SwiftUI Navigation

Von UINavigationController zum SwiftUI-only Coordinator

Christian Menschel · CocoaHeads · 27.08.2026

Was uns heute erwartet

1. Navigation in **UIKit** vs. **SwiftUI**
2. Warum Navigation aus den Views herraus gehört
3. Coordinator im **UIKit**-Stack
4. Coordinator in der **Mixed World** (UIKit & SwiftUI)
5. Probleme der Mixed-World
6. Coordinator mit **SwiftUI-only**
7. Step-by-step: die **Quäldich-Umsetzung**

1. Navigation in UIKit vs. SwiftUI

UIKit: UINavigationController

`UINavigationController` verwaltet einen **Stack von View Controllern** — Navigation ist **imperativ**: wir sagen dem System jeden Schritt

```
// Push
navigationController?.pushViewController(detailVC, animated: true)

// Pop
navigationController?.popViewController(animated: true)
navigationController?.popToRootViewController(animated: true)
navigationController?.popToViewController(someVC, animated: true)

// Stack komplett ersetzen (z. B. für Deep-Link)
navigationController?.setViewControllers([root, list, detail], animated: true)
```

- Navigations-State lebt **im** **`UINavigationController`** — nicht im Modell
- Kein deklaratives Ziel — nur eine Kette von Aufrufen

UIKit Push

```
final class MembersViewController: UITableViewController {
    override func tableView(_ tableView: UITableView,
                            didSelectRowAt indexPath: IndexPath) {
        let detailVC = MemberDetailViewController(member: members[indexPath.row])
        navigationController?.pushViewController(detailVC, animated: true)
    }
}
```

- Der ViewController kennt den **konkreten** nächsten VC
- Der Flow ist in den ViewControllern verdrahtet
- Flow ist nicht zentral anpassbar

UIKit: Modal Presentations

Modals sind ein **zweiter Stack** — auch dezentral gesteuert

```
// Einfaches Modal
let vc = ComposeViewController()
present(vc, animated: true)

// Mit eigener Navigation
let nav = UINavigationController(rootViewController: ComposeViewController())
nav.modalPresentationStyle = .formSheet
present(nav, animated: true)

presentedViewController?.dismiss(animated: true) {
    // completion
}
```

SwiftUI vor iOS 16: `NavigationView` + `NavigationLink`

```
NavigationView {
    List(items) { item in
        NavigationLink(destination: DetailView(item: item)) {
            Text(item.title)
        }
    }
}
```

- **Deklarativ**: Wenn Zelle geklickt, zeige `DetailView`
- Ziel-View ist **direkt** in der View — schwer trennbar
- Flow auch nicht zentral anpassbar
- Unübersichtlich bei komplexens Navigation-Flows

SwiftUI ab iOS 16: `NavigationStack` mit Pfad

```
@State private var path: [Item] = []

NavigationStack(path: $path) {
    List(items) { item in
        Button(item.title) { path.append(item) }
    }
    .navigationDestination(for: Item.self) { item in
        DetailView(item: item)
    }
}
```

- Navigations-Zustand über `path`
- Push = `path.append(...)`, Pop = `path.removeLast()`, Root = `path.removeAll()`
- Ziel-View wird **einmal registriert** — Trigger und Ziel sind entkoppelt
- Aber: State liegt lokal → Auch keine zentrale Flow-Steuerung

SwiftUI: Modale Präsentation

```
@State private var editing: Item?
@State private var showFilter = false

someView
    .sheet(item: $editing) { EditView(item: $0) }
    .fullScreenCover(isPresented: $showFilter) { FilterView() }
```

- Modals sind **Bindings auf State**.

UIKit vs. SwiftUI

	UIKit	SwiftUI
Paradigma	imperativ	deklarativ
Navigation-State	in `UINavigationController`	in `@State` / `@Observable`
Push	`pushViewController(_:)`	`path.append(...)`
Modal	`present(_:animated:)`	`.sheet` / `.fullScreenCover`
Ziel	direkte VC-Instanz	`.navigationDestination(for:)`
Deep Link	Stack neu bauen (imperativ)	Pfad-Daten setzen

Zwischenfazit

- UIKit koppelt Navigation schnell an **konkrete View Controller** und direkte Aufrufe
- SwiftUI macht Navigation greifbarer: als **State**, der den sichtbaren Pfad beschreibt
- Aber: **Wer entscheidet über den nächsten Schritt im Flow?** Diese Logik gehört nicht in die View

2. Warum Navigation von den Views trennen?

UI-Navigation Flow wird komplex & unübersichtlich

- Navigation-Flow ist verteilt auf viele Views
- Keine zentrale Stelle zum Anpassen des Flows (Reihenfolge von Views oder Push vs. Modal)
- Bei komplexen Flows: Schwer für den Code-Review zu verstehen

Was uns das kostet

- Testbarkeit
- Wiederverwendung
- Deep-Linking
- State-Restoration
- Code-Reviews

Der Coordinator-Ansatz

> **Soroush Khanlou (2015):** *„A view controller shouldn't know about its context."*

Idee:

- Views sagen nur **was passiert ist** ("Item geklickt", "Speichern gedrückt")
- Ein **Coordinator** entscheidet, **was als Nächstes** kommt

Was gewinnen wir?

- Views sind **testbar in Isolation** — sie kennen keine anderen Views
- Ziel-Views und Navigation-Flows sind an **einer Stelle** zentral änderbar
- **Deep-Links** treffen den Coordinator, der den State setzt — nicht die View
- **State-Restoration** greift auf serialisierbaren Navigations-State zu
- Wechsel von Modal → Push? **Eine Zeile** im Coordinator

Was gewinnen wir noch?

- **AB-Tests** von Navigation-Flows werden einfacher — Coordinator entscheidet
- **Team-Skalierung**: Feature-Teams bauen Views, ohne den ganzen Navigations-Flow kennen zu müssen
- **Onboarding neuer Devs**: Ein Ort, um zu verstehen, wie die App zusammenhängt

3. Coordinator im UIKit-Stack

Das klassische Coordinator-Protokoll

```
protocol Coordinator: AnyObject {
    var children: [Coordinator] { get set }
    var navigationController: UINavigationController { get }
    func start()
}
```

- **Owner** eines `UINavigationController`
- **Parent** hält seine Kinder
- `start()` = "zeige den ersten View"

AppCoordinator

```
final class AppCoordinator: Coordinator {
    var children: [Coordinator] = []
    let navigationController = UINavigationController()
    let window: UIWindow

    init(window: UIWindow) {
        self.window = window
    }

    func start() {
        window.rootViewController = navigationController
        window.makeKeyAndVisible()
        showList()
    }

    private func showList() {
        let vc = ListViewController()
        vc.onSelect = { [weak self] item in
            self?.showDetail(for: item)
        }
        navigationController.setViewControllers([vc], animated: false)
    }
}
```

Warum Coordinator

- View, ViewModel und Controller sind Coordinator agnostisch / unabhängig
- Einzelne Views sind wiederverwendbar und vom Flow entkoppelt

SwiftUI in der UIKit-Welt

Ausgangslage vieler Teams (auch Quäldich, historisch):

- Bestehende App auf UIKit + Coordinator.
- Neue View werden in **SwiftUI** gebaut.
- Alter Coordinator soll bleiben → SwiftUI-Views werden in `UIHostingController` verpackt.

4. Coordinator in der Mixed World (UIKit & SwiftUI)

UIHostingController

`UIHostingController<Content: View>` ist ein `UIViewController`,
der eine SwiftUI-View hostet:

```
let vc = UIHostingController(rootView: DetailView(item: item))
navigationController.pushViewController(vc, animated: true)
```

- SwiftUI **innerhalb** einer UIKit-Navigation.
- Auf den ersten Blick perfekt: Coordinator bleibt, Views werden modern.

Coordinator mit SwiftUI-View

```swift
final class AppCoordinator: Coordinator {
    var children: [Coordinator] = []
    let navigationController = UINavigationController()

    func showDetail(for item: Item) {
        let view = DetailView(
            item: item,
            onEdit: { [weak self] in self?.showEdit(for: item) }
        )
        let host = UIHostingController(rootView: view)
        navigationController.pushViewController(host, animated: true)
    }

    func showEdit(for item: Item) {
        let view = EditView(item: item, onDone: { [weak self] in
            self?.navigationController.dismiss(animated: true)
        })
        let host = UIHostingController(rootView: view)
        navigationController.present(host, animated: true)
    }
}
```

Callbacks als Brücke

Die SwiftUI-View nimmt Closures entgegen — der Coordinator behandelt sie

```
struct DetailView: View {
    let item: Item
    let onEdit: () -> Void
    let onShare: () -> Void

    var body: some View {
        VStack {
            Text(item.title).font(.title)
            Button("Bearbeiten") { onEdit() }
            Button("Teilen") { onShare() }
        }
    }
}
```

- View bleibt unabhängig vom **Coordinator**
- Viele Closures pro View

Warum wählen Teams diesen Weg?

- **Migration in kleinen Schritten** — kein Rewrite
- Bewährte **Navigations-Infrastruktur** bleibt
- `UINavigationController` **kann Dinge**, die SwiftUI (lange) nicht konnte: interaktive Pop-Geste, Large Titles, Toolbar-Verhalten…
- Team muss nicht sofort `SwiftUI` komplett nutzen

Klingt gut. Ist es aber nicht wirklich.

5. Probleme der Mixed-World

SwiftUI mit UIKit

Problem 1: Animationen

- **Übergangsanimationen** von `pushViewController` und SwiftUI-View kollidieren:
  - SwiftUI baut Layout **während** des Push auf → sichtbares Nachladen.
  - `.navigationTitle` / `.toolbar` wird **nach** dem Push evaluiert → Titel "poppt" nach.
- **Interaktive Pop-Geste**: SwiftUI-State (Scroll-Position, Focus) wird bei abgebrochener Geste teils inkonsistent.
- **Keyboard + `UIHostingController`**: Layout-Sprünge, `safeArea` läuft nicht sauber durch.

Problem 2: SwiftUI-State fehlt

SwiftUI erwartet:

> **State fließt von oben nach unten, Events von unten nach oben.**

Der UIKit-Coordinator bricht das:

- Die "Quelle der Wahrheit" für die Navigation lebt im **UIKit-Stack**, nicht im State
- Ein `@StateObject` bleibt nicht sauber erhalten, wenn der Host reused wird
- Rerendering reagieren nicht auf Coordinator-Änderungen — der Coordinator ist außerhalb des Environment
- UIKit Imperativ vs. SwiftUI State passen nicht gut zusammen

Problem 3: Environment propagiert nicht sauber

Werte, die per `.environment(...)` gesetzt werden, gelten **nur unterhalb** des `UIHostingController`

```
let host = UIHostingController(rootView:
    DetailView().environment(\.colorScheme, .dark)
)
navigationController.pushViewController(host, ...)
// Jeder weitere Push braucht ein eigenes .environment(...)!
```

- Kein Environment-Sharing zwischen gepushten SwiftUI-Views
- Der Coordinator müsste **jede** View einzeln konfigurieren

6. Coordinator SwiftUI-only

1. Das Route-Protokoll

Alles, was navigierbar ist, ist eine **Route**:

```
protocol Routable: View, Hashable, Identifiable {
    var url: URL { get }
    static func view(for url: URL) -> Self
}
```

- `View`: SwiftUI rendert die Route direkt.
- `Hashable`: `NavigationStack` kann sie in den `path` legen.
- `url`: eine stabile Adresse für Deep-Links und Restoration.

2. AppRoute beschreibt alle Views

```
struct AppRoute: Routable {
    enum Target {
        case web(URL)
        case settings
        case memberDetails(Member)
    }

    let target: Target
    @Environment(AppServiceContainer.self) var appService

    var body: some View {
        switch target {
        case let .web(url): WebView(url: url, service: appService.web)
        case .settings: SettingsView(service: appService.settings)
        case let .memberDetails(member): MemberDetailsView(member)
        }
    }
}
```

- Route kennt View und seine Dependencies

3. NavigationCoordinator hält nur State

```
@Observable
final class NavigationCoordinator<Route: Routable> {
    var path: [Route] = []
    var sheet: Route?

    func push(_ route: Route) { path.append(route) }
    func present(_ route: Route) { sheet = route }
    func pop() { _ = path.popLast() }
}
```

- Kein `UINavigationController`
- Kein `UIHostingController`
- Nur SwiftUI-State, den SwiftUI animieren kann

4. NavigationCoordinatorView verbindet alles

```
struct NavigationCoordinatorView<Route: Routable>: View {
    @State var coordinator: NavigationCoordinator<Route>
    let root: Route

    var body: some View {
        NavigationStack(path: $coordinator.path) {
            root
                .navigationDestination(for: Route.self) { $0 }
                .sheet(item: $coordinator.sheet) { route in
                    NavigationCoordinatorView(
                        coordinator: coordinator.makeChild(),
                        root: route
                    )
                }
        }
        .environment(coordinator)
    }
}
```

Push und Sheet laufen über denselben Typ: `Route`.

5. Views nutzen den NavigationCoordinator

```
struct ArticleListView: View {
    @Environment(NavigationCoordinator<AppRoute>.self) private var coordinator

    var body: some View {
        List(articles) { article in
            Button(article.title) {
                coordinator.push(.view(for: article.url))
            }
        }
    }
}
```

- Die View kennt keine `WebView`.
- Sie sagt nur: "öffne diese Route".
- Der Flow bleibt im Coordinator-State sichtbar.

6. TabNavigationCoordinator

`TabNavigationCoordinator` hält die Tabs zusammen:

```
@Observable @MainActor
final class TabNavigationCoordinator<Route: Routable> {
    var selectedIndex = 0
    private var children: [Int: NavigationCoordinator<Route>] = [:]

    var selectedCoordinator: NavigationCoordinator<Route> {
        child(for: selectedIndex)
    }

    func child(for index: Int) -> NavigationCoordinator<Route> {
        if let coordinator = children[index] { return coordinator }

        let coordinator = NavigationCoordinator<Route>()
        children[index] = coordinator
        return coordinator
    }
}
```

7. TabNavigationCoordinatorView

```
TabView(selection: $coordinator.selectedTabIndex) {
    ForEach(coordinator.tabs) { tab in
        NavigationCoordinatorView(
            root: AppRoute.view(for: tab.rootURL),
            coordinator: coordinator.child(for: tab.index)
        )
        .tag(tab.index)
        .tabItem { tab.tabView }
    }
}
```

- Jeder Tab behält seinen eigenen `path`.
- Der Tab-Coordinator entscheidet, welcher Stack angesprochen wird.
- Tabwechsel zerstört keine Navigation.

`TabNavigationCoordinatorView` ist die SwiftUI-Hülle darum.

8. Modals sind neue Root-Stacks

Eine Sheet bekommt nicht nur eine View, sondern einen eigenen Coordinator:

```
.sheet(item: $coordinator.sheet) { route in
    NavigationCoordinatorView(
        coordinator: coordinator.makeChild(),
        root: route
    )
}
```

Damit kann die Sheet selbst wieder pushen oder weitere Modals öffnen.

9. Deep-Link heißt: Route setzen

```
func handleDeepLink(_ url: URL) {
    tabs.selectedIndex = tabIndex(for: url)

    let route = AppRoute.view(for: url)
    tabs.selectedCoordinator.push(route)
}
```

- Push-Nachricht, Universal Link und In-App-Link laufen gleich.
- Kein Stack aus View Controllern wird rekonstruiert.
- Wir setzen Daten, SwiftUI rendert den Zustand.

10. Restoration speichert URLs

```
var path: [Route] = [] {
    didSet { store(route: path.last?.url) }
}

func restore() {
    guard let url = storedURL else { return }
    push(.view(for: url))
}
```

Für Quäldich reicht der zuletzt sichtbare View.

Das Ergebnis

```
TabNavigationCoordinator
└── TabNavigationCoordinatorView
    ├── Tab 0: NavigationCoordinator<AppRoute>
    │   ├── path: [AppRoute]
    │   └── sheet: AppRoute?
    ├── Tab 1: NavigationCoordinator<AppRoute>
    └── Tab 2: NavigationCoordinator<AppRoute>
```

- Route beschreibt den Screen.
- Coordinator beschreibt den Zustand.

Fazit: SwiftUI Coordinators

1. Fang mit einem kleinen `Routable`-Protokoll an.
2. Views in `AppRoute`.
3. Dependency über Environment in der `AppRoute`
4. Halte `path`, `sheet` und `fullScreenCover` im Coordinator.
5. Views melden Aktionen, sie bauen keine Flows.

Optimierungspotenzial

`PresentationStyle` in die Route ziehen

```
enum PresentationStyle { case push, sheet, fullScreenCover }

protocol Routable: View, Hashable, Identifiable {
    var url: URL { get }
    var presentationStyle: PresentationStyle { get }
    static func view(for url: URL) -> Self
}
```

Optimierungspotenzial

`PresentationStyle` in die Route ziehen

```
extension NavigationCoordinator {
    func show(_ route: Route) {
        switch route.presentationStyle {
        case .push:            push(route)
        case .sheet:           sheet = route
        case .fullScreenCover: cover = route
        }
    }
}
```

- Views rufen nur noch `coordinator.show(route)` — kein `push` vs. `present` mehr am Call-Site
- Push/Modal-Wechsel wird zur **Eigenschaft der Route**, nicht des Aufrufers

Optimierungspotenzial

Dependency Injection sauberer

- Aktuell greifen Routes via `@Environment(AppServiceContainer.self)` auf den ganzen Container zu — **implizit** und **zu breit**
- Besser: pro Route nur das injizieren, was sie wirklich braucht — via Factory oder Route-Init
- Vorteil: Routes werden in Previews & Tests trivial instanziierbar, ohne den kompletten Container zu mocken

```
struct AppRoute: Routable {
    let target: Target
    let factory: RouteFactory   // baut Views mit ihren echten Dependencies
    var body: some View { factory.view(for: target) }
}
```

- **Trade-off**: mehr Boilerplate für den Factory-Layer, aber klarere Grenzen

Danke!

Fragen? Diskussion?

Christian Menschel · CocoaHeads · 27.08.2026