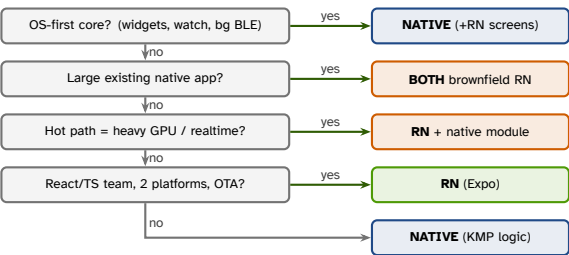


In one line: A senior answers in **trade-offs**: RN = one React/TS codebase driving **real native views**, native code where the OS or the frame budget demands it — and a big app kept **modular, observable, secure, accessible**.

	React Native	Fully native (Swift / Kotlin)	Flutter	KMP / Compose MP
Rendering	JS (Hermes) describes UI; Fabric mounts real UIKit / Android views	UIKit · SwiftUI · Views · Compose	Dart AOT; own engine paints every pixel (Impeller); platform look emulated	shared Kotlin logic + native UI; Compose MP draws its own UI on iOS
Perf ceiling	native views; limits = one JS thread + crossings; hot paths → worklets / native	highest: direct GPU + OS	steady frames, own renderer; embedded platform views cost	native binary (Kotlin/Native); UI as native
Sharing	UI + logic (+ React skills, some web)	none — two codebases	UI + logic, also web/desktop	logic (data, domain, network); UI optional
Native APIs	Turbo / Expo Modules (Swift/Kotlin); big lib ecosystem	day one	platform channels / Pigeon / FFI	iOS via Obj-C interop; Swift-only APIs need a wrapper
Ship fixes	OTA JS updates (EAS Update) per native runtime	store only	store (OTA only via 3rd-party Shorebird)	store only
Hiring	huge React pool; seniors with native depth rare	two specialist teams	smaller Dart pool, fast ramp	Kotlin devs; iOS team must accept Kotlin
Wins when	React org, content/commerce, fast iteration, brownfield	OS-first, perf-critical, deepest polish	pixel-identical brand UI, many targets	native UX + shared logic, incremental

Picture — RN, native, or both?



When to drop to native

- **GPU / realtime** (Metal, AR, video): a native Fabric view wrapped for JS; 2D can stay in JS via Skia. **Frames:** camera/audio on native or worklet threads, never JS per frame.
- **BLE + background:** iOS may suspend the JS runtime; background modes, state restoration, BGTaskScheduler are native — keep the stack native, emit events.
- **OS-first targets:** WidgetKit (SwiftUI, own process), Live Activities, App Intents, watchOS, extensions, CarPlay — native targets sharing data via App Groups.

Architecting a large RN app

- **Monorepo** (workspaces + Nx/Turborepo): apps/mobile, packages/ui, features/* each with a public index; deep imports banned by lint. **Design system:** tokens → primitives → components, a11y built in, Storybook.
- **State by kind:** server cache (TanStack Query) · client (Zustand / Redux Toolkit) · form · nav · persisted (MMKV). Classic bug: server data copied into a global store.
- **Errors:** per-screen ErrorBoundary + global JS handler + native crash SDK; upload **source maps + dSYMs per build and per OTA**. One typed analytics/log facade, PII scrubbed. **Flags:** staged rollout + kill switches, OTA code included.
- **Brownfield (iOS):** JS
AppRegistry.registerComponent('Cart', ...);
a UIViewController hosts it — legacy RCTRootView, current RCTReactNativeFactory →
rootViewFactory.view(withModuleName:). **One** shared runtime; native owns navigation; data via initialProperties + modules/events; release ships main.jsbundle.

Security — the bundle is public

- JS or **Hermes bytecode** is extractable and decompilable; .env values are baked in. **No secrets on the client** — short-lived, scoped tokens only.
- Tokens in **Keychain / Keystore** (react-native-keychain, expo-secure-store), never AsyncStorage (plaintext).

- **Pinning** in the native HTTP layer (TrustKit, OkHttp CertificatePinner); pin SPKI + a backup; a stale native pin needs a **store release**.
- Jailbreak/root checks are heuristics Frida bypasses — cost, not protection; prefer server-verified App Attest / Play Integrity. Minify + bytecode ≠ obfuscation.

Accessibility props

- accessible merges children into **one** element (Pressable default) — nested buttons become unreachable.
- accessibilityLabel · Hint · Role (button, header, switch, adjustable...) · State {disabled, selected, checked, busy, expanded} · Value · accessibilityActions for swipe-only gestures; ARIA aliases (role, aria-label) since 0.71. Keep font scaling; cap with maxFontSizeMultiplier.

Interview traps

- ! “RN is slow because JS” — the cost is *crossings* + a *busy JS thread*; name the thread that dropped frames, in a **release** build.
- ! Error boundaries miss event-handler and async errors.

Remember

Native views, JS brain · cross rarely · no secrets shipped · state by kind.

Questions that separate seniors

1. Why was the bridge slow? — async JSON batches over one queue, no sync calls, all modules init at start; JSI = direct C++ refs, lazy TurboModules.
2. useCallback everywhere? — no: pays only when identity matters (memo’d child, effect deps, renderItem); profile first.
3. OTA vs native code? — an update targets a runtimeVersion (native fingerprint); native change ⇒ new binary.
4. Janky list? — release + Perf Monitor (JS or UI fps?); stable keys, memo’d rows, getItemLayout, sized images, then FlashList.
5. Animation while JS is busy? — native driver (transform/opacity) or Reanimated worklets on the UI thread.
6. What did Fabric change? — shared C++ shadow tree + Yoga, sync layout, concurrent React.
7. Why Hermes? — bytecode compiled at build: faster start, less memory.
8. Crash only in release? — symbolicate (source maps + dSYM); suspect R8/minify, __DEV__ branches.
9. When *not* RN? — OS-first extensions, GPU-heavy core, an all-native team.