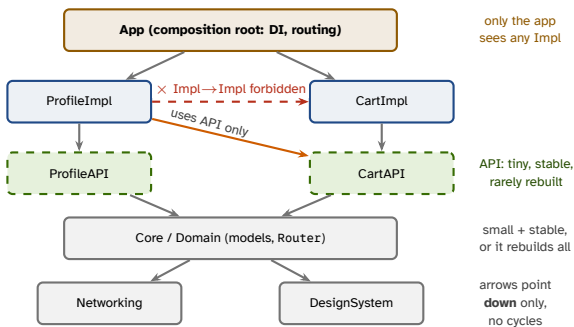
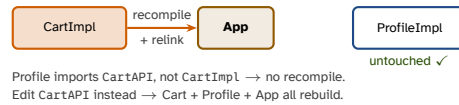


In one line: Cut the app into **modules** (local SPM packages) along seams that change independently; features depend on each other's **interfaces** only, dependencies point **down**, and a thin **app target** composes the concrete implementations. The payoff is faster incremental builds and **compiler-enforced** boundaries.



Edit CartImpl → what rebuilds?



Access across modules (narrow → wide)

private	fileprivate	internal	package	public	open
scope / file		module (default)	same package other modules (Swift 5.9)	use it	+ subclass / override

public class: usable, NOT subclassable outside its module.
 @testable import lifts internal (needs ENABLE_TESTABILITY, Debug); never private.

How it works

- Why:** incremental builds (only a changed module + its dependents recompile), **enforced boundaries** (internal is the default, so nothing leaks by accident), **ownership** (a team per package), isolated tests + fast SwiftUI previews, **reuse** across app, widget, extensions, App Clip.
- Layers:** App → Features → Interfaces/Core → Foundation. Acyclic; SPM rejects cycles, but does *not* forbid Feature→Feature edges — that rule is policy (review, CI graph lint).
- API/Impl split:** FeatureAPI = protocols, routes, DTOs (no heavy deps); FeatureImpl = UI + logic. Others import the API; the app injects the Impl. Cross-feature navigation = a Router/Route enum in Core, resolved by the app.
- Package, swift:** first line // swift-tools-version:5.9; **products** = what consumers can import (.library, .executable, .plugin); **targets** = compilation units = modules; package **dependencies** (which repo/version) and per-target dependencies (.product(name:package:)) — you need both.
- Versions:** from: "1.2.0" = >=1.2.0 <2.0.0; .exact, ranges; commit Package.resolved for apps.
- Resources:** .process (optimised: asset catalogs, localisation) or .copy (verbatim). Load via the synthesised Bundle.module — only generated when the target *has* resources.
- platforms:** [.iOS(.v16)] is the package's *minimum*; it does not set the app's deployment target.
- Binary target:** .binaryTarget(name:url:checksum:) (or path:) wraps a prebuilt .xcframework; checksum of the zip from swift package compute-checksum.
- Plugins:** build-tool (runs every build: SwiftGen, protobuf) vs command (swift package <verb>: lint, format). Sandboxed.

Example — a feature package

```
// swift-tools-version:5.9
import PackageDescription
let package = Package(name: "Cart", platforms: [.iOS(.v16)],
    products: [.library(name: "CartAPI", targets: ["CartAPI"]),
        .library(name: "CartImpl", targets: ["CartImpl"])],
    dependencies: [.package(path: "../Core"),
        .package(url: "https://github.com/apple/swift-log",
            from: "1.5.0")],
    targets: [
        .target(name: "CartAPI", // protocols + models
            .target(name: "CartImpl", dependencies: ["CartAPI",
                .product(name: "Core", package: "Core"),
                .product(name: "Logging", package: "swift-log")],
                resources: [.process("Resources")]), // -> Bundle.module
            .testTarget(name: "CartTests", dependencies: ["CartImpl"])]])
```

Interview traps

- Product ≠ target.** Other packages depend on *products*; a target not exposed as a product is private to its package.
- 40 modules** that import each other's *concrete* Impl = file layout, not a graph: all the overhead, none of the incremental-build win.
- Image blank / nil** inside a package → loaded from Bundle.main; use Bundle.module.
- Over-modularization:** per-module manifest, link and DI cost; clean builds can get *slower*. Cut along ownership/change seams, not per screen.
- A giant Common module** that everything imports and that changes weekly recompiles the world — split it (DesignSystem, Networking, ...).
- ServiceLocator.shared** inside features hides dependencies — inject API protocols through init from the composition root.
- package (Swift 5.9) ≠ @_spi:** package = visible to modules *in the same package*; @_spi(X) = public but only for importers writing @_spi(X) import (underscored, unofficial).
- .library(type:) omitted = automatic** (usually static). Static in app *and* extension = two copies; dynamic = one copy + dyld cost.
- .unsafeFlags** are only allowed in a root package — a dependency using them cannot be consumed by version.

Remember

"API up, Impl hidden, arrows down, app wires." Litmus: *can I build and preview any one feature in seconds with mocks, and does editing X never recompile Y?*

Likely questions

- Why an API/Impl split? — consumers compile against a tiny stable module; Impl edits don't ripple.
- Cross-feature navigation? — Route + Router in Core; the app maps routes to screens.
- public vs open? — open also allows subclass/override outside the module.
- How do SPM resources load? — declared in resources:, read via Bundle.module.
- What does from: "1.2.0" allow? — up to, not including, 2.0.0.
- Where is DI wired? — the composition root in the app target, nowhere else.
- Why commit Package.resolved? — reproducible builds: CI resolves the exact same revisions.