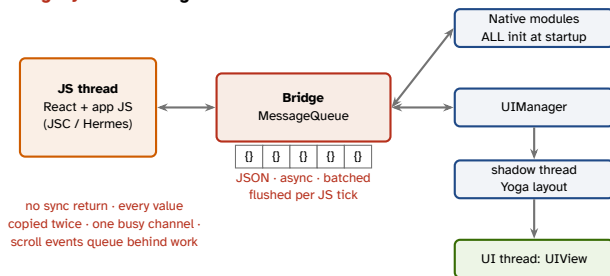
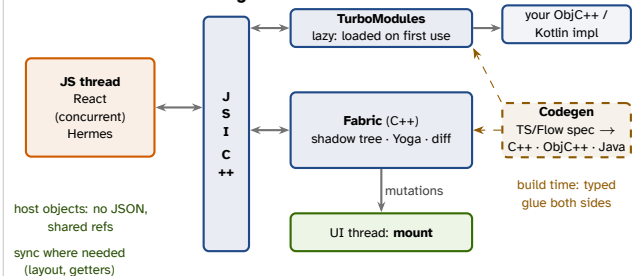


In one line: React runs on a **JS thread**; **real native views** live on the **UI thread**. The legacy **Bridge** joined them with *async, batched JSON*; the **New Architecture** (default since **RN 0.76**) uses **JSI** — JS calls C++ directly, sync or async — with the **Fabric** renderer and lazy, typed **TurboModules** generated by **Codegen**.

Legacy — the Bridge



New Architecture — bridgeless



Threads — who owns what

- JS thread** (one): render, logic, handlers. Block it and JS-driven taps and animations freeze, yet native scrolling still works. Perf Monitor: **JS FPS** $\neq$ **UI FPS**.
- Main/UI thread:** UIViews, gestures, native animations — iOS main-thread rules unchanged. iOS map: JS thread $\approx$ a serial queue owning the model; UI = `DispatchQueue.main`.
- Background:** legacy *shadow thread* ran Yoga; legacy modules ran on their own `methodQueue` (GCD). Fabric lays out on JS or a background thread.

Why the Bridge bottlenecked

- Every call: serialise JSON $\rightarrow$ queue $\rightarrow$ batch-flush $\rightarrow$ parse. **Async only** — no native value in the same frame (measure $\rightarrow$ layout jump).
- One channel: list updates and `onScroll` floods compete. All modules **init at startup** (slow TTI); untyped $\rightarrow$ runtime crashes on bad args.

New Architecture — the pieces

- JSI:** engine-agnostic C++ API (`jsi::Runtime`, `HostObject`, `HostFunction`). JS holds a C++ reference and calls it — no serialisation, **sync** possible.
- Fabric:** C++ renderer shared iOS/Android. *Immutable* shadow tree (clone-on-write, thread-safe), diff in C++, **view flattening** (layout-only Views get no native view). Enables **concurrent React** and **sync layout** reads (useLayoutEffect without flicker).
- TurboModules:** native modules over JSI, **lazy** (first access), typed.
- Codegen:** at build (pod install / Gradle) turns the TS/Flow spec into C++ glue + ObjC++ protocol (`NativeFooSpec`) + Java class. Wrong type = *compile* error.
- Bridgeless:** no Bridge object at all. **Interop layer:** legacy `RCT_EXPORT_MODULE` modules and `requireNativeComponent` views still run — the migration path.

Example — a TurboModule spec

```
// specs/NativeHaptics.ts (name MUST start "Native")
import type { TurboModule } from 'react-native';
import { TurboModuleRegistry } from 'react-native';
export interface Spec extends TurboModule {
  isSupported(): boolean; // SYNC over JSI
  impact(style: string): void; // fire-and-forget
  prepare(): Promise<void>; // async, resolves later
}
export default TurboModuleRegistry.getEnforcing<Spec>('Haptics');
// + package.json codegenConfig; iOS .mm adopts <NativeHapticsSpec>
```

Render $\rightarrow$ commit $\rightarrow$ mount



- Several tree revisions: laid out off-main, swapped in atomically — never a half-applied UI. **Yoga** = C++ flexbox, Auto Layout's role, solved off-main.
- Finger-tracking UI must not round-trip through JS: native-driven animation (Reanimated worklets, `useNativeDriver`).

Hermes — JS engine (default since RN 0.70)

- hermes compiler JS to **bytecode at build time**; app mmap's it — no parse on device $\rightarrow$ faster TTI, less RAM. **No JIT** (JSC in an iOS app has none either). Concurrent GC **Hades**; debug via Chrome DevTools Protocol (React Native DevTools). Trades peak CPU throughput for startup.

Interview traps

- ! "JSI makes RN fast" — it cuts *boundary* cost; JS is still 1 thread.
- ! A slow *sync* TurboModule method **blocks the JS thread** — I/O goes in Promise methods.
- ! Bridge cost $\neq$ just JSON: *async-only* + *one queue* + *eager init*.
- ! Old remote debugging ran JS *in Chrome* — impossible with JSI.
- ! Swift-only module still needs ObjC++ glue — the spec is C++/ObjC++.

Remember

Bridge = mail (JSON letters). **JSI = a phone line to C++.** Fabric draws, TurboModules serve, Codegen types, Hermes pre-compiles.

Likely questions

- Why did the Bridge limit perf? — *async JSON*, one queue, eager init.
- Fabric adds? — C++ immutable tree, concurrent React, sync layout.
- Old module? — interop layer first; then TS spec + Codegen.
- Why Hermes? — AOT bytecode: faster start, less RAM.