

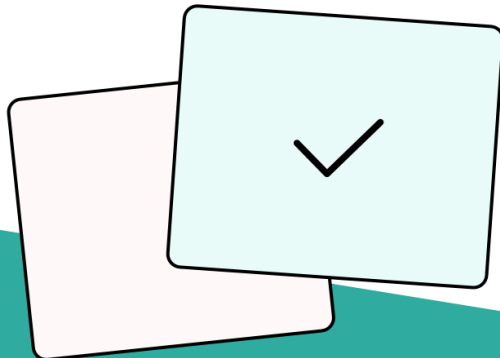
FREE SAMPLE

The SwiftUI Way

A field guide to SwiftUI patterns
and anti-patterns

Natalia Panferova

Released - 2026



Introduction

SwiftUI is very approachable, allowing developers to build functional interfaces quickly and easily. However, as projects grow in complexity, the initial ease of use can give way to subtle architectural challenges.

It may not always be obvious when design and architectural choices are merely a matter of style or fundamental errors that will eventually compromise the application. While some decisions are subjective, others can directly impact application stability and runtime performance.

In an era where AI-assisted coding is becoming the norm, the ability to discern the right patterns from hard-to-spot anti-patterns has never been more critical.

I wrote “The SwiftUI Way” for developers already working with SwiftUI who are looking for deeper guidance on making the right technical trade-offs.

In this guide, we’ll explore the nuances of view structuring to ensure architectures remain manageable as features grow. We’ll examine the selection of tools for data dependencies to avoid unnecessary re-renders and identify performance bottlenecks before they impact the user experience. Furthermore, we’ll look at how to leverage SwiftUI’s native tools to build robust interfaces while knowing exactly when and how to customize them appropriately. The goal of this book is to help you build an intuition for working with the framework rather than against it.

The insights gathered here stem from my experience using SwiftUI in production since its initial release, contributing to projects of various scales, and consulting for numerous developers. This perspective has allowed me to identify common pitfalls and recognize where many developers struggle with the framework.

Crucially, this guidance also draws on my experience working on the core SwiftUI team at Apple. Having been involved in the internal mechanics of

the framework, I have seen firsthand how decisions about APIs are made, what the creators of SwiftUI expect from the call site, and how the system is fundamentally designed to be used.

I hope this book will help you build a deeper alignment with the framework's core principles, allowing you to focus less on fighting the system and more on crafting exceptional user experiences.

The content of this book is copyright Nil Coalescing Limited. It may not be shared or redistributed without prior written permission. If you discover any issues or would like to provide feedback, you can contact support@nilcoalescing.com. Release notes and updates are available at books.nilcoalescing.com/the-swiftui-way.

Building a performant and stable interface

A professional interface must remain fluid and responsive even as layouts become more sophisticated and datasets grow. While SwiftUI provides powerful APIs for layout, collections, and animations, maintaining a stable UI requires a deliberate approach to how we monitor view geometry, manage dynamic content, and trigger motion.

By adopting modern geometry observers, structuring our lists to support efficient loading and updates, and adding intentional animations and transitions, we can ensure our interface stays predictable and performant.

In this chapter, we will see how to leverage coordinate-aware modifiers to monitor view geometry without the overhead of traditional containers, examine strategies for optimizing dynamic lists to ensure stable row identity and lazy loading, and learn techniques for applying precise animations.

Adopting modern layout patterns

This subchapter is not included in the free sample.

Maximizing the performance of dynamic lists

SwiftUI has introduced significant under-the-hood optimizations for lists in recent releases to improve scrolling responsiveness and reduce load times. However, the way we construct these containers can directly impact the framework's ability to manage them efficiently. To ensure a smooth user experience, we should design our list content to support fast identification and lazy view creation.

✅ Recommended patterns

Efficient list performance depends on how quickly SwiftUI can gather identifiers and determine the number of rows to display.

Lists gather all element identifiers eagerly to track data changes and manage view lifetimes. It is critical that accessing these IDs is nearly instantaneous. Using stable, unique identifiers like UUID or integer-based primary keys ensures that SwiftUI can differentiate between elements without performing expensive computations during the identification pass.

```
struct DayWalksView: View {
    @State private var viewModel = DayWalksViewModel()

    var body: some View {
        List {
            ForEach(viewModel.walks) { walk in
                WalkRow(walk: walk)
            }
        }
    }
}

struct Walk: Identifiable {
    var id = UUID()
    var name: String
    var description: String
    var durationInDays: Int
}
```

The example with `DayWalksView` follows a standard pattern for efficient dynamic lists. By conforming the `Walk` model to `Identifiable` with a stored

UUID, we allow SwiftUI to resolve the identity of all items instantly.

While row IDs are gathered upfront, SwiftUI only creates the actual view content for rows on demand, specifically for those in the visible region plus a small buffer. For this lazy creation to work, the content of our `ForEach` must resolve to a constant number of views.

```
@Observable class DayWalksViewModel {
    var walks: [Walk] = []

    func loadWalks() async {
        let allWalks = await loadAllWalks()
        walks = allWalks.filter { $0.durationInDays == 1 }
    }

    private func loadAllWalks() async → [Walk] {
        // ... load all walks ...
    }
}

struct DayWalksView: View {
    @State private var viewModel = DayWalksViewModel()

    var body: some View {
        List {
            ForEach(viewModel.walks) { walk in
                WalkRow(walk: walk)
            }
        }.task {
            await viewModel.loadWalks()
        }
    }
}

struct WalkRow: View {
    let walk: Walk

    var body: some View {
        VStack(alignment: .leading) {
            // ... row subviews ...
        }
    }
}
```

Caching the filtering logic within the view model in our example ensures that the `ForEach` receives a stable, pre-processed collection, moving the computational cost of data preparation out of the identification pass. And since the row is

extracted into a dedicated, single view with a `VStack`, each element resolves to a constant view count per identifier. This predictability allows SwiftUI to know exactly how many rows to expect, enabling it to defer the creation of the row view bodies until they are required for display.

✗ Potentially harmful patterns

List performance can suffer when identifiers are derived from mutable or expensive data, or when the content closure prevents SwiftUI from determining the number and structure of rows efficiently.

Using the entire item as its identifier

When a model conforms to `Hashable`, it can be tempting to use the complete value as its identifier by passing `\.self` to `ForEach`.

```
struct Walk: Identifiable, Hashable {
    let id: UUID
    var name: String
    var description: String
    var durationInDays: Int
}

struct DayWalksView: View {
    let walks: [Walk]

    var body: some View {
        List {
            // ⚠ Potentially harmful: The entire Walk value is used as its ID
            ForEach(walks, id: \.self) { walk in
                WalkRow(walk: walk)
            }
        }
    }
}
```

In this example, the synthesized `Hashable` conformance includes every stored property of `Walk`. Changing the name, description, or duration therefore changes the value used as the row's identity. SwiftUI can interpret the edited walk as one row being removed and another being inserted, which can disrupt row state and animations.

Using the complete value as an identifier can also make collection diffing more expensive. SwiftUI hashes and compares row identifiers when it processes updates, so using a model with large strings, collections, or many stored properties makes this work more costly than using a small identifier such as a UUID or integer.

Conforming our types to Hashable is not itself the problem, because that conformance may be useful elsewhere in the application. But instead of using their complete values as IDs in lists, we should prefer small, stable identifiers that do not change when an instance's other properties are edited.

Returning a variable number of views for each element

Row structure presents a separate performance consideration. To create list content lazily, SwiftUI needs to determine how many views each element produces. If it can't determine this number without evaluating the content closure, it may be forced to build all rows in the list upfront.

A common error is placing conditional logic directly inside a `ForEach` that changes the number of views returned. For example, using an `if` statement to optionally show a view per row forces SwiftUI to visit and instantiate those rows to determine the final count.

```
struct DayWalksView: View {
    @State private var allWalks: [Walk] = []

    var body: some View {
        List {
            ForEach(allWalks) { walk in
                // ⚠ Evaluates for every element upfront
                if walk.durationInDays == 1 {
                    VStack(alignment: .leading) {
                        // ... walk row ...
                    }
                }
            }
        }
        .task {
            allWalks = await loadAllWalks()
        }
    }
}
```

```

private func loadAllWalks() async → [Walk] {
    // ... load all walks ...
}
}

```

In this case, SwiftUI will evaluate the `ForEach` closure for every single element in the collection upfront. This can be particularly harmful for performance if the collection is large and those rows contain intensive memory requirements, such as images. By forcing the system to resolve these views during the identification pass rather than on-demand, we risk saturating the main thread and increasing the memory footprint before a single row is even displayed.

Erasing the type of row content

Type erasure can interfere with lazy row creation in a similar way. Wrapping row content in `AnyView` hides its underlying structure from SwiftUI, so the framework cannot determine the number of rows without evaluating the `ForEach` closure for every element in the collection upfront. This eliminates the performance benefits of lazy loading, leading to significant overhead and a larger memory footprint as the list grows.

```

struct DayWalksView: View {
    @State private var viewModel = DayWalksViewModel()

    var body: some View {
        List {
            ForEach(viewModel.walks) { walk in
                // ⚠ Evaluates for every element upfront
                AnyView(
                    // ... row content ...
                )
            }
        }
        .task {
            await viewModel.loadWalks()
        }
    }
}

```

Efficient list performance is built on predictability. When SwiftUI can calculate the total number of rows and their identifiers without executing every content closure, it can maintain its efficiency even as the dataset grows. By providing

stable identifiers and ensuring each element in a `ForEach` resolves to a constant view count, we can help the system allocate resources only to what is currently visible.

Applying precise and predictable animations

This subchapter is not included in the free sample.