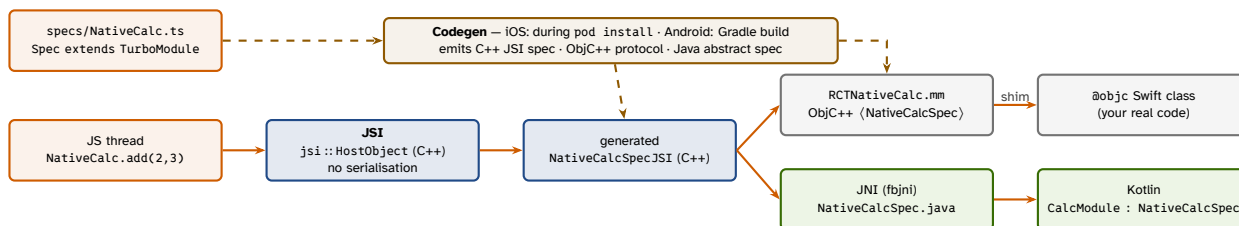


In one line: A typed **TypeScript spec** is the contract; **Codegen** turns it into C++/ObjC++/Java glue; at runtime JS holds a **JSI host object** and calls straight into C++ and on to Swift/Kotlin — no JSON bridge, **sync possible**, modules **lazy-loaded**.



sync add(): number — runs on the JS thread, JS waits: keep it microseconds, never I/O

async hash(): Promise — iOS: module methodQueue (serial, not main) · Android: native-modules thread → UI work hops to main (DispatchQueue.main / UiThreadUtil.runOnUiThread) → resolve once

When you need native — and the flow

- **Need it for:** an API no library covers (HealthKit, a vendor SDK), heavy work off the JS thread (crypto, images), native UI (map, camera), existing Swift/Kotlin code. **First** try an Expo / community module: your native code is yours to upgrade.
- **Flow:** spec Native*.ts (codegen finds it by that prefix) + codegenConfig in package.json → Codegen → implement the generated protocol / abstract class → autolinking registers it → JS TurboModuleRegistry.getEnforcing (throws if absent; get → null for optional modules).
- **iOS:** the protocol and the JSI class use C++ types, so the module is **ObjC++** (.mm): returns std::make_shared<NativeCalcSpecJSI>(params) from getTurboModule:. Swift cannot conform directly — thin ObjC++ shim over an @objc Swift class. **Android:** Kotlin subclass of the generated spec, exposed by a ReactPackage.
- **Events:** JS new NativeEventEmitter(mod); the spec needs addListener + removeListeners. iOS: subclass RCTEventEmitter, list supportedEvents, call sendEvent(withName:body:). Android: emit through RCTDeviceEventEmitter.
- **Fabric UI component:** spec MapViewNativeComponent.ts calls codegenNativeComponent<NativeProps>; props extend ViewProps with codegen types (Double, WithDefault, DirectEventHandler); imperative calls via codegenNativeCommands. iOS: an RCTViewComponentView (updateProps:oldProps:); Android: a ViewManager. Yoga lays out in C++; mounting on main.

Example — TS spec · Expo Module in Swift

```

import type { TurboModule } from 'react-native';
import { TurboModuleRegistry } from 'react-native';
export interface Spec extends TurboModule {
  add(a: number, b: number): number; // sync: JS thread
  hash(input: string): Promise<string>; // async: queue
}
export default
  TurboModuleRegistry.getEnforcing<Spec>('NativeCalc');

import ExpoModulesCore
public class CalcModule: Module { // no ObjC++, no codegen
  public func definition() -> ModuleDefinition {
    Name("Calc")
    Function("add") { (a: Double, b: Double) in a + b }
    AsyncFunction("hash") { (s: String) async -> String in
      await Hasher.sha256(s) }
    Events("onProgress") // sendEvent("onProgress", [...])
  } // JS: requireNativeModule('Calc')
}
  
```

Expo Modules API · legacy · linking

- **Expo Modules** — Swift/Kotlin DSL (Name, Function, AsyncFunction, Events, View+Prop), JSI underneath, works in bare RN too (needs expo-modules-core). Scaffold: npx create-expo-module. **Usually the simplest path** for app code. Raw TurboModule when you want zero Expo deps or shared C++ modules.
- **Legacy:** RCT_EXPORT_MODULE / RCT_EXPORT_METHOD (Swift via RCT_EXTERN_MODULE), Android @ReactMethod, JS NativeModules.X. New Arch default since **0.76**; an **interop layer** keeps old modules running untyped — migrate, don't rely.
- **Autolinking:** CLI reads each dependency's podspec / Gradle; Podfile use_native_modules! (Expo: use_expo_modules!); library podspec install_modules_dependencies(s). Native change ⇒ pod install + rebuild — Metro reload never loads native code.
- **Types:** number→double, object → NSDictionary / ReadableMap, array → NSArray / ReadableArray, Promise → resolve/reject.
- **Test:** jest.mock the spec — JS tests never touch native; XCTest/JUnit on a plain Swift/Kotlin core (keep glue thin); Detox/Maestro on the example app (npx create-react-native-library scaffolds one).
- **Version:** the spec is an **ABI**. Changing it needs a store build; semver-major for breaking spec changes; peerDependencies on react-native.

Interview traps

- ! Sync method doing disk/network → JS frozen, frames dropped.
- ! UIKit touched from an async method: it is **not** on main.
- ! Promise resolved twice, or never (a hung await).
- ! OTA JS calls a method the binary lacks → crash (runtime version).
- ! “could not be found” = not linked / not rebuilt, not a JS bug.
- ! Swift conforms to a TurboModule spec only via an ObjC++ shim.

Remember

Spec = contract · Codegen = glue · JSI = direct call · sync blocks JS · async ≠ main.

Likely questions

1. Bridge vs JSI? — async JSON batches vs direct C++ calls.
2. Why Codegen? — one spec, typed glue; mismatch fails at build.
3. Which thread? — sync: JS; async: module queue; UI: main.
4. Expo Module or TurboModule? — Expo, unless no-Expo / C++.
5. Native change? — new binary; OTA = JS on the same ABI.