

In one line: SwiftUI keeps state and view lifetime per **identity**, and re-runs body only for views whose **dependencies** changed. Fast SwiftUI = **stable identity** (state survives, diff is cheap) + **narrow dependencies** (few bodies run) + **cheap bodies** (each run is fast).

How it works

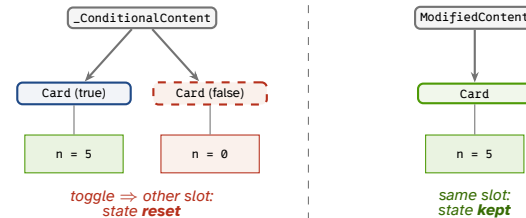
- **Structural identity** — a view's *type* + *position* in the body tree. *if/else* becomes `_ConditionalContent`: the two branches are *different slots*, so switching branch = destroy one view, create the other (state gone, `onAppear` again, transition plays).
- **Explicit identity** — `ForEach` ids (`Identifiable` / `id: \.x`) and `.id(_:)`. A new `.id` value = a brand-new view: legit to *reset* a form or scroll view, costly if it changes every update.
- **AnyView** erases the type, so SwiftUI cannot see the structure: when the wrapped type changes the whole subtree is replaced, and diffing is slower. Prefer `@ViewBuilder`, some `View`, or a modifier on one view (`.opacity`, `.disabled`) over a branch.
- **What invalidates body:** a `@State/@Binding` it reads; an `ObservableObject`'s `objectWillChange` (*any* `@Published` — whole object); an `@Observable` (17) property it *read* during the last body (per property); an `@Environment` value it reads; or new, non-equal inputs from the parent.
- **Skipping:** SwiftUI compares a child's inputs before calling its body; closures and non-equatable fields defeat that. Equatable view + `.equatable()` lets *your* `==` decide — worth it only if `==` is cheap *and* complete.
- **Cheap body:** no `DateFormatter()`, sorting, filtering, image decode or I/O inside body or `init` — precompute in the model, cache in a static `let`, or load in `.task(id:)` (cancelled + restarted when `id` changes).
- **Narrow state:** push state into the smallest subview that reads it; pass plain values, not the whole model, to leaf views.
- **List** (`UICollectionView`-backed since 16) **reuses** cells; `LazyVStack` in a `ScrollView` builds rows *on demand*, *no reuse*; `VStack` builds all rows up front.
- **Pagination:** load more when the last (or *n*-th from last) row appears, guarded by `isLoading` + a page cursor — `onAppear` re-fires on scroll-back.
- **Measure:** `let _ = Self._printChanges()` in body (DEBUG, prints `@self, @identity` or the property that changed). Instruments' **SwiftUI** template: *View Body* counts (pre-Xcode 26); Xcode 26's SwiftUI instrument adds long-update lanes and a *Cause & Effect* graph. Pair with Time Profiler + Hangs.

Example — a lean, paged feed

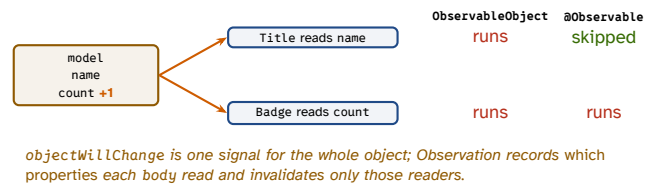
```
@MainActor @Observable final class Feed {
    var items: [Post] = []; var isLoading = false
    func loadMore() async {
        guard !isLoading else { return }; isLoading = true
        defer { isLoading = false }
        items += await api.page(after: items.last?.id) } }
    struct FeedView: View { let feed: Feed
        var body: some View {
            List(feed.items) { post in // Identifiable, stable id
                Row(title: post.title) // a value: Row redraws only
                    .task { // when its title changes
                        if post.id == feed.items.last?.id {
                            await feed.loadMore() } }
            } } }
```

Picture — identity & dependencies

`if on { Card() } else { Card() } | Card().opacity(on ? 1 : 0.4)`



Why re-runs body when `model.count` changes?



Interview traps

- ! `ForEach(items, id: \.self)` on mutable values, array indices, or `UUID()` made in body — every row looks new: lost row state, wrong animations, duplicate-id glitches. Use a stable, entity-owned id.
- ! “`@Observable` fixes everything” — a view that reads a high-churn property, or a whole model passed down, still redraws a lot. Computed properties count as reads of what they touch.
- ! `.id(UUID())` or an `.id` tied to often-changing data silently rebuilds the subtree (and a wrapped `UIView`) on every update.
- ! An incomplete `==` on an Equatable view $\Rightarrow$ **stale UI**; an expensive one is slower than just re-rendering.
- ! `VStack` of 10 000 rows inside a `ScrollView` — eager; use `List` or `LazyVStack`. A `GeometryReader` per row adds layout passes.

Remember

“Same slot, same state. Read less, redraw less. Body is hot code.”

Likely questions

1. Why does *if/else* reset state? — two branches = two structural identities.
2. Why avoid `AnyView`? — hides structure: coarse diffs, subtree replaced on type change.
3. `@Observable` vs `ObservableObject`? — per-property reads vs whole-object signal.
4. `List` vs `LazyVStack`? — reuse + system chrome vs lazy-only + free layout.
5. Why did this view redraw? — `Self._printChanges()`, then the SwiftUI Instrument.