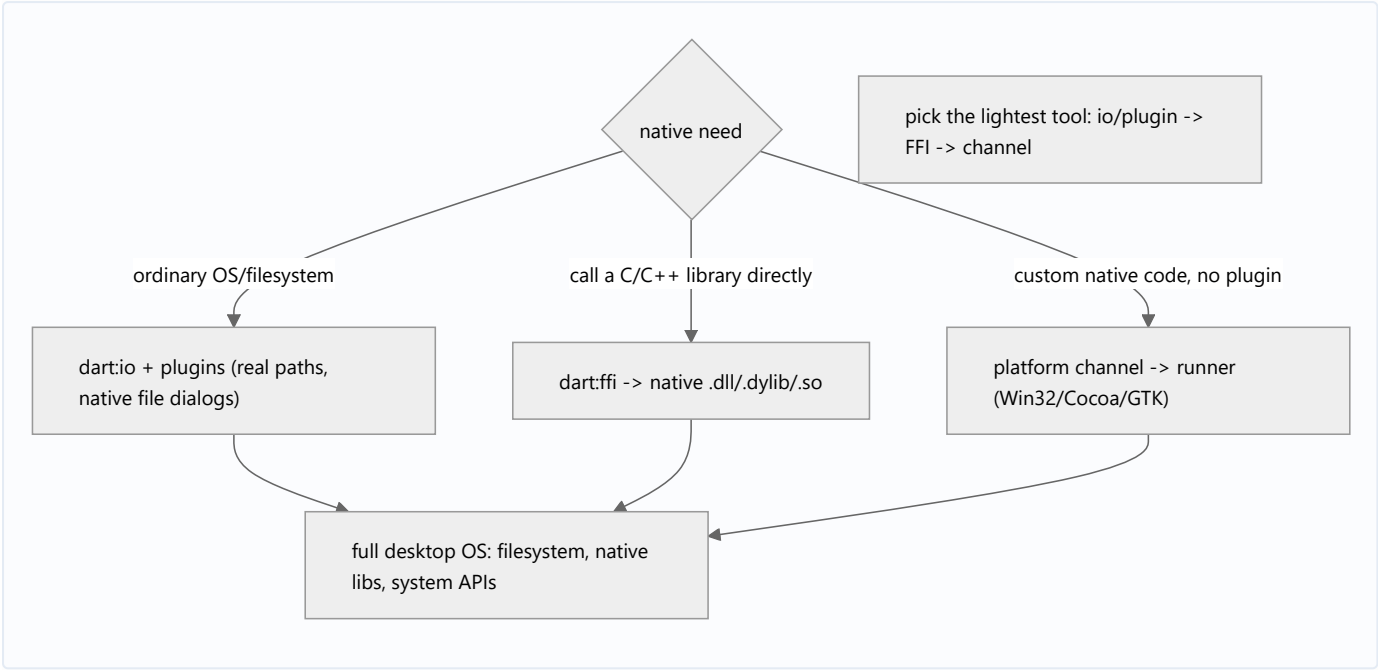
Why this concept exists

Desktop apps routinely need real OS access — read/write anywhere the user chooses, call system libraries, integrate with native tooling — well beyond a mobile sandbox or browser. Flutter provides layered mechanisms (`dart:io` /plugins, channels, FFI) so you can access the OS at the right level. Knowing which to use avoids over-engineering (a channel where `dart:io` suffices) or under-reaching (missing FFI for a native lib).

Real-world analogy

Desktop native access is like having **full keys to the building** (vs a mobile guest pass): most rooms you enter directly (`dart:io` /plugins — real filesystem), for a **specialized machine you already own** you plug straight into it (**FFI** to a C/C++ library), and for **custom on-site work** you hire a local contractor who speaks the building's language (**platform channel** to native runner code). You use the **simplest access that gets the job done**, not a contractor for opening a door.

Internal Working



- **Full filesystem access (the big desktop win)**:
  - Desktop has **real, unrestricted filesystem** via `dart:io` (`File` / `Directory` at actual OS paths) — **not** mobile's scoped/sandboxed storage. Read/write user-chosen locations (Documents, arbitrary paths).
  - **Native file dialogs**: use `file_selector` / `file_picker` for **open/save dialogs** returning real paths; `path_provider` for standard dirs. Users pick files/folders anywhere (a core desktop expectation).
  - (macOS **App Store** builds are **sandboxed** with entitlements — plan file access accordingly; direct-download macOS + Windows/Linux are unsandboxed.)
- **Platform channels on desktop** (Module 26):
  - Same `MethodChannel` / `EventChannel` mechanism as mobile, but the native side is the **desktop runner**: **Win32/C++** (`windows/`), **Cocoa/Swift-ObjC** (`macos/`), **GTK/C++** (`linux/`).
  - Use when you need **custom native code** the platform provides but no plugin exposes (e.g., a Windows registry call, a macOS API). You implement the handler in each platform's runner.
- **FFI (`dart:ffi`) — direct native calls**:
  - Call **C/C++ (and C-ABI) libraries directly** from Dart — no channel serialization/round-trip. Load a `.dll` / `.dylib` / `.so`, bind functions/structs, call them synchronously.
  - **Ideal for**: reusing an **existing native library** (image codecs, crypto, hardware SDKs, computation), **high-performance** native work, or wrapping a C API. `ffigen` generates bindings from C headers.
  - Runs on the **calling thread** (blocking) — offload heavy/long FFI calls to an **isolate** (desktop has full isolates) to avoid jank. Manage native memory (allocate/free) carefully —

FFI is unsafe if mishandled.
    - FFI works on mobile too, but is **especially common on desktop** for reusing native libs.
- **Choosing the mechanism** (lightest that fits):
    - `dart:io` **/existing plugin** → ordinary OS/filesystem needs (first choice).
    - **FFI** → calling a **C/C++ library** directly (reuse/perf) — no per-platform native code to write.
    - **Platform channel** → **custom platform code** with **no library/plugin** and needing OS-specific APIs — most work (write native per OS).
- **System integration** (via plugins/channels): system tray, notifications, clipboard, global shortcuts, single-instance, launch-at-startup, deep links — many via community desktop plugins (check support; FFI/channel for gaps — 01_desktop_fundamentals.md).
- **Isolates for heavy native work**: desktop has **full isolates** — run heavy FFI/native computation off the UI isolate (Module 02).

Memory Representation

`dart:io` uses OS file handles at real paths. FFI holds pointers to native memory (you allocate/free) + loaded library handles — mismanagement leaks/crashes. Channel calls marshal data to the runner. Desktop's full OS resources are available (no sandbox except macOS App Store).

Compiler Behavior

FFI binds against native symbols (via `ffigen`/manual); native libs must be **bundled/linked** for the target. Channels compile Dart↔native (runner) code per platform. `dart:io` is fully available (unlike web).

Runtime Behavior

`dart:io` reads/writes real paths; native dialogs return real paths; FFI calls execute native code synchronously on the calling thread (isolate for heavy); channels round-trip to the runner. Full OS access at runtime.

Flutter Engine Behavior

The desktop embedder hosts the runner (where channel handlers live) + the engine; FFI bypasses the engine (direct Dart↔native). Native windows/menus/tray integrate via embedder/plugins.

Dart VM Behavior

AOT with **full isolates + FFI +** `dart:io` (unlike web). Heavy FFI/native work → isolate to keep the UI isolate responsive; FFI is synchronous on its thread.

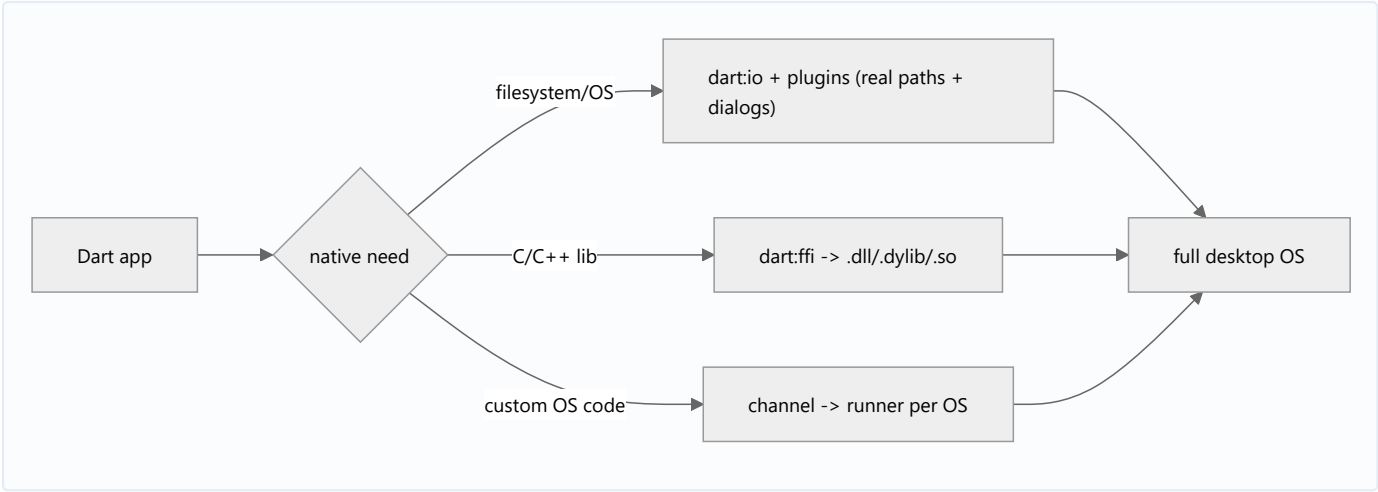
Examples

```
// Full filesystem — real paths + native open/save dialogs (desktop)
import 'dart:io';
import 'package:file_selector/file_selector.dart';

Future<void> saveReport(String content) async {
  final path = await getSaveLocation(suggestedName: 'report.txt');   // native save dialog
  if (path != null) await File(path.path).writeAsString(content);    // real path, unrestricted
}
final file = await openFile(acceptedTypeGroups: [XTypeGroup(extensions: ['csv'])]); // native open
```

```
// FFI — call a C library directly (e.g., a native computation lib)
import 'dart:ffi';
import 'package:ffi/ffi.dart';

typedef _NativeAdd = Int32 Function(Int32 a, Int32 b);
typedef _DartAdd = int Function(int a, int b);
final _lib = DynamicLibrary.open(Platform.isWindows ? 'native.dll'
    : Platform.isMacOS ? 'libnative.dylib' : 'libnative.so');
final int Function(int, int) nativeAdd =
    _lib.lookupFunction<_NativeAdd, _DartAdd>('add');
// Heavy FFI work -> run in an isolate (Isolate.run/compute) to avoid blocking the UI.
```

```
Choose the mechanism (lightest that fits):
  ordinary OS/filesystem  -> dart:io + plugins (file_selector/path_provider)   [first choice]
  reuse a C/C++ library / high-perf native  -> FFI (dart:ffi, ffigen)          [no per-OS native code]
  custom OS API, no plugin  -> platform channel -> runner (Win32/Cocoa/GTK)    [write native per OS]
  # heavy native work -> offload to an isolate (desktop has full isolates)
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
Using scoped-storage mobile patterns	Desktop has real filesystem	Use `dart:io` real paths + native dialogs
Writing a channel where `dart:io` /plugin suffices	Over-engineering	Use `dart:io` /existing plugin first
Ignoring FFI for a native C/C++ lib	Reinventing / slow round-trips	FFI to call the library directly
Blocking the UI on heavy FFI/native	Jank/freeze	Offload to an isolate
Mismanaging FFI native memory	Leaks/crashes (FFI is unsafe)	Allocate/free carefully; wrap safely
Assuming a plugin supports desktop	Thinner ecosystem	Check; FFI/channel fallback
Forgetting macOS App Store sandbox	File access blocked	Entitlements / plan sandbox
Not bundling native libs	FFI load fails	Bundle/link `.dll` / `.dylib` / `.so` for the target

Best Practices

- Use the **lightest mechanism that fits**: `dart:io` **/plugins** for ordinary OS/filesystem (real paths + **native file dialogs**), **FFI** to call **C/C++ libraries** directly (reuse/perf, no per-OS native code), **platform channels** only for **custom OS code with no plugin** (write native per runner).
- Leverage **full, unrestricted filesystem** (real paths, not scoped) — but handle the **macOS App Store sandbox** (entitlements) when applicable.
- **Offload heavy FFI/native work to isolates** (desktop has full isolates) and **manage FFI native memory carefully** (allocate/free; FFI is unsafe if mishandled); **bundle native libs** for each target.

- **Check plugin desktop support**; fall back to **FFI/channels** for gaps; use `ffigen` to generate FFI bindings.

Performance

FFI is **fast** (direct native calls, no serialization) — ideal for heavy computation/native libs — but **synchronous/blocking** on its thread, so isolate heavy calls. `dart:io` is native-fast. Channels have marshaling overhead (fine for occasional calls). Desktop's full isolates keep the UI responsive during native work (Module 21).

Advantages / Disadvantages

- **+** Full OS power (real filesystem/native dialogs), direct native-library reuse + performance (FFI), custom OS code (channels), full isolates — far beyond mobile/web.
- **−** Thinner plugin ecosystem (more FFI/channel work), FFI is unsafe (memory/threading discipline), per-OS native code for channels, macOS sandbox nuance, must bundle native libs.

Interview Questions

1. 🟢 **What are the three ways to access native functionality on desktop?** — `dart:io`/plugins (ordinary OS/filesystem + native dialogs), FFI (call C/C++ libraries directly), and platform channels (custom native runner code) — pick the lightest that fits.
2. 🟢 **How does desktop filesystem access differ from mobile?** — Desktop has real, unrestricted filesystem (actual paths via `dart:io` + native open/save dialogs), not mobile's scoped/sandboxed storage (macOS App Store builds are the sandboxed exception).
3. 🟡 **When do you use FFI vs a platform channel?** — FFI to call an existing C/C++ library directly (reuse/high perf, no per-OS native code, no round-trip); a channel for custom OS-specific code with no library/plugin (write native per runner).
4. 🟡 **What are FFI's risks and how do you handle them?** — It's synchronous/blocking (offload heavy calls to an isolate) and unsafe with native memory (allocate/free carefully; wrap safely); bundle the native lib for each target.
5. 🟡 **Why prefer `dart:io`/plugins first?** — They're the simplest for ordinary OS/filesystem needs; a channel/FFI there is over-engineering.
6. 🔴 **How do platform channels differ on desktop vs mobile?** — Same mechanism, but the native side is the desktop runner (Win32/C++, Cocoa/Swift, GTK/C++) — you implement handlers per OS.
7. 🔴 **How do you keep the UI responsive during heavy native work?** — Offload to an isolate (desktop has full isolates) since FFI/native calls can block the calling thread.

Senior Engineer Tips

- Reach for FFI when you have an existing C/C++ library or need real native performance; it avoids per-OS channel code and channel round-trips — but isolate heavy calls and manage native memory rigorously, because FFI mistakes crash the app.
- Use `dart:io` + native file dialogs (`file_selector`) for the common desktop case (real filesystem); porting mobile scoped-storage patterns to desktop is both unnecessary and worse UX.
- Check plugin desktop support early and plan the FFI/channel fallback; the thinner desktop ecosystem is where "there's no plugin for that" surprises happen.

Architect Perspective

Native integration is where desktop's full-OS power pays off: layered mechanisms (`dart:io`/plugins → FFI → channels) let you access the filesystem, reuse native libraries, and call OS APIs at the right level, with isolates keeping the UI responsive. Choosing the lightest tool, handling FFI's unsafety, and planning for the thinner plugin ecosystem are the architect's calls — enabling desktop apps to do real system work (a key reason to pick desktop over web) while staying maintainable (01_desktop_fundamentals.md, Module 26, Module 34).

Summary

- Desktop gives full OS access via three mechanisms: `dart:io`/plugins (real unrestricted filesystem + native dialogs), FFI (`dart:ffi` — call C/C++ libraries directly, reuse/perf), and platform channels (custom native runner code) — pick the lightest that fits.
- FFI is fast but synchronous/unsafe → isolate heavy calls + manage native memory + bundle libs; channels are per-OS native code; `dart:io` is the first choice for filesystem/OS.
- Check thinner plugin support (FFI/channel fallback); mind the macOS App Store sandbox; use full isolates for heavy native work.

Revision Notes

- Filesystem: real/unrestricted via `dart:io` (actual paths) + native open/save dialogs (`file_selector` / `file_picker`) + `path_provider` — not mobile scoped storage (macOS App Store = sandboxed exception, entitlements).
- FFI (`dart:ffi`): call C/C++ (C-ABI) libraries directly (`DynamicLibrary.open`, `lookupFunction`, `ffigen`); fast (no round-trip) but synchronous/blocking (isolate heavy) + unsafe native memory (allocate/free); bundle `.dll` / `.dylib` / `.so`. Ideal for reuse/perf.
- Channels: same as mobile but native side = desktop runner (Win32/C++, Cocoa/Swift, GTK/C++) for custom OS code w/o plugin. Choose lightest: `dart:io`/plugin → FFI → channel.

Full isolates; thinner plugin ecosystem (FFI/channel fallback); system tray/notifications/clipboard/global-shortcuts via plugins.

Practice Questions

1. When do you use `dart:io`, FFI, and platform channels respectively?
2. How does desktop filesystem access differ from mobile, and what dialogs do you use?
3. What are FFI's performance benefits and risks?

Coding Questions

1. Read/write a user-chosen file via native open/save dialogs + `dart:io`.
2. Call a C function via FFI (load lib + `lookupFunction`), offloading heavy work to an isolate.
3. Add a platform channel handler in a desktop runner for a custom OS call.

Mini Project

Desktop native integration (Flutter Desktop): Build a tool that (a) opens/saves user-chosen files via native dialogs + `dart:io` (real filesystem), (b) calls a native C/C++ function via **FFI** (with heavy work offloaded to an isolate + safe native-memory handling), and (c) uses a **platform channel** for one custom OS call with no plugin — choosing the lightest mechanism for each and checking plugin desktop support. Acceptance: real filesystem access via native dialogs (`dart:io`); FFI call to a native lib (bundled, isolate-offloaded, memory-safe); one channel for custom OS code (per-runner); lightest-tool-per-need justified; macOS-sandbox + plugin-ecosystem caveats noted.

Packaging & Distribution

> Desktop has **no single store flow** — you **package + sign + distribute per platform**: **Windows** → **MSIX** (Microsoft Store or sideload) or an installer (Inno Setup/NSIS), signed with a **code-signing cert** (else SmartScreen warnings); **macOS** → a `.app` in a `.dmg` / `.pkg`, **code-signed + notarized** by Apple (Gatekeeper blocks unnotarized apps) or shipped via the Mac App Store (sandboxed); **Linux** → **AppImage / Flatpak / Snap /** `.deb` / `.rpm` across distros. Distribution is **stores** (discovery + auto-update, but review/fees/sandbox) **or direct download** (full control, but you handle signing + **auto-updates** yourself). Plan packaging/signing/updates **per OS**, automated in **per-OS CI**.

Introduction

This file covers turning built binaries into distributable, signed, updatable apps on each OS, and the stores-vs-direct distribution choice. It's the desktop analog of mobile deployment (Module 51) but **three-headed** and store-optional, building on signing concepts (50 · build_signing).

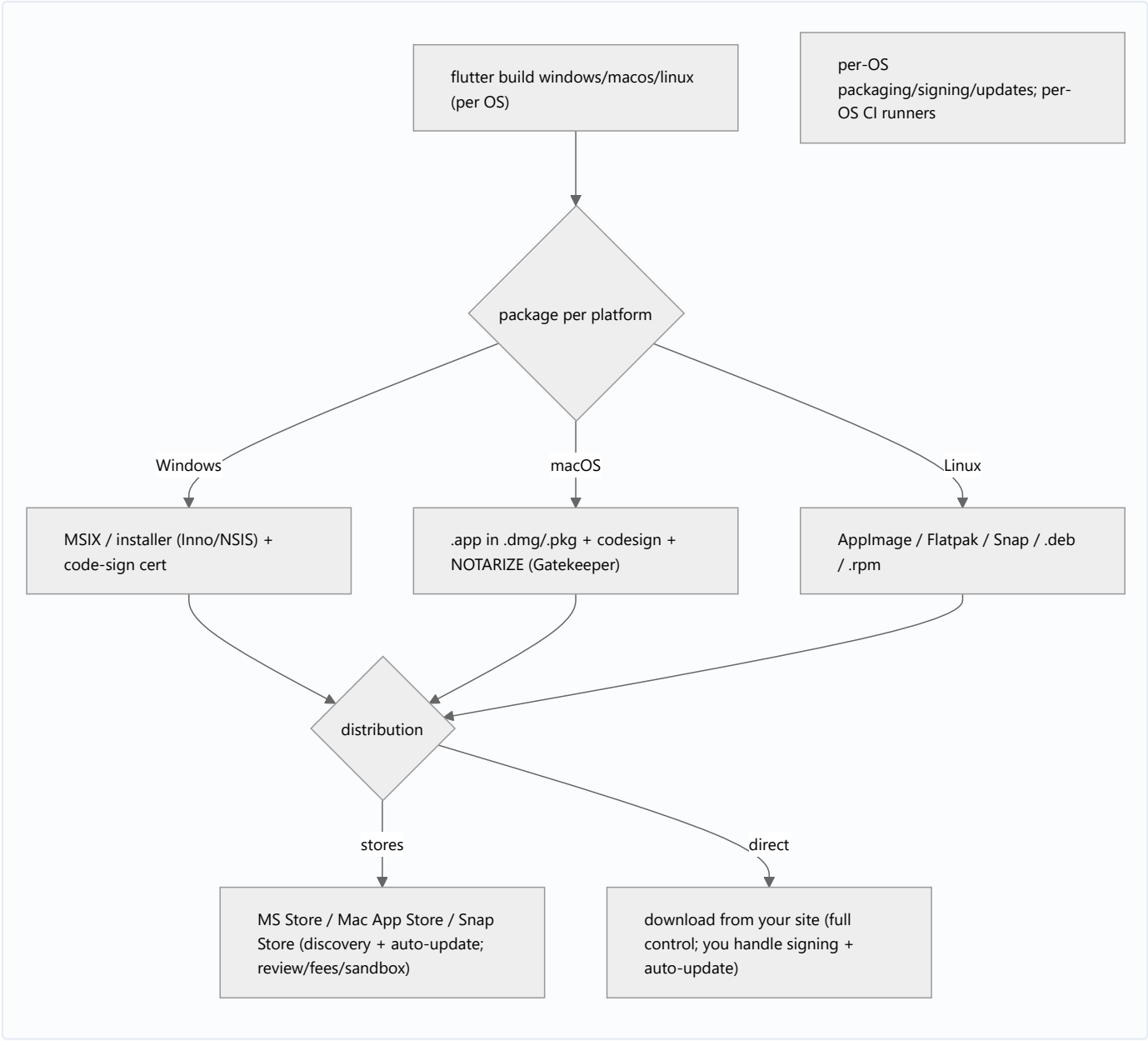
Why this concept exists

Unlike mobile's two stores, desktop is **three OSes with divergent packaging formats, signing/notarization requirements, and distribution channels** — and users can get **security warnings/blocks** for unsigned/unnotarized apps. There's no unified "upload and release." Knowing each platform's packaging + signing + distribution + update story is what gets a trusted, installable, updatable app to users on all three.

Real-world analogy

Shipping desktop is like **selling a physical product in three countries**, each with its own **packaging regulations, safety certification, and retail rules**: one needs a specific box + a safety seal (MSIX + code sign), another requires **customs inspection/notarization** before it clears (macOS notarization/Gatekeeper), the third accepts several **container types** depending on the shop (AppImage/Flatpak/Snap/deb/rpm). You either sell **through regional retailers** (stores: discovery + returns/updates, but fees/rules) or **ship direct to customers** (you handle certification, delivery, and **product recalls/updates** yourself).

Internal Working



- **Windows packaging**:
  - **MSIX** (modern): for the **Microsoft Store** or sideload; supports clean install/uninstall + Store auto-update. Flutter tooling/ `msix` package builds it.
  - **Installer** (Inno Setup / NSIS / WiX): a classic `.exe` installer for **direct download** — common outside the Store.
  - **Code signing**: sign with an **Authenticode certificate** — **unsigned apps trigger SmartScreen/AV warnings** and erode trust; get a cert (OV/EV) and sign the binary/installer.
- **macOS packaging**:
  - Ship the `.app` bundle inside a `.dmg` (drag-to-Applications) or a `.pkg` installer.
  - **Code-sign + notarize (required for direct distribution)**: sign with an **Apple Developer ID** cert and **notarize** with Apple (upload → Apple scans → staple ticket) — else

Gatekeeper blocks the app ("can't be opened, unidentified developer"). Fastlane/`notarytool` automate it.
    - **Mac App Store** alternative: distribution + auto-update via the Store, but **sandboxed** (entitlements) + review + fees.
- **Linux packaging (most fragmented)**:
    - **AppImage** (single portable file, runs across distros — easy direct download), **Flatpak** (sandboxed, Flathub distribution), **Snap** (Snap Store), or native `.deb` / `.rpm` packages. Choose based on target distros/audience; AppImage/Flatpak are common for broad reach.
    - Signing/trust is less centralized than Windows/macOS.
- **Distribution: stores vs direct** (the choice):
    - **Stores** (MS Store / Mac App Store / Snap Store / Flathub): **discovery + trusted install + auto-update handled**, but **review, fees, sandboxing (esp. Mac App Store)**, and platform rules.
    - **Direct download** (your website): **full control + no fees/review + no forced sandbox**, but **you handle signing/notarization + auto-updates + trust** yourself. Common for pro tools/internal apps.
    - Often **both** (Store for reach + direct for control).
- **Auto-updates (you own it for direct distribution)**: stores auto-update; **direct downloads need your own updater** — check-for-update + download + apply (e.g., an updater framework, **Sparkle** on macOS, custom on Windows/Linux, or a package like `auto_updater`). Plan versioning + update UX; mobile-style "forward-fix" applies (can't force instant update).
- **Signing/notarization = trust**: unsigned/unnotarized apps get **OS warnings/blocks** (SmartScreen/Gatekeeper) that scare users off — signing (+ macOS notarization) is **essential for direct distribution**, not optional.
- **Automate in per-OS CI** (Module 50): build each platform on its **own OS runner**, package + sign (secrets in encrypted store) + notarize (macOS) + upload to store/hosting — per-platform jobs. iOS-style secret handling for certs/keys.

Memory Representation

Not app state — **per-platform distributable artifacts** (MSIX/installer `.exe`, signed+notarized `.app`/`.dmg`, AppImage/Flatpak/`.deb`) + signing material (certs/keys in encrypted CI secrets) + an update channel/manifest (for direct auto-update).

Compiler / Build Behavior

`flutter build <platform>` (on its OS) → the native binary; packaging tools wrap + sign it; macOS notarization is a post-sign upload/scan/staple step. CI needs per-OS runners + secret-held signing material.

Runtime Behavior

Signed/notarized apps install + launch without OS warnings; unsigned ones trigger SmartScreen/Gatekeeper. Store apps auto-update; direct apps update via your updater. macOS App Store apps run sandboxed.

Flutter Engine Behavior

Not applicable (distribution wraps the built app). The packaged app runs the native engine as built.

Dart VM Behavior

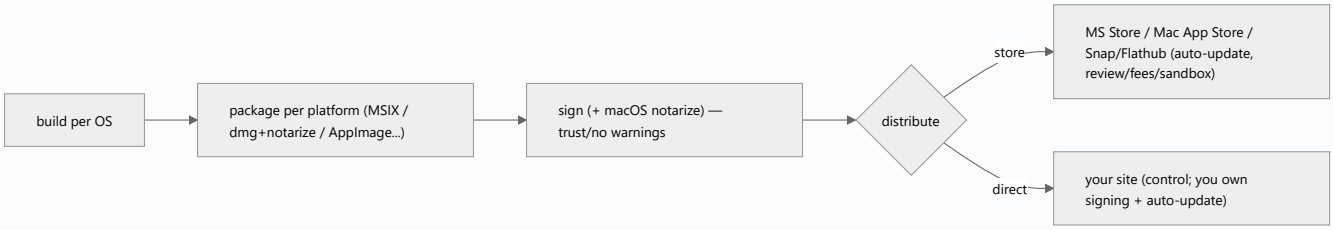
Not applicable (AOT native binary distributed).

Examples

```
Per-platform packaging + signing + distribution:
  Windows: flutter build windows -> MSIX (`msix` pkg) [MS Store/sideload] OR Inno/NSIS installer [direct]
           -> sign with Authenticode cert (avoid SmartScreen/AV warnings)
  macOS:   flutter build macos -> .app in .dmg/.pkg -> codesign (Developer ID) + NOTARIZE (notarytool)
           -> staple ticket (else Gatekeeper blocks) OR Mac App Store (sandboxed + review)
  Linux:   flutter build linux -> AppImage (portable) / Flatpak (Flathub) / Snap / .deb / .rpm (by
audience)

Distribution choice:
  stores  = discovery + auto-update handled, but review/fees/sandbox
  direct  = full control + no fees, but YOU handle signing/notarization + AUTO-UPDATES + trust
  often BOTH.
```

```
# CI (per-OS runners) — build + package + sign per platform (secrets encrypted)
jobs:
  windows: { runs-on: windows-latest, steps: [ build windows, make MSIX, sign (cert from secret) ] }
  macos:   { runs-on: macos-latest,   steps: [ build macos, codesign (Developer ID), notarize (notarytool),
       staple, make dmg ] }
  linux:   { runs-on: ubuntu-latest,  steps: [ build linux, make AppImage/Flatpak ] }
# signing certs/keys/notary creds -> encrypted CI secrets (like mobile signing, Module 50)
```

Diagrams



Common Mistakes

Mistake	Why it's a problem	Fix
Shipping unsigned Windows apps	SmartScreen/AV warnings scare users	Authenticode code-sign the binary/installer
Skipping macOS notarization	Gatekeeper blocks the app	codesign (Developer ID) + notarize + staple
Assuming a single store/flow	Desktop is 3 OSes, store-optional	Per-platform packaging + distribution plan
Ignoring auto-update for direct distribution	Users stuck on old versions	Ship an updater (Sparkle/custom/auto_updater)
Building all platforms on one OS	Must build on each	Per-OS CI runners
One Linux format for everyone	Distro fragmentation	Choose AppImage/Flatpak/Snap/deb by audience
Secrets/certs in the repo	Security breach	Encrypted CI secrets (like mobile signing)
Forgetting Mac App Store sandbox	File/feature access blocked	Entitlements / plan sandbox

Best Practices

- **Package per platform**: Windows **MSIX/installer** (Store/direct), macOS `.app` **in** `.dmg` **/** `.pkg`, Linux **AppImage/Flatpak/Snap/deb/rpm** (by audience) — built on **per-OS runners**.
- **Sign for trust**: Windows **Authenticode**, macOS **Developer ID sign + notarize + staple** (avoid SmartScreen/Gatekeeper blocks); store signing material in **encrypted CI secrets**.

- Choose **distribution deliberately**: **stores** (discovery + auto-update, but review/fees/sandbox) vs **direct** (control, but you own **signing/notarization + auto-updates + trust**) — often **both**.
- Provide **auto-updates** for direct distribution (Sparkle/custom/ `auto_updater`) with versioning + forward-fix UX; **automate** the whole package→sign→notarize→distribute pipeline in CI (Module 50).

Performance

Not runtime perf — the concern is **install trust + updatability + distribution reach**. Signing/notarization prevent warnings (adoption); auto-updates keep users current; format choice affects install size/portability. Automation (CI) makes releasing repeatable across three OSes.

Advantages / Disadvantages

- **+** Native installers per OS, trusted installs (signing/notarization), store reach + auto-update, or full control via direct + custom updater; automatable in CI.
- **−** Three divergent packaging/signing/distribution flows (no single store), macOS notarization + Linux fragmentation, you own auto-updates for direct, per-OS CI runners + secret management.

Interview Questions

1. 🟢 **How is desktop distribution different from mobile?** — No single store: you package/sign/distribute per OS (Windows MSIX/installer, macOS .dmg + notarize, Linux AppImage/Flatpak/etc.), via stores or direct download.
2. 🟢 **Why sign (and on macOS notarize) desktop apps?** — Unsigned Windows apps trigger SmartScreen/AV warnings and unnotarized macOS apps are blocked by Gatekeeper — signing/notarization = trust + no scary warnings.
3. 🟡 **What are the Windows/macOS/Linux packaging options?** — Windows: MSIX (Store/sideload) or Inno/NSIS installer; macOS: `.app` in `.dmg` / `.pkg` (or Mac App Store); Linux: AppImage/Flatpak/Snap/ `.deb` / `.rpm` .
4. 🟡 **Stores vs direct distribution — trade-offs?** — Stores: discovery + auto-update handled, but review/fees/sandbox; direct: full control + no fees, but you handle signing/notarization + auto-updates + trust — often both.
5. 🟡 **Who handles auto-updates, and how?** — Stores auto-update; for direct distribution you ship your own updater (Sparkle on macOS, custom/ `auto_updater` elsewhere) with versioning + forward-fix.
6. 🔴 **Why must you build + package on each OS with CI runners?** — Each platform uses its native toolchain (like iOS needing a Mac); CI needs Windows/macOS/Linux runners to build +

package + sign per platform.

7. 🔴 **What's the macOS App Store caveat vs direct?** — App Store distribution is sandboxed (entitlements) + reviewed + fees; direct distribution is unsandboxed but requires Developer ID signing + notarization yourself.

Senior Engineer Tips

- Code-sign Windows and sign+notarize macOS from the first release; unsigned/unnotarized apps hit SmartScreen/Gatekeeper and users assume malware — it's the biggest adoption killer for direct distribution.
- Decide stores vs direct (or both) per audience and plan auto-updates accordingly; for direct distribution, ship an updater early — a desktop app with no update path strands users on old versions.
- Automate the three per-OS package→sign→notarize→distribute pipelines in CI with secrets in the encrypted store; doing three OSes by hand each release is slow and error-prone.

Architect Perspective

Packaging & distribution is desktop's fragmented last mile: three OSes with divergent formats, signing/notarization for trust, and a stores-vs-direct choice (often both) where direct means owning auto-updates. Unlike mobile's two-store flow, it demands per-OS packaging/signing/CI and a deliberate distribution+update strategy. Getting it right (signed/notarized, updatable, per-audience formats, automated) turns built binaries into trusted, maintainable products; ignoring it yields warning-blocked, un-updatable downloads (01_desktop_fundamentals.md, Module 51, Module 50).

Summary

- No single store: package per OS (Windows MSIX/installer, macOS `.app` + `.dmg` notarized, Linux AppImage/Flatpak/Snap/deb/rpm), built on per-OS runners.
- Sign for trust (Windows Authenticode; macOS Developer ID sign + notarize + staple — avoid SmartScreen/Gatekeeper); secrets in encrypted CI.
- Distribute via stores (discovery + auto-update, review/fees/sandbox) or direct (control, but you own signing/notarization + auto-updates), often both; automate the per-OS pipeline in CI.

Revision Notes

- Windows: MSIX (`msix` pkg — Store/sideload) or Inno/NSIS installer (direct); **Authenticode code-sign** (else SmartScreen/AV warnings).

- macOS: `.app` in `.dmg` / `.pkg`; **codesign (Developer ID) + notarize (notarytool) + staple** (else Gatekeeper blocks) OR Mac App Store (sandboxed + review + fees).
- Linux: AppImage (portable) / Flatpak (Flathub) / Snap / `.deb` / `.rpm` by audience. Distribution: stores (discovery + auto-update; review/fees/sandbox) vs direct (control; YOU own signing/notarization + **auto-updates** — Sparkle/custom/ `auto_updater`), often both.
- Build + package + sign per OS (CI per-OS runners; certs/keys/notary creds in encrypted secrets); automate package→sign→notarize→distribute.

Practice Questions

1. What are the packaging formats + signing needs per OS?
2. Why sign/notarize, and what happens if you don't?
3. Stores vs direct distribution — trade-offs and who handles updates?

Coding Questions

1. Build + package a Windows MSIX (and note code-signing).
2. Codesign + notarize a macOS `.dmg` (outline notarytool steps).
3. Write a per-OS CI matrix that builds + packages + signs each platform.

Mini Project

Desktop packaging + distribution plan (Flutter Desktop): For an app, produce a per-platform packaging/signing/distribution plan: Windows (MSIX or installer + Authenticode signing), macOS (`.app` / `.dmg` + Developer ID sign + notarize + staple, or Mac App Store), Linux (chosen format(s) by audience); a stores-vs-direct distribution decision (with auto-update strategy for direct — Sparkle/custom); and a per-OS CI pipeline (build+package+sign+notarize, secrets encrypted). Acceptance: correct per-OS packaging + signing (notarization for macOS); stores-vs-direct decision + auto-update plan for direct; per-OS CI with encrypted secrets; SmartScreen/Gatekeeper trust addressed; Linux-fragmentation + macOS-sandbox caveats noted.

Flutter Desktop Integration (Capstone: A Production Desktop App)

> Assemble a production desktop app: **confirm fit** (cross-platform tool/productivity/companion), build **native** on each OS, deliver **desktop-idiomatic UX** (window management + native menu bar + keyboard shortcuts + mouse/right-click + responsive layout), integrate **natively** (full filesystem via native dialogs, one FFI/channel native call), and **package + sign + distribute per platform** (MSIX/notarized-DMG/AppImage; stores vs direct + auto-updates) via **per-OS CI** — plus a documented **fit + platform-difference analysis**. This turns "our app also builds for desktop" into a real, native-feeling, distributable Windows/macOS/Linux product — where desktop genuinely fits.

Introduction

This module capstone composes fundamentals/fit, desktop UX/windowing, native integration, and packaging/distribution into one coherent production desktop app + analysis. It's the "put it all together for desktop" deliverable.

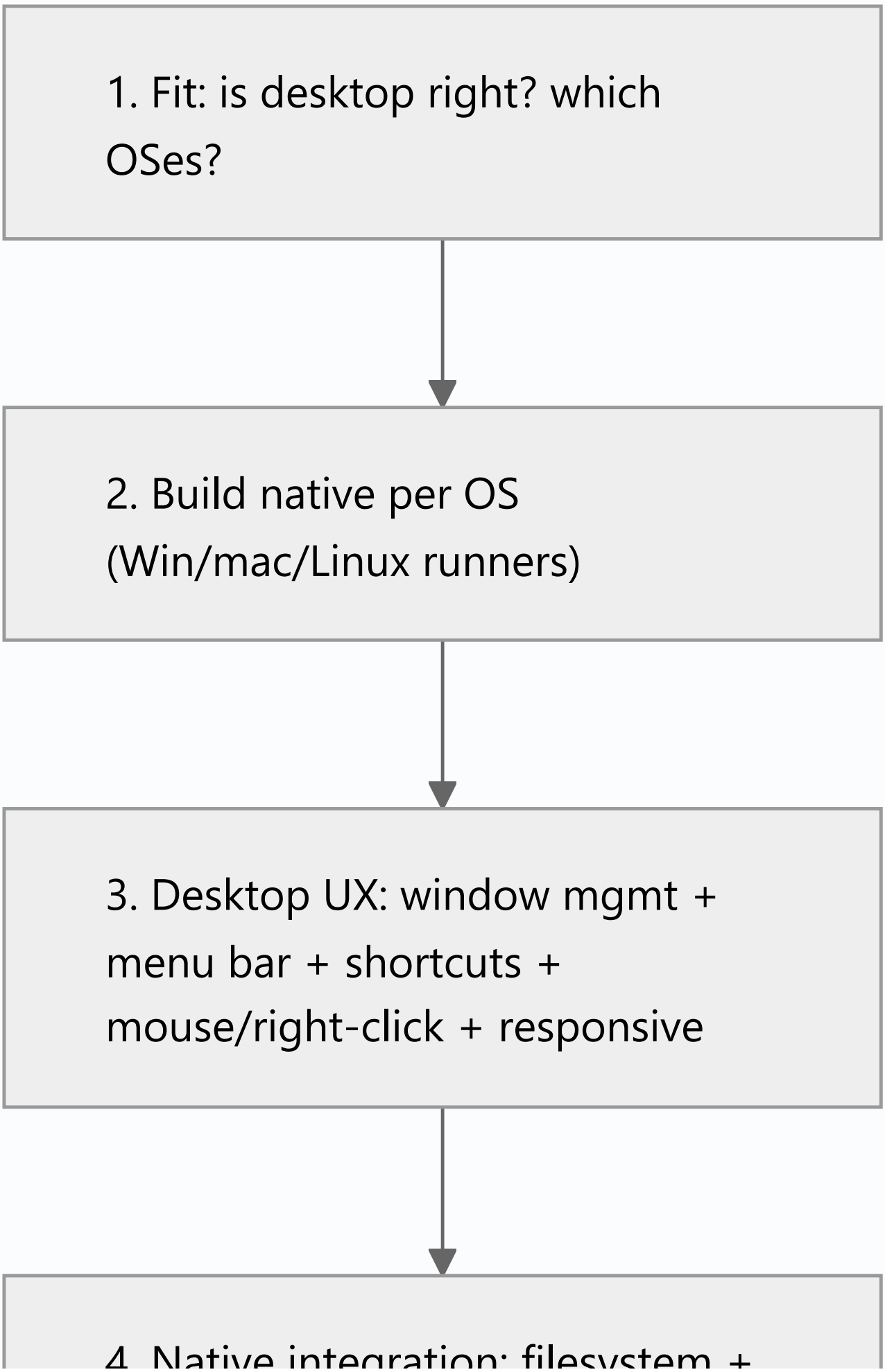
Why this concept exists

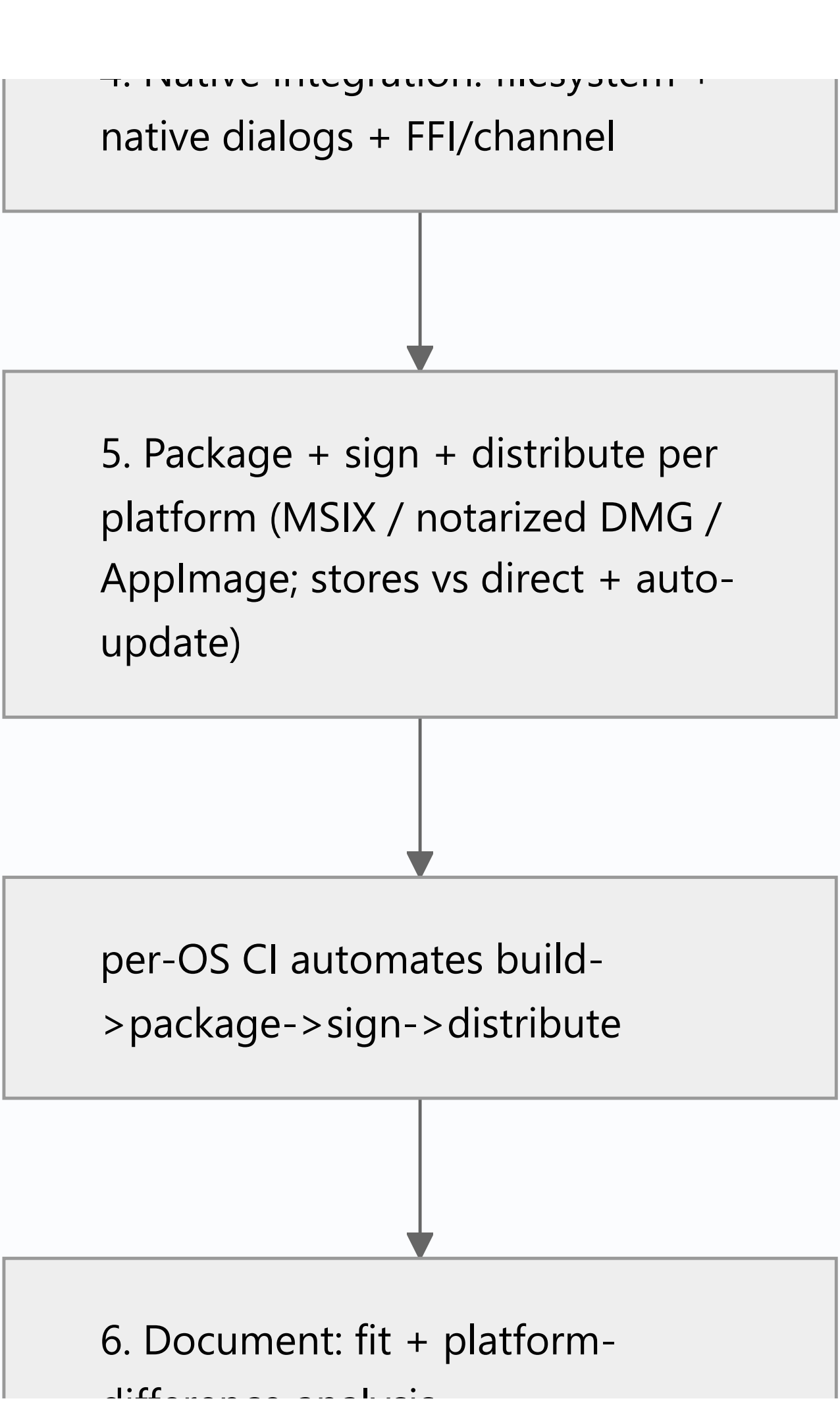
The desktop pieces only yield a good product when **assembled coherently**: fit + UX + native integration + per-platform packaging/signing/distribution, with an honest **fit decision**. This capstone provides that integrated exemplar and cements when/how Flutter Desktop works well.

Real-world analogy

It's **launching a physical appliance across three regions**: confirm there's demand for the appliance (fit), manufacture it **natively in each region** (per-OS build), design it for **desk use** (windows/menus/keyboard/mouse), let it **plug into the home's systems** (filesystem/FFI/native), and **package + certify + ship + service (updates)** per region's rules (MSIX/notarized/AppImage; stores vs direct). Assembled, it's a real product line; half-done, it's a phone gadget awkwardly sold as a desktop appliance.

Internal Working


1. Fit: is desktop right? which OSes?
2. Build native per OS (Win/mac/Linux runners)
3. Desktop UX: window mgmt + menu bar + shortcuts + mouse/right-click + responsive
4. Native integration: filesystem +


native dialogs + FFI/channel
5. Package + sign + distribute per platform (MSIX / notarized DMG / AppImage; stores vs direct + auto-update)
per-OS CI automates build->package->sign->distribute
6. Document: fit + platform-

difference analysis

- **1. Fit** (01_desktop_fundamentals.md): confirm desktop **fits** (cross-platform tool/productivity/creative/companion/enterprise with shared custom UI); pick target OSes; consider **web/native** alternatives where they'd fit better.
- **2. Build native per OS**: enable + build **Windows/macOS/Linux** on their **own OS runners** (native rendering, full OS access); check **plugin desktop support** (FFI/channels for gaps).
- **3. Desktop UX** (02_desktop_ux_and_windowing.md): **window management** (resizable + min size + remembered position, `window_manager`; maybe multi-window/tray), a **native menu bar** (`PlatformMenuBar`) with **keyboard shortcuts** (platform-aware Cmd/Ctrl) + focus traversal, **mouse conventions** (hover/cursor/right-click/scrollbars), **responsive layout** (breakpoints + max width + desktop patterns), selectable text, and **per-OS HIG** adaptation.
- **4. Native integration** (03_native_integration_desktop.md): **full filesystem** via `dart:io` + **native open/save dialogs** (`file_selector`); one **FFI** call to a native lib (isolate-offloaded, memory-safe) **or** a **platform channel** for custom OS code — lightest mechanism per need.
- **5. Package + sign + distribute** (04_packaging_and_distribution.md): **Windows** MSIX/installer + **Authenticode**; **macOS** `.app` / `.dmg` **signed + notarized**; **Linux** AppImage/Flatpak/etc.; choose **stores vs direct** (+ **auto-update** for direct); automate in **per-OS CI** with secrets encrypted (Module 50).
- **6. Document fit + differences**: a short analysis — **why desktop fits here**, **which OSes + why**, **platform differences handled** (HIG/shortcuts/packaging/sandbox), plugin-ecosystem gaps + FFI/channel fallbacks, and **accepted trade-offs** (reuse vs not-pixel-native + per-OS overhead).
- **The payoff**: a native-feeling, fully-integrated, signed, distributable, updatable Windows/macOS/Linux app from your shared Flutter codebase — genuinely production-quality **where desktop fits**.
- **Right-sizing**: an internal single-OS tool needs fit+build+basic UX+direct-distribution; a public cross-OS product adds full UX polish, per-OS packaging/signing/notarization, store presence, and auto-updates. Scale effort to the product.

Memory Representation

Not app state — a **native product per OS**: platform runners + engine + AOT Dart + assets, packaged (MSIX/DMG/AppImage) + signed/notarized, with an update channel (direct) and a fit/difference doc. Runtime = native processes with windows + full OS resources.

Compiler Behavior

`flutter build <platform>` (per OS) AOT-compiles + builds the native runner; packaging tools wrap + sign; macOS adds notarization. Per-OS CI runners; FFI binds native libs (bundled); conditional code branches per OS.

Runtime Behavior

Runs as a native desktop app: resizable window + menus + shortcuts + mouse; full filesystem/FFI/native access; signed/notarized (no warnings); store or direct-updated. Full isolates for heavy work.

Flutter Engine Behavior

Real engine (Skia/Impeller) → native window via the platform embedder; `PlatformMenuBar` → native menu; the semantics layer supports accessibility.

Dart VM Behavior

AOT native with full isolates + `dart:io` + FFI (unlike web) — heavy native work offloaded to isolates.

Examples

```
// Bootstrap: window mgmt + native menu + shortcut + native file access
Future<void> main() async {
  await windowManager.ensureInitialized();
  await windowManager.setMinimumSize(const Size(900, 600));   // desktop window
  runApp(const DesktopApp());
}

class DesktopApp extends StatelessWidget {
  const DesktopApp({super.key});
  @override
  Widget build(BuildContext context) => PlatformMenuBar(
    menus: [PlatformMenu(label: 'File', menus: [
      PlatformMenuItem(label: 'Open…',
        shortcut: SingleActivator(LogicalKeyboardKey.keyO, meta: Platform.isMacOS, control:
!Platform.isMacOS),
        onSelected: _openFile),                                  // native dialog + dart:io
    ])],
    child: MaterialApp(home: LayoutBuilder(builder: (c, cns) =>
      cns.maxWidth >= 900 ? const _DesktopShell() : const _CompactShell())),  // responsive
  );
}
Future<void> _openFile() async {
  final f = await openFile();                                    // native open dialog (file_selector)
  if (f != null) await File(f.path).readAsString();              // full filesystem
}
```

```
Ship checklist (per OS):
  fit confirmed + target OSes chosen
  build native per OS (own runners) + plugin support checked (FFI/channel fallback)
  UX: window (min size/remembered/multi/tray) + menu bar + shortcuts + mouse/right-click + responsive +
selectable text + per-OS HIG
  native: full filesystem + native dialogs + one FFI/channel integration (isolate for heavy)
  package+sign: Windows MSIX/installer + Authenticode; macOS .dmg sign+notarize+staple; Linux
AppImage/Flatpak/...
  distribute: stores vs direct (+ auto-update for direct); per-OS CI (secrets encrypted)
  document: fit + platform differences + trade-offs
```

Diagrams

Common Mistakes

Mistake	Why	Fix
Shipping a mobile build unchanged to desktop	Un-desktop UX, no packaging	Do fit/UX/native/packaging work
Wrong use case (web-deliverable)	Install overhead vs web reach	Consider Flutter Web/PWA
Fixed phone-size window, no menus/shortcuts	Feels ported	Window mgmt + menu bar + shortcuts + responsive
Unsigned/unnotarized distribution	SmartScreen/Gatekeeper blocks	Sign (Win) + sign/notarize (mac)
Building all OSes on one machine	Must build per OS	Per-OS CI runners
No auto-update for direct distribution	Users stranded	Ship an updater
Ignoring plugin gaps	Missing functionality	Check support; FFI/channel fallback
No fit/difference analysis	Wrong-tool surprises	Document fit + platform differences + trade-offs

Best Practices

- **Confirm fit + target OSes** first; **build native per OS** (per-OS runners; check plugin support, FFI/channel fallbacks).
- Deliver **desktop-idiomatic UX** (window management + native menu bar + keyboard shortcuts + mouse/right-click + responsive + selectable text + per-OS HIG) and **native integration** (full filesystem/native dialogs + one FFI/channel, isolate heavy work).
- **Package + sign + distribute per platform** (MSIX/notarized-DMG/AppImage; stores vs direct + auto-update for direct); **automate in per-OS CI** with encrypted secrets.
- **Document the fit + platform-difference + trade-off analysis**; **right-size** effort (internal single-OS tool vs public cross-OS product).

Performance

Native rendering + AOT + full isolates give strong desktop performance; usual Flutter perf (scoped rebuilds/virtualization) applies at desktop scale, with heavy native work offloaded to isolates (Module 21). Startup is native-fast; no web download/SEO issues.

Advantages / Disadvantages

- **+** Native-feeling, fully-integrated, signed/distributable Windows/macOS/Linux product from a shared codebase; full OS access + performance; great for tools/productivity.
- **−** Real effort across all dimensions (per-OS build/UX/native/packaging/signing/distribution/updates), thinner plugin ecosystem, not pixel-native to each HIG, no single store; must right-size + document trade-offs.

Interview Questions

1. 🟢 **What are the steps to a production Flutter Desktop app?** — Confirm fit + OSes → build native per OS → desktop UX (window/menu/keyboard/mouse/responsive) → native integration (filesystem/FFI/channel) → package+sign+distribute per platform (+ auto-update) → document fit/differences.
2. 🟢 **How do you make it feel native on desktop?** — Window management (resizable + min size + remembered), native menu bar + platform-aware shortcuts + focus traversal, mouse/right-click conventions, responsive layout, selectable text, per-OS HIG adaptation.
3. 🟡 **How do you integrate with the OS on desktop?** — Full filesystem via `dart:io` + native dialogs (`file_selector`); FFI to call C/C++ libraries directly; platform channels for custom OS code — lightest tool per need, heavy work on isolates.
4. 🟡 **How do you distribute across three OSes?** — Package per platform (MSIX/installer, notarized `.dmg` , AppImage/Flatpak), sign (Authenticode / Developer ID + notarize), choose stores vs direct (+ auto-update for direct), automate in per-OS CI.
5. 🟡 **What must the fit/difference analysis cover?** — Why desktop fits (vs web/native), which OSes, platform differences handled (HIG/shortcuts/packaging/sandbox), plugin gaps + FFI/channel fallbacks, and accepted trade-offs.
6. 🔴 **Why per-OS CI runners?** — Each platform builds + packages + signs with its native toolchain (like iOS→Mac); CI needs Windows/macOS/Linux runners.
7. 🔴 **How do you right-size the effort?** — Internal single-OS tool: fit+build+basic UX+direct distribution; public cross-OS product: full UX polish + per-OS packaging/signing/notarization + store presence + auto-updates.

Senior Engineer Tips

- Do the fit decision honestly (desktop vs web/native) and pick target OSes deliberately; the biggest failures are shipping desktop for a web-deliverable app or dumping an unmodified mobile build into a window.
- Front-load desktop UX (window/menu/keyboard/mouse) + signing/notarization + auto-update strategy; those are exactly what "just builds for desktop" skips, and they're what make it feel real and trustworthy.

- Automate the three per-OS package→sign→(notarize)→distribute pipelines in CI and plan for the thinner plugin ecosystem (FFI/channel fallbacks); manual three-OS releases and missing-plugin surprises are the recurring pain.

Architect Perspective

Flutter Desktop integration is the synthesis that turns cross-platform reuse into genuine native Windows/macOS/Linux products: the right fit, per-OS native builds, desktop-idiomatic UX, full OS/native integration, and per-platform packaging/signing/distribution/updates — with an honest limitation analysis. Done fully and where it fits, it delivers a native-feeling, integrated, distributable app from one codebase; done partially or for the wrong use case, it disappoints. The architect's contribution is the fit decision + coherent assembly + per-OS distribution strategy + documented trade-offs (01_desktop_fundamentals.md, 04_packaging_and_distribution.md, Module 53).

Summary

- Production Flutter Desktop = fit + OSes → native build per OS → desktop UX (window/menu/keyboard/mouse/responsive) → native integration (filesystem/FFI/channel) → per-platform package+sign+distribute (+ auto-update) via per-OS CI → documented fit/difference analysis.
- Delivers a native-feeling, integrated, signed, distributable Windows/macOS/Linux app from a shared codebase — where desktop fits (tools/productivity/companion, not web-deliverable).
- Right-size effort to the product; document platform differences + trade-offs (reuse vs not-pixel-native + per-OS overhead).

Revision Notes

- Steps: (1) fit + target OSes (vs web/native) (2) build native per OS (own runners; check plugin support, FFI/channel fallback) (3) desktop UX (window_manager min-size/remembered/multi/tray + `PlatformMenuBar` + shortcuts (Cmd/Ctrl) + focus + mouse/right-click/scrollbars + responsive + selectable + per-OS HIG) (4) native (full filesystem + native dialogs `file_selector` + FFI/channel, isolate heavy) (5) package+sign+distribute (Win MSIX/installer + Authenticode; mac .dmg sign+notarize+staple; Linux AppImage/Flatpak/...; stores vs direct + auto-update for direct; per-OS CI, encrypted secrets) (6) document fit + platform differences + trade-offs.
- Payoff: native-feeling, integrated, signed, distributable, updatable cross-OS app from shared codebase where it fits; right-size; not for web-deliverable use cases.

Practice Questions

1. Walk the steps from fit decision to a distributed desktop app.
2. How do you make it feel native + integrate with the OS?
3. How do you package/sign/distribute + update across three OSes?

Coding Questions

1. Bootstrap window mgmt + a native menu bar + a shortcut + native file access.
2. Add one FFI or platform-channel native integration (isolate for heavy).
3. Write a per-OS CI matrix that builds+packages+signs (+notarizes macOS).

Mini Project

Production desktop app (capstone — Flutter Desktop): Ship a cross-platform tool: confirm fit + target OSes; build native per OS; deliver desktop UX (window management + native menu bar + shortcuts + mouse/right-click + responsive + selectable text + per-OS HIG); integrate natively (full filesystem via native dialogs + one FFI/channel call, isolate for heavy); package + sign + distribute per platform (MSIX/Authenticode, notarized DMG, AppImage/Flatpak; stores vs direct + auto-update for direct) via per-OS CI; and write a fit + platform-difference + trade-off analysis. Acceptance: native builds per OS; desktop-idiomatic UX (not phone-in-window); full-OS native integration (filesystem + FFI/channel); per-platform packaging + signing (+ macOS notarization) + distribution + auto-update plan; per-OS CI with encrypted secrets; documented fit/differences/trade-offs; right-sized.