Integrating SwiftUI into UIKit Apps

Natalia Panferova

2022

FREE SAMPLE

Introduction

Thank you for downloading a free sample of “Integrating SwiftUI into UIKit Apps”. The sample includes the second subchapter of Chapter 2 - SwiftUI in a separate view controller, and shows how to set up Hosting Controller in storyboards to present a SwiftUI view within a UIKit app. To leann more about the book and to get the complete copy with 6 chapters (each with multiple subchapters), a PDF, an EPUB and 6 full projects with code illustrating various ways to use SwiftUI in existing UIKit projects, you can visit our website: books.nilcoalescing.com/integrating-swiftui.

To get the most out of the subchapter provided in the free sample, you can follow along with the code examples where we will be adding filtering functionality to a UIKit app called Puppy Training. In the sample bundle you can find the PuppyTraining-starter project that contains the initial setup and the PuppyTraining-final project with the integrated SwiftUI part. In case you don’t have the complete free sample bundle yet, you can get it from the subchapter page: books.nilcoalescing.com/integrating-swiftui/swiftui-in-a-view-controller/hosting-controller-in-storyboards. The projects were created using Xcode 14 and are set to target iOS 16 by default.

The subchapter provided in the free sample already assumes that you have a good understanding of SwiftUI fundamentals. The complete book includes an entire chapter just covering the essentials of the SwiftUI framework that would bring you up to speed in case you haven’t worked with SwiftUI before or need a refresher.

material in the free sample, but you are required to give an attribution. The app icons for the projects in the sample were made using an SVG taken from `iconmonstr.com`.

If you have any questions about the "Integrating SwiftUI into UIKit Apps" book, feel free to reach out to `support@nilcoalescing.com`.

SwiftUI in a separate view controller

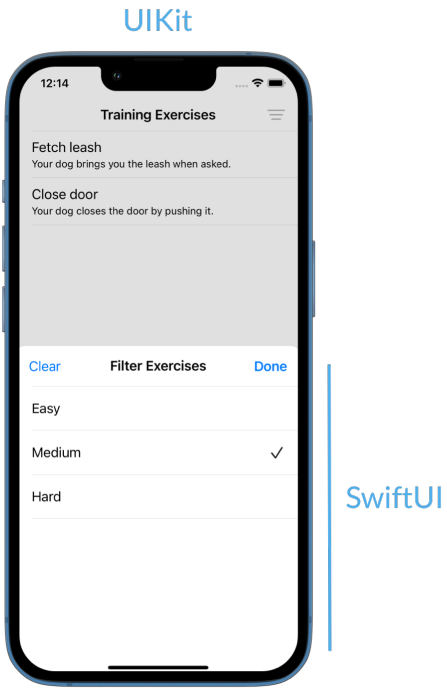
Presenting UIHostingController programmatically

This subchapter is not included in the free sample.

Setting up Hosting Controller in storyboards

When working with storyboards in UIKit, we can use a Hosting Controller from the object library to present a SwiftUI view hierarchy. The controller created in the storyboard can be prepared for presentation inside the storyboard segue action.

We will add a filter view built in SwiftUI to our sample Puppy Training app where users can filter the exercises based on the difficulty level. The filter will be presented in a bottom sheet using a storyboard segue.



Screenshot of the sample puppy training app showing the main collection view built in UIKit and a filter view built in SwiftUI and presented in a bottom sheet

Prepare the filter state data model

We'll start by defining the data model for the filter in a separate file called `Exercise-FilterState.swift`. The view controller in UIKit will own the data model and pass it to the SwiftUI filter view to modify the selection. So that the change in selection can be reflected in both the filter list and the collection view, the object encapsulating it has to conform to the `ObservableObject` protocol, and the `selection` property has to be marked with the `@Published` property wrapper. Both the `ObservableObject` and the `@Published` property wrapper are included in the Foundation framework, so it's the only import we need in this file.

```
import Foundation

class ExerciseFilterState: ObservableObject {
    @Published var selection: Exercise.Difficulty?
}
```

The `ExercisesViewController` defined inside the `ExercisesViewController.swift` file will store an instance of the `ExerciseFilterState` and later pass it to the SwiftUI view. We are going to assign the object to a private property on the view controller.

```
class ExercisesViewController: UIViewController,
    UICollectionViewDelegate {

    private let filterState = ExerciseFilterState()

    ...
}
```

To make sure that the `List` view in the filter sheet can iterate over the difficulty levels, we'll make the `Exercise.Difficulty` enum conform to `CaseIterable`. The exercise model can be found in the `Exercise.swift` file in the `Data` folder.

```
struct Exercise: Hashable {
    enum Difficulty: CaseIterable {
        case easy
        case medium
        case hard

        var description: String {
            switch self {
            case .easy: return "easy"
            case .medium: return "medium"
            case .hard: return "hard"
            }
        }
    }

    ...
}
```

Build the filter view in SwiftUI

Next, we'll build the SwiftUI view we want to present. We are going to define the list of the exercise difficulty levels in a separate component and then embed it into a container with the navigation bar title and controls.

We'll create a new file called `DifficultyFilterList.swift` using the SwiftUI View template. The view will receive a binding to an optional `Exercise.Difficulty` to mark the selection. Each difficulty level will be presented in a `List` row. The currently selected level for the filter will be marked with a checkmark symbol image.

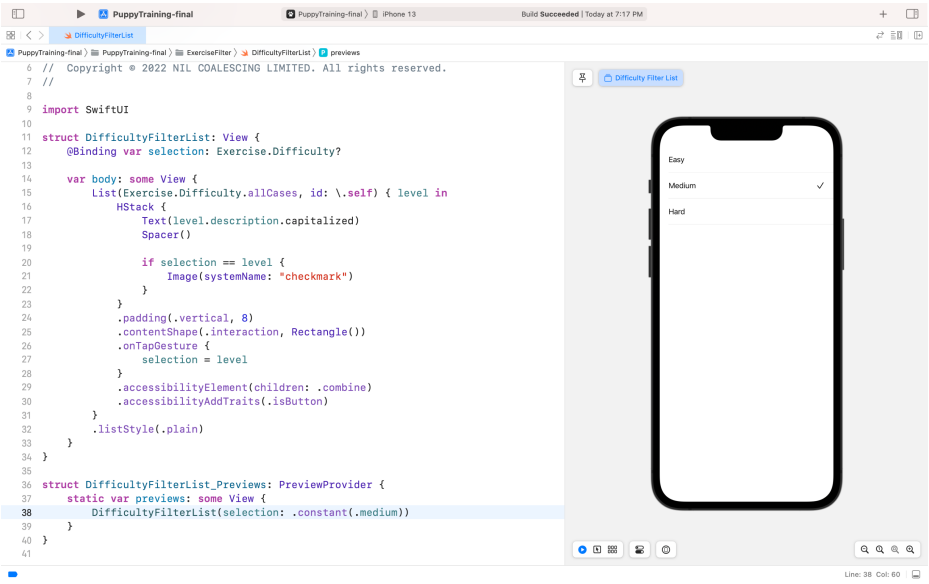
The user will be able to select the difficulty in the filter by tapping on a row. To make sure that the tap gesture is activated when the user taps anywhere in the row and not just on the text or the image, we'll apply the `contentShape(.interaction, Rectangle())` modifier.

```
struct DifficultyFilterList: View {
    @Binding var selection: Exercise.Difficulty?

    var body: some View {
        List(Exercise.Difficulty.allCases, id: \.self) { level in
            HStack {
                Text(level.description.capitalized)
                Spacer()

                if selection == level {
                    Image(systemName: "checkmark")
                }
            }
            .padding(.vertical, 8)
            .contentShape(.interaction, Rectangle())
            .onTapGesture {
                selection = level
            }
            .accessibilityElement(children: .combine)
            .accessibilityAddTraits(.isButton)
        }
        .listStyle(.plain)
    }
}
```

To be able to preview the `DifficultyFilterList` view in the canvas, we have to

slightly modify the preview provider. The `DifficultyFilterList` has to accept a binding to a difficulty level, but we can create a constant one just for the preview purposes.

```
struct DifficultyFilterList_Previews: PreviewProvider {
    static var previews: some View {
        DifficultyFilterList(selection: .constant(.medium))
    }
}
```



Screenshot of Xcode previews showing the DifficultyFilterList view

Now we can define the filter view itself. We'll create another file called `ExerciseFilterView.swift` also with the SwiftUI View template. The `ExerciseFilterView` will receive the `ExerciseFilterState` object from UIKit, so we'll add a `filterState` property to the view struct. The property has to be marked with the `@ObservedObject` wrapper, because the view has to update when the selection changes to show the checkmark.

The `DifficultyFilterList` that we created earlier will be wrapped into a `Naviga-`

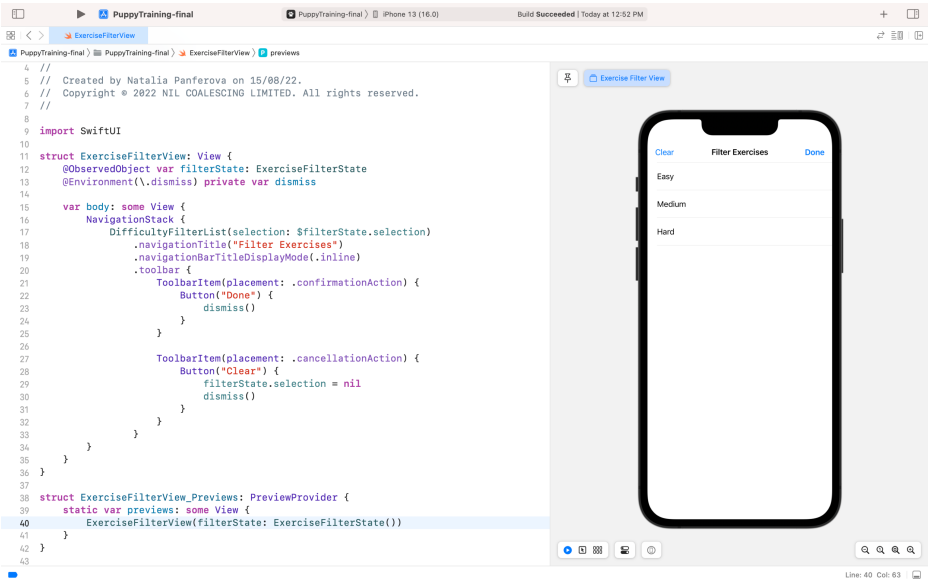
tionStack or NavigationView if targeting iOS 15. Since in this case the navigation bar comes from the SwiftUI layer, we can set it up using SwiftUI APIs. We can add the title using the navigationTitle() modifier, set its display mode to inline and add some navigation bar buttons. When we add buttons using the toolbar() modifier in SwiftUI, they go into the navigation bar by default.

```
struct ExerciseFilterView: View {
    @ObservedObject var filterState: ExerciseFilterState
    @Environment(\.dismiss) private var dismiss

    var body: some View {
        NavigationStack {
            DifficultyFilterList(selection: $filterState.selection)
                .navigationTitle("Filter Exercises")
                .navigationBarTitleDisplayMode(.inline)
                .toolbar {
                    ToolbarItem(placement: .confirmationAction) {
                        Button("Done") {
                            dismiss()
                        }
                    }

                    ToolbarItem(placement: .cancellationAction) {
                        Button("Clear") {
                            filterState.selection = nil
                            dismiss()
                        }
                    }
                }
        }
    }
}
```

Note that we can dismiss the sheet from within the SwiftUI view using a SwiftUI API too. The dismiss action is injected into the environment by the framework and can be used to dismiss modals or pop views from the navigation stack.

To preview ExerciseFilterView in the canvas we simply need to pass it an ExerciseFilterState object instance in the preview provider.

```
struct ExerciseFilterView_Previews: PreviewProvider {
    static var previews: some View {
        ExerciseFilterView(filterState: ExerciseFilterState())
    }
}
```



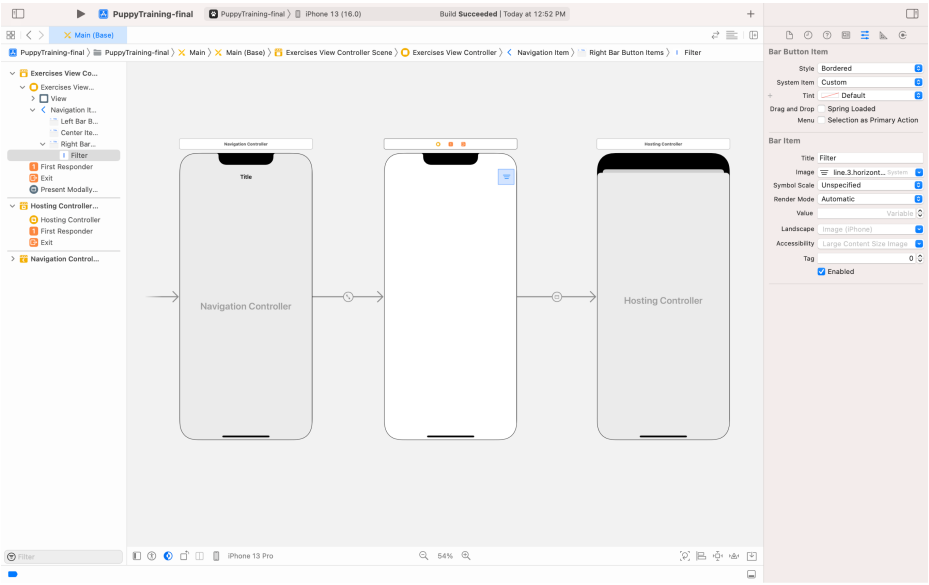
Screenshot of Xcode previews showing the ExerciseFilterView view

Add a Hosting Controller in the storyboard

After we defined our SwiftUI view, we need to add a Hosting Controller to present the SwiftUI hierarchy from the UIKit layer of the app. We are going to do that in the `Main.storyboard` file.

First, we'll add a Bar Button Item to the `ExercisesViewController` in the storyboard that will tigger the sheet presentation. We'll call it `Filter` and set the `line.3.horizontal.decrease` symbol as the image.

Then we'll drag a Hosting Controller from the object library to the canvas and add a `Present Modally` segue from the filter button to the Hosting Controller.

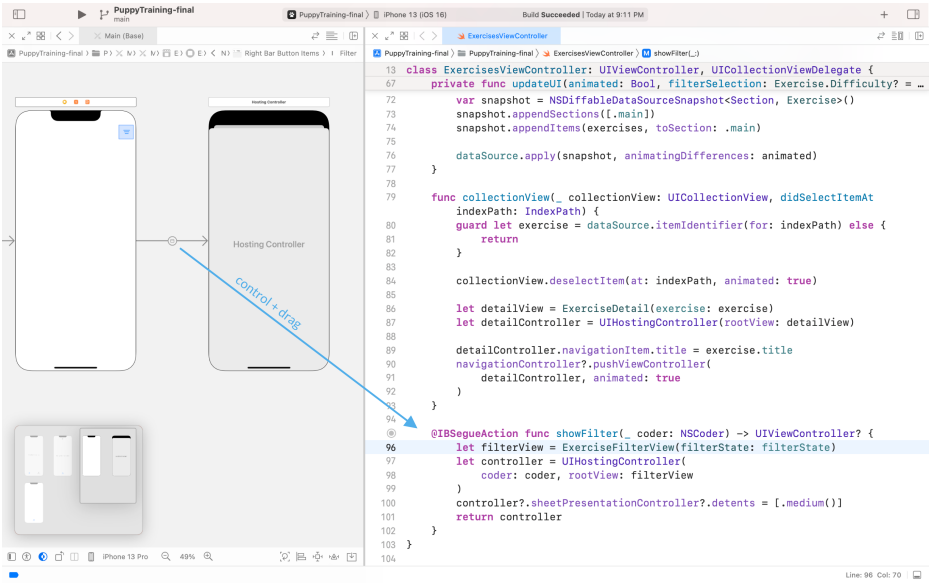


Screenshot of Xcode showing the Main.storyboard file with a Hosting Controller in the canvas

We still need to set our `ExerciseFilterView` as the root view of the Hosting Controller we just added. We can do that in the segue action method. We'll open the `ExercisesViewController.swift` in the assistant editor and control-drag from the segue to an area inside the controller. Using the popup that appears, we'll create a Segue Action and call it `showFilter`. Inside this action we can prepare the Hosting Controller for presentation.

We will create an instance of the `ExerciseFilterView` and pass it the `filterState` stored in the view controller. Then we will initialize a `UIHostingController` with the `coder` passed to the action and the SwiftUI view. Once the controller is created, we can customize it the way we would a regular `UIViewController`. Here we are going to set a `medium` detent on its `sheetPresentationController` property so that the filter sheet only covers half of the screen.

```
class ExercisesViewController: UIViewController,
    UICollectionViewDelegate {
    ...

    @IBSegueAction func showFilter(
        _ coder: NSCoder
    ) -> UIViewController? {
        let filterView = ExerciseFilterView(
            filterState: filterState
        )
        let controller = UIHostingController(
            coder: coder, rootView: filterView
        )
        controller?.sheetPresentationController?.detents = [
            .medium()
        ]
        return controller
    }
]
```



Screenshot of Xcode with the storyboard and the ExercisesViewController open in assistant editor

If we run the app now and press the filter button the sheet will appear as expected. The SwiftUI side already works correctly and tapping on a difficulty level will add a checkmark to the row. But the UIKit part is not set up to react to the changes in the

filter state yet.

Update UIKit layer from SwiftUI

The last thing left to do for the filter to function is to update the collection view to reflect the selection. We are going to use the Combine framework to subscribe to changes in the published property of the observable object and trigger a UI refresh.

We need to change the `updateUI()` method defined in the `ExercisesViewController` to accept a `filterSelection` parameter. It will use the selection value to filter the exercises provided by the `ExercisesController` and update the data source with a new snapshot.

```
class ExercisesViewController: UIViewController,
    UICollectionViewDelegate {
    ...

    private func updateUI(
        animated: Bool, filterSelection: Exercise.Difficulty? = nil
    ) {
        let exercises = exercisesController.exercises
            .filter { exercise in
                filterSelection
                    .map { $0 == exercise.difficulty } ?? true
            }

        var snapshot = NSDiffableDataSourceSnapshot<
            Section, Exercise
        >()

        snapshot.appendSections([.main])
        snapshot.appendItems(exercises, toSection: .main)

        dataSource.apply(snapshot, animatingDifferences: animated)
    }
}
```

To observe the changes in filter selection we will add a subscriber to the `selection` property of the `ExerciseFilterState` in the view controller. We need to import Combine and add a `cancellable` property that will hold the subscription. Inside the `viewDidLoad()` method we will create a subscription that triggers the UI update with

animation when selection changes.

```
import UIKit
import SwiftUI
import Combine

class ExercisesViewController: UIViewController,
    UICollectionViewDelegate {

    private var cancellable: AnyCancellable?

    override func viewDidLoad() {
        ...

        cancellable = filterState.$selection
            .sink { [weak self] selection in
                if selection != self?.filterState.selection {
                    self?.updateUI(
                        animated: true,
                        filterSelection: selection
                    )
                }
            }
    }

    ...

}
```

Now the UIKit controller and the SwiftUI filter view are fully connected and the changes in the filter are properly reflected in the collection view.

Using Combine is just one of the possible ways to react to changes in the observable object on the UIKit side. I find it to be the most concise in this case, but you could implement what works best within your app pattern. You could observe the changes to the property with `didSet` and post a notification or call a delegate method to notify the controller about the change.

Subclassing UIHostingController

This subchapter is not included in the free sample.

Further reading links

- ObservableObject protocol - https://developer.apple.com/documentation/combine/observableobject
- ObservedObject property wrapper - https://developer.apple.com/documentation/swiftui/observedobject
- Binding property wrapper - https://developer.apple.com/documentation/swiftui/binding
- DismissAction - https://developer.apple.com/documentation/SwiftUI/DismissAction
- Combine framework - https://developer.apple.com/documentation/combine