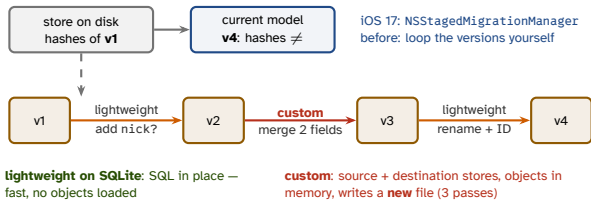
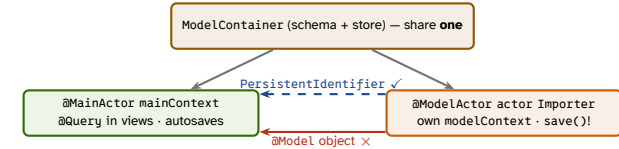


In one line: On open, Core Data compares the store's recorded **entity version hashes** with the current model; if they differ it must **migrate** — **lightweight** (mapping *inferred*, schema-shaped changes only) or **custom** (mapping model + `NSEntityMigrationPolicy` code), one hop at a time. **SwiftData** (iOS 17) is a macro layer on the same engine: `@Model` classes, a `ModelContext` per actor, `PersistentIdentifier` across actors, `VersionedSchema` + `SchemaMigrationPlan` for the same lightweight/custom stages.

Opening an old store: staged, one hop at a time



SwiftData: one container, a context per actor



models are **not** Sendable and belong to their context; re-resolve the id with `context.model(for: id)`. Background write not in the UI? unsaved, or a second container.

Core Data — how it works

- `.xcdatamodeld` holds versions; the **current** one is ticked. Each entity/property has a `versionHash`; the store metadata keeps the set it was written with. Same schema, new meaning → bump `versionHashModifier`.
- Store description flags `shouldMigrateStoreAutomatically` + `shouldInferMappingModelAutomatically` both **default true**: `NSPersistentContainer` migrates lightweight inside `loadPersistentStores`, **synchronously**.
- Custom**: an `.xcmappingmodel` (\$source.x expressions) + an `NSEntityMigrationPolicy` subclass overriding `createDestinationInstances(forSource:in:manager:)`. `NSMigrationManager` writes a new store; swap it in on success.
- Staged** (iOS 17): an `NSStagedMigrationManager` of `NSLightweightMigrationStage` / `NSCustomMigrationStage` (`will/didMigrateHandler`); models as `NSManagedObjectModelReference`; store option `NSPersistentStoreStagedMigrationManagerOptionKey`.

Lightweight can	Needs custom
add / remove attribute, entity optional → required <i>with default</i>	merge first+last → name
rename via Renaming ID	split one entity into two
to-one ↔ to-many, add relationship	de-duplicate, fix bad data
	type change needing logic

Risks · testing

- Data loss: rename without `Renaming ID` (drop + add); required attribute without default (fails); “delete the store” — only for a re-downloadable **cache**.
- Big migration on main at launch → watchdog 0x8badf00d: load off main, show progress, back up first.
- Use `destroyPersistentStore` / `replacePersistentStore`, not `FileManager` — SQLite has `-wal/-shm` sidecars.
- Tests: **golden stores** written by each *old app version* (with sidecars) → migrate a copy → assert counts + values; the whole chain. CloudKit-synced model: additive changes only.

Interview traps

- ! Passing a `@Model` to another actor — pass `persistentModelID`.
- ! `.cascade` with no inverse declared → children may survive.
- ! Adding `.unique` over duplicate rows fails the migration — dedupe in a `.custom` stage's `willMigrate`.
- ! Lightweight “handles everything” — it never computes values.

SwiftData — how it works (iOS 17)

- `@Model` final class: the macro adds `PersistentModel` + `Observable`, backing storage per property, `persistentModelID`. `@Attribute(.unique)`, `@Attribute(originalName:)` (rename), `.externalStorage`, `@Transient`. iOS 18: `#Unique`, `#Index`.
- `.modelContainer(for:)` injects `mainContext` (`@MainActor`, `autosaveEnabled`) into the environment. `@Query(filter:sort:)` is SwiftUI-only and re-renders on change; for a runtime filter set `_items = Query(filter: ...)` in the view's init. Elsewhere: `context.fetch(FetchDescriptor)` with `#Predicate`, `fetchLimit`, `fetchCount`; `context.delete(model:where:)` = batch delete.
- `@ModelActor` synthesises `init(modelContainer:)`, `modelContext`, `modelExecutor`. Background contexts: call `save()` yourself.
- `@Relationship(deleteRule: .cascade, inverse: \Item.folder)`; rules `.nullify` (default) · `.cascade` · `.deny` · `.noAction`. To-many arrays come back **unordered**.

Example — dedupe, then make it unique

```
enum V1: VersionedSchema {
    static let versionIdentifier = Schema.Version(1, 0, 0)
    static var models: [any PersistentModel.Type] { [Trip.self] }
    @Model final class Trip { var name = "" } }
enum V2: VersionedSchema { /* 2.0.0: @Attribute(.unique) */ }
enum Plan: SchemaMigrationPlan {
    static var schemas: [any VersionedSchema.Type] { [V1.self, V2.self] }
    static var stages: [MigrationStage] { [dedupe] }
    static let dedupe = MigrationStage.custom(
        fromVersion: V1.self, toVersion: V2.self,
        willMigrate: { ctx in /* delete duplicate V1.Trips */
            try ctx.save(); didMigrate: nil } }
    let c = try ModelContainer(for: V2.Trip.self,
        migrationPlan: Plan.self)
```

SwiftData vs Core Data

SwiftData: Swift-native, `Observable`, SwiftUI-first, iOS 17+. Core Data still for FRC-driven UIKit lists, batch insert/update, derived attributes, fine context control, older OS, complex migrations. Both can open one store (same schema; rename the `NSManagedObject` subclasses so class names don't clash).

Remember · likely questions

Hash → hop → verify on a golden store. Lightweight vs custom? — inferred vs your code. Cross-actor? — the id. UI stale? — no `save()`, or two containers.