SwiftUI + UIKit Integration Guide

Real-Time, Production-Grade Architecture for Modern iOS Development (2025 Edition)

linkedin.com/in/rajani-bhimanadam

Here's a **real-time, production-grade guide** on how to **integrate SwiftUI into UIKit and UIKit into SwiftUI - common hybrid architecture** used in many modern iOS apps to adopt SwiftUI **gradually**.

Q1: When should you choose SwiftUI over UIKit in modern iOS development (2025 & beyond)?

Answer: You should choose **SwiftUI** when:

- You're building a **new app** that targets **iOS 15 or higher**
- You want to leverage **faster UI development** using a **declarative syntax**
- You need **cross-platform support** (iOS, macOS, watchOS, tvOS) with shared UI code
- You're following **MVVM architecture** and plan to use **Combine** or **Swift Concurrency**
- You want to benefit from **live previews** and **real-time UI iterations** in Xcode

---

Q2: When is UIKit a better choice than SwiftUI?

Answer: UIKit is preferred when:

- You're working on a **legacy codebase** or maintaining **existing UIKit-based apps**
- You need **full control** over the view lifecycle, animations, and system APIs
- You rely on **older SDKs, frameworks**, or **third-party components** that are UIKit-only
- You're building features like **advanced gesture recognizers**, **text input**, or **custom transitions**
- Your app must **support iOS 12 or earlier**

Q3: What's the best practice for teams building scalable iOS apps today?

Answer: The best practice today is to adopt a **Hybrid Approach**:

- Use **SwiftUI** for **new screens, modern UI, and lightweight features**
- Use **UIKit** for:
  - **Complex navigation flows**
  - **Legacy features**
  - **Deep integrations** with lower-level APIs
- **Bridge both** using:
  - UIHostingController to embed SwiftUI into UIKit
  - UIViewControllerRepresentable / UIViewRepresentable to embed UIKit into SwiftUI

This allows for **incremental migration**, preserves existing investments, and prepares your app for the future of Apple platforms.

Q4: Can SwiftUI completely replace UIKit today?

Answer: Not yet. SwiftUI is evolving rapidly, but it still:

- Lacks full parity with UIKit for some advanced use cases

- Has **bugs and inconsistencies across iOS versions**
- May not support **certain animations, gestures**, or **third-party integrations**

Therefore, UIKit remains essential for **production-critical**, **performance-sensitive**, or **deeply customized** applications.

SwiftUI is the Future — but UIKit is Still Essential in 2025

Criteria	SwiftUI ☑ (Modern)	UIKit 🧱 (Essential)
Modern App Development	✔ Designed for declarative, reactive UIs	✘ Verbose and imperative
Multi-Platform Support	✔ iOS, macOS, watchOS, tvOS	✘ iOS-focused
Rapid UI Prototyping	✔ Live Previews + less boilerplate	✘ Requires Storyboards or verbose setup
Performance & Stability	✘ Still evolving (bugs per release)	✔ Mature, proven for complex apps
Legacy/Enterprise Apps	⚠ Limited support for old features	✔ Fully compatible and stable
Fine-grained Control	⚠ Limited customization in some areas	✔ Full control over animations, lifecycle

Best Practice Today: Hybrid Approach

SwiftUI + UIKit Integration is the winning combo. Use SwiftUI for new screens & modern UI, and UIKit for system features, legacy views, or highly custom UIs.

UIKit vs SwiftUI :

1. Architecture and Scalability (Priority: High)

Criteria	UIKit	SwiftUI
Design Patterns	MVC (widely used), MVVM, MVP with manual binding	Encourages MVVM with reactive state (@State, @Binding, ObservableObject)
Scalability	Proven in enterprise-level, large-scale apps	Scalable, but state management can get tricky in complex apps
Separation of Concerns	Requires discipline to maintain separation of UI, logic, and data	Enforced by design, though abstracting logic cleanly still takes effort

Real-World Insight: In a large app, UIKit gives granular control but needs strong architectural discipline. SwiftUI simplifies structure but may need extra layers (e.g., ViewModel, Combine) for complex state flows.

---

2. Performance and Stability (Priority: High)

Criteria	UIKit	SwiftUI
Runtime Performance	Highly optimized and predictable	Fast, but performance drops in nested views or lists

Criteria	UIKit	SwiftUI
Animation Efficiency	Fine-tuned animations with UIViewPropertyAnimator, Core Animation	Declarative animations with .animation(), less control
Stability	Very stable, mature, fewer bugs in production	Still maturing, new bugs appear with each iOS release

Tip: UIKit remains the safer choice in mission-critical apps. SwiftUI needs testing across iOS versions to avoid unexpected behaviors.

---

3. Team Productivity and Collaboration (Priority: Medium-High)

Criteria	UIKit	SwiftUI
Team Onboarding	Steep for juniors, but mature tools and patterns exist	Easier for new devs due to cleaner syntax
Code Review & Diffing	Verbose diffs, view controllers often get bloated	Clean diffs, modular and testable views
Tooling Support	Xcode Interface Builder, Instruments, Debug Hierarchy	Live previews, code-driven only, some tooling gaps

Note: SwiftUI speeds up development for new screens or prototypes. UIKit is easier to debug with Xcode tools in complex UIs.

---

4. Interoperability and Migration (Priority: Medium)

Criteria	UIKit	SwiftUI
Interop	Native in all existing projects	Can embed via UIHostingController, but interop has limitations
Migration Strategy	No migration needed, but hard to modernize	Ideal for gradual adoption: new screens in SwiftUI, legacy stays UIKit

Strategy: Use **SwiftUI for new modules**, maintain **UIKit for core/legacy logic**, migrate gradually.

---

5. CI/CD and Testability (Priority: Medium)

Criteria	UIKit	SwiftUI
Unit Testing	Well-supported MVC/MVVM test structure	Logic in ViewModel is testable, Views are harder to unit test
Snapshot/UI Testing	XCUITest, KIF, Snapshot testing via FBSnapshotTestCase	XCUITest works, snapshot tools still catching up
CI/CD Compatibility	Mature, predictable pipelines	Works in CI but SwiftUI Previews can cause flaky builds

Tip: In CI/CD-heavy teams, UIKit currently offers better testability. SwiftUI still lacks fine-grained UI test tools.

---

6. App Store Readiness and Maintenance (Priority: High for Production)

Criteria	UIKit	SwiftUI
Production Usage	Used in 90%+ of production iOS apps (as of 2025)	Ideal for MVPs, modern iOS-only apps
OS Support	iOS 9+, excellent backward compatibility	iOS 13+, requires fallback strategies for older versions
Long-Term Maintenance	Stable over years, slow API changes	Requires updates per iOS version, API-breaking changes in SwiftUI

Advice: UIKit is better for apps with **multi-year support cycles**. SwiftUI is future-facing but not backward-friendly.

7.Verdict: When to Choose What

Scenario	Best Choice
Large-scale enterprise app	UIKit (with MVVM)
Greenfield project, iOS 14+ only	SwiftUI (with Combine)
Gradual migration to modern architecture	Hybrid (UIKit + SwiftUI)
Complex custom UI / gestures / animations	UIKit
Prototyping or PoC apps	SwiftUI
Need for maximum stability across iOS versions	UIKit

Real-Time Strategy for Teams

- Use **SwiftUI for new features/modules** that are UI-heavy and self-contained.
- Keep **UIKit for complex flows**, navigation, or legacy systems.
- Create **ViewModels** that are UI-agnostic so they can work with both.
- Use **Snapshot testing** with UIKit; rely on **unit + integration testing** in SwiftUI.
- Train juniors in SwiftUI to speed up UI delivery; have seniors own complex UIKit modules.

---

9. Best Practices for UIKit + SwiftUI Integration

Best Practice	Description
Isolate SwiftUI in modules	Keep SwiftUI views self-contained and testable.
Use ViewModels for binding logic	Always use ObservableObject or StateObject for testable shared state.
Use hosting controllers in UIKit storyboards	Use UIHostingController inside a storyboard via container view if needed.
Create bridges using Combine	Use Combine's PassthroughSubject or CurrentValueSubject for event flow.
Avoid deep nesting in SwiftUI	Break complex UI into smaller, reusable SwiftUI views.

Best Practice	Description
Previews + UIKit testing	Use UIViewControllerRepresentable to preview UIKit controllers in SwiftUI (rare, but useful for testing).

Best Practices for UIKit in SwiftUI Integration:

#	Best Practice	Description	Code Snippet / Notes
1	Use UIViewRepresentable or UIViewControllerRepresentable	Wrap UIKit views/controllers for SwiftUI compatibility	struct UIKitView: UIViewRepresentable { ... }
2	Use Coordinators for Delegates	Handle UIKit delegates and events (e.g., UITextFieldDelegate)	class Coordinator: NSObject, UITextFieldDelegate { ... }
3	Use @Binding / @ObservedObject	Enable two-way data binding between SwiftUI and UIKit	@Binding var text: String
4	Respect Layout Using .frame() or Constraints	Manage size/position when embedding UIKit in SwiftUI	UIKitView().frame(height: 200)
5	Apply SwiftUI Modifiers Carefully	Not all SwiftUI modifiers work well on UIKit views	Test .padding(), .background() etc. on wrapped views
6	Avoid Embedding Navigation/Tab Controllers	Use SwiftUI's NavigationStack / TabView instead	Embedding UINavigationController may cause conflicts
7	Handle UIKit Lifecycle Properly	Use updateUIView or updateUIViewController for side effects	Great for initializing camera or map SDKs
8	Isolate UIKit Logic	Keep UIKit-related logic outside SwiftUI views	Use a wrapper UIViewRepresentable to separate concerns
9	Use ViewModifier for Reusable UIKit Logic	Apply UIKit-specific effects in reusable modifiers	Ideal for shadows, blurs, gesture bridges, etc.
10	Test UIKit-SwiftUI Components Separately	Write unit/UI tests for representables and their bindings	Snapshot test the view with known states
🔁	Use Swift Packages for UIKit Wrappers	Encapsulate UIKit code in Swift Packages for reuse	Makes UIKit integration portable and modular

---

Best Practices for SwiftUI in UIKit Integration

#	Best Practice	Description	Code Snippet / Notes
1	Use UIHostingController	Embed SwiftUI views inside UIKit view controllers	let host = UIHostingController(rootView: MyView())
2	Set Auto Layout Constraints on Hosting Controller	Use constraints to position SwiftUI view properly in UIKit	Use NSLayoutConstraint.activate([...])
3	Use @Binding, @ObservedObject, @EnvironmentObject	Enable data flow between UIKit and SwiftUI	MyView(isPresented: $isPresented)

#	Best Practice	Description	Code Snippet / Notes
4	Avoid Deeply Nesting Hosting Controllers	Keep SwiftUI nesting simple and clear to avoid layout issues	Prefer composition over deeply embedded UIHostingControllers
5	Let UIKit Control the Lifecycle	UIKit should own SwiftUI views via UIHostingController	Push/present SwiftUI via UIKit flow
6	Use Combine to Sync Data Between Worlds	Use @Published, PassthroughSubject, or ObservableObject	Bridge data reactively between UIKit and SwiftUI
7	Handle Dismissals in SwiftUI via Bindings or Environment	Use @Environment(\.dismiss) or bindings to dismiss views	For modals: @Environment(\.dismiss) var dismiss
8	Use Hosting Controller for Entire Screens	Best to embed full SwiftUI screens in navigation or tab flow	Avoid mixing UIKit and SwiftUI in the same screen too heavily
9	Inject Dependencies from UIKit into SwiftUI	Pass models, state, or environment via initializer or .environmentObject()	UIHostingController(rootView: MyView().environmentObject(...))
10	Test SwiftUI Views in Isolation Before Integration	Test SwiftUI independently before embedding in UIKit	Ensure layout, state, and performance are stable

10 real-time examples that illustrate how you can integrate SwiftUI views into a UIKit-based project.

1. Embedding a SwiftUI View in a UIViewController

- **Use case:** Display a SwiftUI-based settings panel within an existing UIKit screen.
- **Approach:**
  - Wrap the SwiftUI view using UIHostingController.
  - Add the hosting controller as a child view controller in your UIViewController.
- **Code:**

```
struct SettingsView: View {
    var body: some View {
        VStack {
            Text("Settings")
            Toggle("Enable Feature", isOn: .constant(true))
        }
    }
}

class SettingsViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let settingsView = SettingsView()
        let hostingController = UIHostingController(rootView: settingsView)
        addChild(hostingController)
        hostingController.view.frame = view.bounds
        hostingController.view.autoresizingMask = [.flexibleWidth,
.flexibleHeight]
        view.addSubview(hostingController.view)
        hostingController.didMove(toParent: self)
```

```
    }
}
```

2. Using SwiftUI as a TableView Header

- **Use case:** Add a SwiftUI header with a user profile picture and name above a UIKit UITableView.
- **Approach:**
  - Create a UIHostingController with the SwiftUI view.
  - Set the hosting controller's view as the tableHeaderView.
- **Code:**

```
struct ProfileHeaderView: View {
    var body: some View {
        HStack {
            Image(systemName: "person.circle")
                .resizable()
                .frame(width: 40, height: 40)
            Text("Username")
        }
        .padding()
    }
}

class ProfileViewController: UITableViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let headerView = ProfileHeaderView()
        let hostingController = UIHostingController(rootView: headerView)
        hostingController.view.frame.size.height = 100
        tableView.tableHeaderView = hostingController.view
    }
}
```

3. Integrating a SwiftUI View as a Custom TableView Cell

- **Use case:** Display complex, dynamic content within a UITableViewCell using SwiftUI.
- **Approach:**
  - Use a UIHostingController to host the SwiftUI content.
  - Add it as a subview of the table view cell's content view.
- **Code:**

```
struct CustomCellView: View {
    var title: String
    var body: some View {
        HStack {
            Text(title)
            Spacer()
            Image(systemName: "chevron.right")
        }
```

```
        .padding()
    }
}

class CustomCell: UITableViewCell {
    private var hostingController: UIHostingController<CustomCellView>?

    func configure(with title: String) {
        let cellView = CustomCellView(title: title)
        hostingController = UIHostingController(rootView: cellView)
        if let hostView = hostingController?.view {
            hostView.frame = contentView.bounds
            hostView.autoresizingMask = [.flexibleWidth, .flexibleHeight]
            contentView.addSubview(hostView)
        }
    }
}
```

4. SwiftUI View as a Navigation Bar Title

- **Use case:** Add a SwiftUI view (e.g., a styled title with an image) as the navigation bar title in a UIKit UIViewController.
- **Approach:**
    - Wrap the SwiftUI view in UIHostingController.
    - Use the hosting controller's view as the navigation item's title view.
- **Code:**

```
struct NavBarTitleView: View {
    var body: some View {
        HStack {
            Image(systemName: "star")
            Text("Custom Title")
        }
    }
}

class MyViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let titleView = NavBarTitleView()
        let hostingController = UIHostingController(rootView: titleView)
        navigationItem.titleView = hostingController.view
    }
}
```

5. Using SwiftUI in a Modal Presentation

- **Use case:** Present a SwiftUI-based login form modally from a UIKit view controller.
- **Approach:**

- Instantiate UIHostingController and present it modally.

- **Code:**

```
struct LoginView: View {
    var body: some View {
        VStack {
            Text("Login")
            TextField("Username", text: .constant(""))
            SecureField("Password", text: .constant(""))
            Button("Submit") { /* Handle login */ }
        }
        .padding()
    }
}

class MainViewController: UIViewController {
    func presentLogin() {
        let loginView = LoginView()
        let hostingController = UIHostingController(rootView: loginView)
        present(hostingController, animated: true, completion: nil)
    }
}
```

6. SwiftUI as a UITableView Footer

- **Use case:** Display a SwiftUI-based promotional banner or a call-to-action button at the bottom of a UITableView.
- **Approach:**
  - Create a UIHostingController with the SwiftUI content.
  - Set it as the tableFooterView of the UITableView.
- **Code:**

```
struct FooterView: View {
    var body: some View {
        Text("Try our premium features!")
            .padding()
            .background(Color.blue)
            .foregroundColor(.white)
            .cornerRadius(10)
    }
}

class TableViewController: UITableViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let footerView = FooterView()
        let hostingController = UIHostingController(rootView: footerView)
        hostingController.view.frame.size.height = 100
        tableView.tableFooterView = hostingController.view
```

```
    }
}
```

7. SwiftUI in a UICollectionView Cell

- **Use case:** Display complex layouts or interactive elements in collection view cells.
- **Approach:**
  - Create a UICollectionViewCell subclass.
  - Use a UIHostingController to embed the SwiftUI view into the cell's content.
- **Code:**

```
struct CollectionCellView: View {
    var title: String
    var body: some View {
        VStack {
            Text(title)
            Divider()
            Text("Additional details here.")
        }
        .padding()
    }
}

class CollectionCell: UICollectionViewCell {
    private var hostingController: UIHostingController<CollectionCellView>?

    func configure(with title: String) {
        let cellView = CollectionCellView(title: title)
        hostingController = UIHostingController(rootView: cellView)
        if let hostView = hostingController?.view {
            hostView.frame = contentView.bounds
            hostView.autoresizingMask = [.flexibleWidth, .flexibleHeight]
            contentView.addSubview(hostView)
        }
    }
}
```

8. SwiftUI Form in a UIKit View

- **Use case:** Replace a legacy UIKit form with a SwiftUI form, making it easier to manage state and layout.
- **Approach:**
  - Embed a SwiftUI Form or List in a UIHostingController.
  - Present it in a UIViewController or as part of a UIKit hierarchy.
- **Code:**

```
struct RegistrationFormView: View {
    @State private var name: String = ""
    @State private var email: String = ""
```

```
    var body: some View {
        Form {
            Section(header: Text("Personal Information")) {
                TextField("Name", text: $name)
                TextField("Email", text: $email)
            }
            Section {
                Button("Register") {
                    // Handle registration
                }
            }
        }
    }
}

class RegistrationViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let formView = RegistrationFormView()
        let hostingController = UIHostingController(rootView: formView)
        addChild(hostingController)
        hostingController.view.frame = view.bounds
        hostingController.view.autoresizingMask = [.flexibleWidth,
.flexibleHeight]
        view.addSubview(hostingController.view)
        hostingController.didMove(toParent: self)
    }
}
```

9. SwiftUI as a UITabBar Item Content

- **Use case:** Implement a custom tab bar item that uses SwiftUI for its appearance or interactions.
- **Approach:**
    - Create a SwiftUI view and wrap it in a UIHostingController.
    - Use it as the custom view for a UITabBarItem.
- **Code:**

```
struct CustomTabItemView: View {
    var body: some View {
        VStack {
            Image(systemName: "star.fill")
            Text("Custom Tab")
        }
        .padding()
    }
}
```

```
class CustomTabBarController: UITabBarController {
    override func viewDidLoad() {
        super.viewDidLoad()
•
        let hostingController = UIHostingController(rootView:
CustomTabItemView())
        let tabBarItem = UITabBarItem()
        tabBarItem.customView = hostingController.view
        // Assign tabBarItem to the desired tab
    }
}
```

10. SwiftUI View in a UIAlertController

- **Use case:** Show a SwiftUI-powered confirmation dialog or informational panel within a UIKit alert.
- **Approach:**
    - Create a UIHostingController with a SwiftUI view.
    - Set the hosting controller's view as the contentViewController of a UIAlertController.
- **Code:**

```
struct AlertContentView: View {
    var message: String
    var body: some View {
        VStack {
            Text(message)
                .font(.headline)
            Button("Okay") {
                // Dismiss action
            }
        }
        .padding()
    }
}

class AlertController: UIViewController {
    func showSwiftUIAlert() {
        let alert = UIAlertController(title: nil, message: nil,
preferredStyle: .alert)

        let contentView = AlertContentView(message: "This is a SwiftUI
alert.")
        let hostingController = UIHostingController(rootView: contentView)

        // Note: For full-screen usage, `alert.contentViewController =
hostingController` would be set.
        // Ensure proper sizing and layout of the hostingController's view.
    }
}
```

Integrating UIKit components into a SwiftUI-based project can be done using UIViewRepresentable and UIViewControllerRepresentable protocols

1. Integrating a UIKit UITextField in a SwiftUI View

- **Use case:** A custom text field that uses UIKit's delegate pattern.
- **Approach:** Create a UIViewRepresentable struct that wraps the UITextField and handles configuration and delegation.
- ***Code:***

```
struct UIKitTextField: UIViewRepresentable {
    @Binding var text: String

    class Coordinator: NSObject, UITextFieldDelegate {
        var parent: UIKitTextField

        init(parent: UIKitTextField) {
            self.parent = parent
        }

        func textFieldDidChangeSelection(_ textField: UITextField) {
            parent.text = textField.text ?? ""
        }
    }

    func makeCoordinator() -> Coordinator {
        Coordinator(parent: self)
    }

    func makeUIView(context: Context) -> UITextField {
        let textField = UITextField()
        textField.delegate = context.coordinator
        return textField
    }

    func updateUIView(_ uiView: UITextField, context: Context) {
        uiView.text = text
    }
}
```

2. Embedding a UIKit UIActivityIndicatorView in SwiftUI

- **Use case:** Show a loading spinner with UIKit's standard appearance.
- **Approach:** Use UIViewRepresentable to wrap UIActivityIndicatorView.
- **Code:**

```
struct ActivityIndicator: UIViewRepresentable {
    var isAnimating: Bool

    func makeUIView(context: Context) -> UIActivityIndicatorView {
```

```
        return UIActivityIndicatorView(style: .medium)
    }

    func updateUIView(_ uiView: UIActivityIndicatorView, context: Context)
{
        if isAnimating {
            uiView.startAnimating()
        } else {
            uiView.stopAnimating()
        }
    }
}
```

3. Integrating a UIKit UIImageView

- **Use case:** Display images with UIKit's built-in image view.
- **Approach:** Use UIViewRepresentable to wrap UIImageView.
- **Code:**

```
struct UIKitImageView: UIViewRepresentable {
    let image: UIImage?

    func makeUIView(context: Context) -> UIImageView {
        return UIImageView()
    }

    func updateUIView(_ uiView: UIImageView, context: Context) {
        uiView.image = image
    }
}
```

4. Embedding a UIScrollView in SwiftUI

- **Use case:** Allow scrolling when SwiftUI's ScrollView doesn't meet specific requirements.
- **Approach:** Wrap UIScrollView using UIViewRepresentable and configure it as needed.
- **Code:**

```
struct UIKitScrollView: UIViewRepresentable {
    func makeUIView(context: Context) -> UIScrollView {
        return UIScrollView()
    }

    func updateUIView(_ uiView: UIScrollView, context: Context) {
        // Configure scroll view content
    }
}
```

5. Using a UIKit UIPageControl

- **Use case:** Display page indicators for a carousel.

- **Approach:** Wrap UIPageControl in a UIViewRepresentable struct.
- **Code:**

```
struct UIKitPageControl: UIViewRepresentable {
    @Binding var currentPage: Int
    let numberOfPages: Int

    func makeUIView(context: Context) -> UIPageControl {
        return UIPageControl()
    }

    func updateUIView(_ uiView: UIPageControl, context: Context) {
        uiView.numberOfPages = numberOfPages
        uiView.currentPage = currentPage
    }
}
```

6. Embedding UICollectionView for Custom Layouts

- **Use case:** Display complex grid layouts not natively supported in SwiftUI.
- **Approach:** Use UIViewControllerRepresentable to wrap a UICollectionViewController.
- **Code:**

```
struct UIKitCollectionView: UIViewControllerRepresentable {
    func makeUIViewController(context: Context) ->
UICollectionViewController {
        return UICollectionViewController(collectionViewLayout:
UICollectionViewFlowLayout())
    }

    func updateUIViewController(_ uiViewController:
UICollectionViewController, context: Context) {
        // Update collection view items or layout
    }
}
```

7. Integrating UIWebView or WKWebView

- **Use case:** Display a web page in a SwiftUI view.
- **Approach:** Wrap a WKWebView (preferred over UIWebView) using UIViewRepresentable.
- **Code:**

```
struct WebView: UIViewRepresentable {
    let url: URL

    func makeUIView(context: Context) -> WKWebView {
        return WKWebView()
    }

    func updateUIView(_ uiView: WKWebView, context: Context) {
```

```
        uiView.load(URLRequest(url: url))
    }
}
```

8. Integrating UIKit Alert Controller

- **Use case:** Present a UIKit-style alert.
- **Approach:** Use UIViewControllerRepresentable to wrap a UIAlertController.
- **Code:**

```
struct UIKitAlert: UIViewControllerRepresentable {
    let title: String
    let message: String

    func makeUIViewController(context: Context) -> UIViewController {
        let controller = UIViewController()
        let alert = UIAlertController(title: title, message: message,
preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: .default,
handler: nil))
        controller.present(alert, animated: true, completion: nil)
        return controller
    }

    func updateUIViewController(_ uiViewController: UIViewController,
context: Context) {}
}
```

9. Using UIKit UINavigationController in SwiftUI

- **Use case:** Embed a UIKit navigation controller for advanced navigation features.
- **Approach:** Wrap UINavigationController using UIViewControllerRepresentable.
- **Code:**

```
struct UIKitNavigationController: UIViewControllerRepresentable {
    func makeUIViewController(context: Context) -> UINavigationController {
        let rootViewController = UIViewController()
        rootViewController.view.backgroundColor = .white
        let navigationController =
UINavigationController(rootViewController: rootViewController)
        return navigationController
    }

    func updateUIViewController(_ uiViewController: UINavigationController,
context: Context) {}
}
```

10. Adding a UIKit UISlider to SwiftUI

- **Use case:** Use a UIKit slider instead of SwiftUI's native slider.
- **Approach:** Wrap UISlider in a UIViewRepresentable struct.
- **Code:**

```swift
struct UIKitSlider: UIViewRepresentable {
    @Binding var value: Float

    func makeUIView(context: Context) -> UISlider {
        return UISlider()
    }

    func updateUIView(_ uiView: UISlider, context: Context) {
        uiView.value = value
    }
}
```