# SwiftUI performance & identity

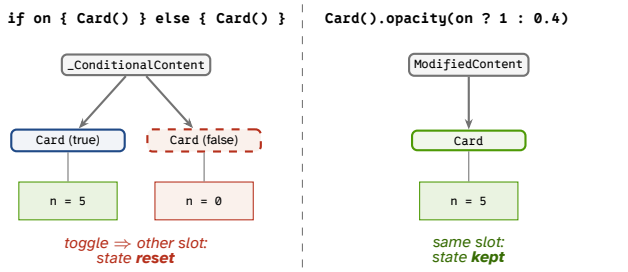


**In one line:** SwiftUI keeps state and view lifetime per **identity**, and re-runs `body` only for views whose **dependencies** changed. Fast SwiftUI = **stable identity** (state survives, diff is cheap) + **narrow dependencies** (few bodies run) + **cheap bodies** (each run is fast).

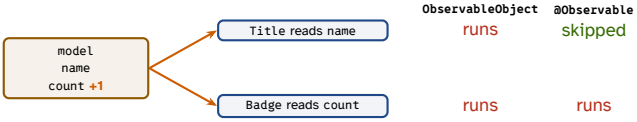
## How it works

- **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). Two states of *one* view? Fold the condition into a value (picture); a `.if(cond) { … }` helper is a branch in disguise.
- **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 `some View` and generics.
- **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 (a missed field ⇒ **stale UI**).
- **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).
- **`@State var m = Model()`** evaluates `Model()` on *every* view init and keeps only the first (`@StateObject` takes an autoclosure, run once): a heavy or side-effecting init ⇒ optional `@State`, created in `.task`.
- **Lazy rows**: `LazyVStack` builds rows *on demand*; a row scrolled away stops updating, is kept for a number of updates, then freed *with its `@State`* — keep row data in the model. Prefetch can run a row's `body` and never call `onAppear`; `onAppear` re-fires on scroll-back, so guard paging with `isLoading` (example). `VStack` builds every row up front.
- **Measure**: `let _ = Self._printChanges()` in `body` (DEBUG, prints `@self`, `@identity` or the property that changed). Instruments' **SwiftUI** instrument (Xcode 26+): long-update lanes and a *Cause & Effect* graph; Xcode 27 also records why a layout was not cached. Pair with Time Profiler + Hangs.

## Identity & dependencies




**Who re-runs `body` when `model.count` changes?**




*`objectWillChange` is one signal for the whole object; Observation records which properties each `body` read and invalidates only those readers.*

## 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 row whose `isFavorite(item)` scans the whole favorites array depends on all of it: one new favorite re-runs *every* row's `body`. Give rows a per-item model or plain values. 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.
- ! `VStack` of 10 000 rows inside a `ScrollView` — eager; use `List` or `LazyVStack`.

## Remember

**"Same slot, same state. Read less, redraw less. Body is hot code."**

## Example — a lean, paged feed

```swift
@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() } }
    } } }
```

*Related:* ios-swift/swiftui-state · swiftui-async-task-data-flow (`.task` keyed by `id`) · swiftui-lists-scroll-lazy (`scrollPosition`, `onScrollVisibilityChange`) · rendering-pipeline (image downsampling) · debugging/swiftui-instrument-hitches-hangs (Time Profiler, Hangs)