NAVIGATION API IN SWIFTUI

MOHAMMAD AZAM

Navigation has always been a pain point in SwiftUI. Unlike React and Flutter which provides a centralized navigation engine, SwiftUI navigation is tightly coupled with the views. Fortunately, iOS 16 introduces a brand new Navigation API for SwiftUI applications, which allows you to setup global navigation. Navigation API also supports programmatic navigation, giving complete control in the hands of the developer.

Before iOS 16, if you wanted to navigate from one view to the other you will create a NavigationView and use a NavigationLink to go to the destination. This is shown in **Listing 1**.

```
struct ContentView: View {
    var body: some View {
        NavigationView {
            NavigationLink("Go to details") {
                Text("Details")
            }
        }
    }
}
```

Listing 1: NavigationView and NavigationLink

NavigationView is deprecated in iOS 16 and has been replaced by NavigationStack. The code in **Listing 1** can be implemented as NavigationStack as shown in **Listing 2**.

```
struct ContentView: View {
    var body: some View {
        NavigationStack {
            NavigationLink(“Go to details") {
                Text("Details")
            }
        }
    }
}
```

Listing 2: NavigationStack and NavigationLink

This is the most simplest use case for NavigationStack. We simply replaced the NavigationView with NavigationStack, while keeping the same behavior. If you

tap on the link it will simply take you to the Text view with content set as "**Details**".

Programmatic Navigation

The main benefit you receive from using NavigationStack is the power of programmatic navigation. Take a look at **Listing 3**.

```
struct ContentView: View {
    var body: some View {
        NavigationStack {
            VStack {
                NavigationLink("Link 1", value: "Hello World")
            }.navigationDestination(for: String.self) { stringValue in
                Text(stringValue)
            }
        }
    }
}
```

Listing 3: Navigation using navigationDestination

In **Listing 3** we performed navigation based on the type of value provided in the NavigationLink. The **navigationDestination** is the new method added in SwiftUI, which returns the destination view based on the data type of the value in the NavigationLink. When NavigationLink is pressed, it is handled by the provided navigationDestination method, which returns the view.

Take a look at **Listing 4**, where we are using different value types for the NavigationLink and providing several navigationDestination implementations.

```
struct Movie: Hashable {
    let name: String
    let year: Int
}

struct ContentView: View {
    var body: some View {
        NavigationStack {
            VStack {
                NavigationLink("Link 1", value: "Hello World")
                NavigationLink("Link 2", value: 100)
                NavigationLink("Link 3", value: Movie(name: "Lord of the
Rings", year: 2001))
```

```
            }
            .navigationDestination(for: String.self) { stringValue in
                Text(stringValue)
            }
            .navigationDestination(for: Int.self) { intValue in
                        Text("\(intValue)")
            }
            .navigationDestination(for: Movie.self) { movie in
                Text(movie.name)
            }

        }
    }
}
```

Listing 4: Multiple navigationDestinations based on the value type

For each value type we have implemented a designated navigationDestination method. If you have a NavigationLink with a value whose type is not available in navigationDestination, then that navigation will never fire.

Another important point to remember is to add the navigationDestination modifier on the correct view. Let's take a look at couple of examples, where the navigation does not work.

```
struct ContentView: View {
    var body: some View {
        NavigationStack {
            VStack {
                NavigationLink("Link 1", value: "Hello World")
            }
        }.navigationDestination(for: String.self) { stringValue in
            Text(stringValue)
        }
    }
}
```

Listing 5: NavigationStack

In **Listing 5** we are placing the navigationDestination modifier on the NavigationStack itself. If you run the code, you will notice that it will not work and the navigation will not take you anywhere. The navigationDestination should be applied to views inside the NavigationStack and not on the NavigationStack itself.

Let's take a look at another example, where the navigationDestination modifier is applied incorrectly.

```
struct ContentView: View {
    var body: some View {
        NavigationStack {
            List(1...100, id: \.self) { index in
                NavigationLink("Item \(index)", value: index)
                    .navigationDestination(for: Int.self) { intValue in
                        Text("\(intValue)")
                    }
            }
        }
    }
}
```

Listing 6: Applying navigationDestination on the NavigationLink inside the List

In **Listing 6**, navigationDestination is applied to the NavigationLink inside the List. This means that the destination is only initialized for the items currently displayed on the screen and not for the rows that are off screen. Another thing to note is that navigationDestination is triggered 100 times for each row inside the List view. This can easily be avoided by placing navigationDestination higher up in the view hierarchy as shown in **Listing 7**.

```
struct ContentView: View {
    var body: some View {
        NavigationStack {
            List(1...100, id: \.self) { index in
                NavigationLink("Item \(index)", value: index)
            }
            .navigationDestination(for: Int.self) { intValue in
                Text("\(intValue)")
            }
        }
    }
}
```

Listing 7: Applying navigationDestination the List

The main benefit of moving the modifier up to the List is that the NavigationStack can see the navigation destination regardless of the List view scroll position.

Next time you are adding navigationDestination modifier make sure to put it on views up in the hierarchy of the view tree. This will allow them to see the destinations without having to scroll etc.

NavigationStack keeps track of user's navigation history using path. When the app starts, the path is empty, as the user moves from one view to the other, NavigationStack pushes new values into the path, building the user navigation history. Let's take a look at an example in Listing 8, which demonstrates a master-detail view.

```
struct ContentView: View {

    let movies = [Movie(name: "Lord of the Rings", year: 2001), Movie(name:
"Superman", year: 2010), Movie(name: "Spiderman", year: 2012), Movie(name:
"Batman", year: 2016)]

    var body: some View {
        NavigationStack {
            List(movies, id: \.name) { movie in
                NavigationLink(movie.name, value: movie)
            }
            .navigationDestination(for: Movie.self) { movie in
                MovieDetail(movie: movie)
            }
        }
    }
}
```

Listing 8: Master detail view

When the user navigates back to a view then the path is popped out. If all the values in the path are popped out then the user ends up on the root view of the application. This technique of removing everything from the NavigationStack path collection can allow you to create unwind segues.

Path Binding in Navigation Stack

As explained earlier NavigationStack pushes a navigation path onto a stack for view history management. You can also control the paths by using a bindable expression in NavigationStack. This will allow you to programmatically push/pop different paths into the collection and provide custom navigation.

We will start by creating a local state variable of type Movie array called path and pass it to the NavigationStack as shown in Listing 9.

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

```
NavigationStack(path: $path) {
 // code here
}
```

Listing 9: Custom path management

Since, the path is defined as a collection of Movie type, any navigation conforming to the movie navigation destination will be placed in the path array. You can even assign a particular movie to the path and it will perform the navigation. This is shown in **Listing 10**.

```
struct ContentView: View {

    @State private var path: [Movie] = []

    let movies = [Movie(name: "Lord of the Rings", year: 2001,
isMovieOfTheDay: true), Movie(name: "Superman", year: 2010), Movie(name:
"Spiderman", year: 2012), Movie(name: "Batman", year: 2016)]

    var body: some View {
        NavigationStack(path: $path) {
            // code here
            VStack {
                List(movies, id: \.name) { movie in
                    NavigationLink(movie.name, value: movie)
                }

                Button("Show Featured Movie") {
                    guard let featuredMovie = movies.first(where:
{ $0.isMovieOfTheDay }) else { return }
                    path = [featuredMovie]
                }

            }.navigationDestination(for: Movie.self) { movie in
                MovieDetails(movie: movie)
            }
            .navigationTitle("Movies")
        }
    }
}
```

Movie model is updated to store a new property called **isMovieOfTheDay**. By default this property is set to false on all the movies, except for "**Lord of the Rings**". When the user taps on the "**Show Featured Movie**" button, we assign movie of the day to the path. This triggers navigationDestination method and performs the navigation to the MovieDetails screen, passing the movie in the

path. Finally, the movie of the day i.e “Lord of the Rings” is displayed on the MovieDetails screen.

At present, our path is only available in one view. This is because we have used private local state and not the global state. In the next section, we will learn how to access path globally in other views so they have an opportunity to control the navigation.

Accessing Path Globally

The main reason to access and manipulate path globally is to give control to other views to perform navigation. You may have a scenario, where you are multi-level down in the view hierarchy and want to instantly move to the root. This behavior is called unwind segue or pop to root.

The first step is to implement a class, which will hold the path. The instance of this class will act as the global state and will be injected as an environment object. The implementation of AppState class is shown in **Listing 11**.

```
class AppState: ObservableObject {
    @Published var path: [Movie] = []
}
```

Listing 11: AppState global object

The AppState class consists of a path variable, which keeps track of the navigation history. The NavigationStack uses the global environment object as a path as shown in **Listing 12**.

```
struct ContentView: View {

    @EnvironmentObject var appState: AppState

    let movies = [Movie(name: "Lord of the Rings", year: 2001), Movie(name:
"Superman", year: 2010), Movie(name: "Spiderman", year: 2012), Movie(name:
"Batman", year: 2016)]

    var body: some View {
```

```
        NavigationStack(path: $appState.path) {
            List(movies, id: \.name) { movie in
                NavigationLink(movie.name, value: movie)
            }
            .navigationDestination(for: Movie.self) { movie in
                MovieDetail(movie: movie)
            }
        }
    }
}
```

Listing 12: AppState global object

Now, you can access the NavigationStack path through the environment object in any view. Keep in mind that when new path is added or removed, root view is rendered again since it is observing changes through the use of NavigationStack path binding.

At present the NavigationStack is configured to work with paths containing Movie type but in real world you may have other types too. In the next section, you will learn how to use NavigationPath to support programmatic navigation for multiple types.

NavigationPath

NavigationPath is a type erased list of data representing the content of a navigation stack. This means that instead of holding a single type, NavigationPath can allow navigation to be based on several different types. Let's take a look at a an example in **Listing 13**.

```
struct ContentView: View {

    let movies  = [Movie(name: "Lord of the Rings"), Movie(name: "Batman"),
Movie(name: "Spiderman")]

    @State private var path = NavigationPath()

    var body: some View {
        NavigationStack(path: $path) {
            VStack {
                Button("Go to a random movie") {

                    guard let randomMovie = movies.randomElement() else {
                        return
                    }
```

```
                    path.append(randomMovie)

                }.buttonStyle(.borderedProminent)
            }.navigationDestination(for: Movie.self) { movie in
                Text(movie.name)
            }
        }
    }
}
```

Listing 13: Appending path using NavigationPath

Listing 13 introduces the NavigationPath instance as a state variable. The path is passed to the NavigationStack and now NavigationStack will keep track of history of the user's navigation. When the "**Go to a random movie**" button is pressed, we find a random movie and appends it to the path variable. This causes the navigationDestination to fire and perform the navigation based on the type of object placed in the path instance.

Let's add another type to our app, which will allow us to perform navigation on the Actor type as shown in **Listing 14**.

```
struct ContentView: View {

    let movies  = [Movie(name: "Lord of the Rings"), Movie(name: "Batman"),
Movie(name: "Spiderman")]
    let actors = [Actor(name: "Tom Hanks"), Actor(name: "Kate Winslet"),
Actor(name: "Bill Murray")]

    @State private var path = NavigationPath()

    var body: some View {
        NavigationStack(path: $path) {
            VStack {
                Button("Go to a random movie") {

                    guard let randomMovie = movies.randomElement() else {
                        return
                    }

                    path.append(randomMovie)

                }.buttonStyle(.borderedProminent)
            }.navigationDestination(for: Movie.self) { movie in
                VStack {
                    Text(movie.name)
                    Button("Go to random actor") {
                        guard let randomActor = actors.randomElement() else {
```

```
                        return
                    }
                    path.append(randomActor)
                }
            }
        }
        .navigationDestination(for: Actor.self) { actor in
            Text(actor.name)
        }
    }
}
```

Listing 14: Adding navigation for Actor type

The basic idea is the same. We added a navigationDestination for type Actor. Once the user taps on the "**Go to random actor**" button, we append an actor to the path, which causes the navigation to take place.

Currently, all our views are in the same file and that is why they are able to access the path variable. If they were in separate files then you can use EnvironmentObject to hold a global path variable as explained in the section "**Accessing Path Globally**".

You can also perform an unwind segue navigation, which will allow you to jump from a nested view directly to the root view. This is shown in **Listing 15**.

```
.navigationDestination(for: Actor.self) { actor in
            VStack {
                Text(actor.name)
                Button("Go to root") {
                    path.removeLast(path.count)
                }
            }
        }
```

Listing 15: Go to root from a nested view

In **Listing 15**, we used the removeList function on the NavigationPath instance to remove all the paths that were added by NavigationStack. This will take us instantly back to the root view.

Popping to the root was also possible in SwiftUI with NavigationView but it was much more difficult. I have a video on Unwind Segues in SwiftUI that you can check out here. NavigationStack makes it super easy to perform pop to root behavior, without the hassle of passing additional binding parameters to the nested views.

NavigationSplitView

NavigationStack allows your views to be stacked on top of each other, which is the desired behavior for iPhone apps but on larger devices like iPad, Mac and iPhones in portrait mode, you may want to utilize the larger screen real estate. In iOS 16 Apple introduced **NavigationSplitView** to divide the screen into a sidebar view and detail view.

Listing 16 shows the basic setup of NavigationSplitView.

```
struct ContentView: View {
    var body: some View {
        NavigationSplitView {
            Text("SideBar")
        } detail: {
            Text("Detail")
        }

    }
}
```

Listing 16: Basic setup of NavigationSplitView

The result is shown in **Figure 1**.

The result is shown in **Figure 1**.

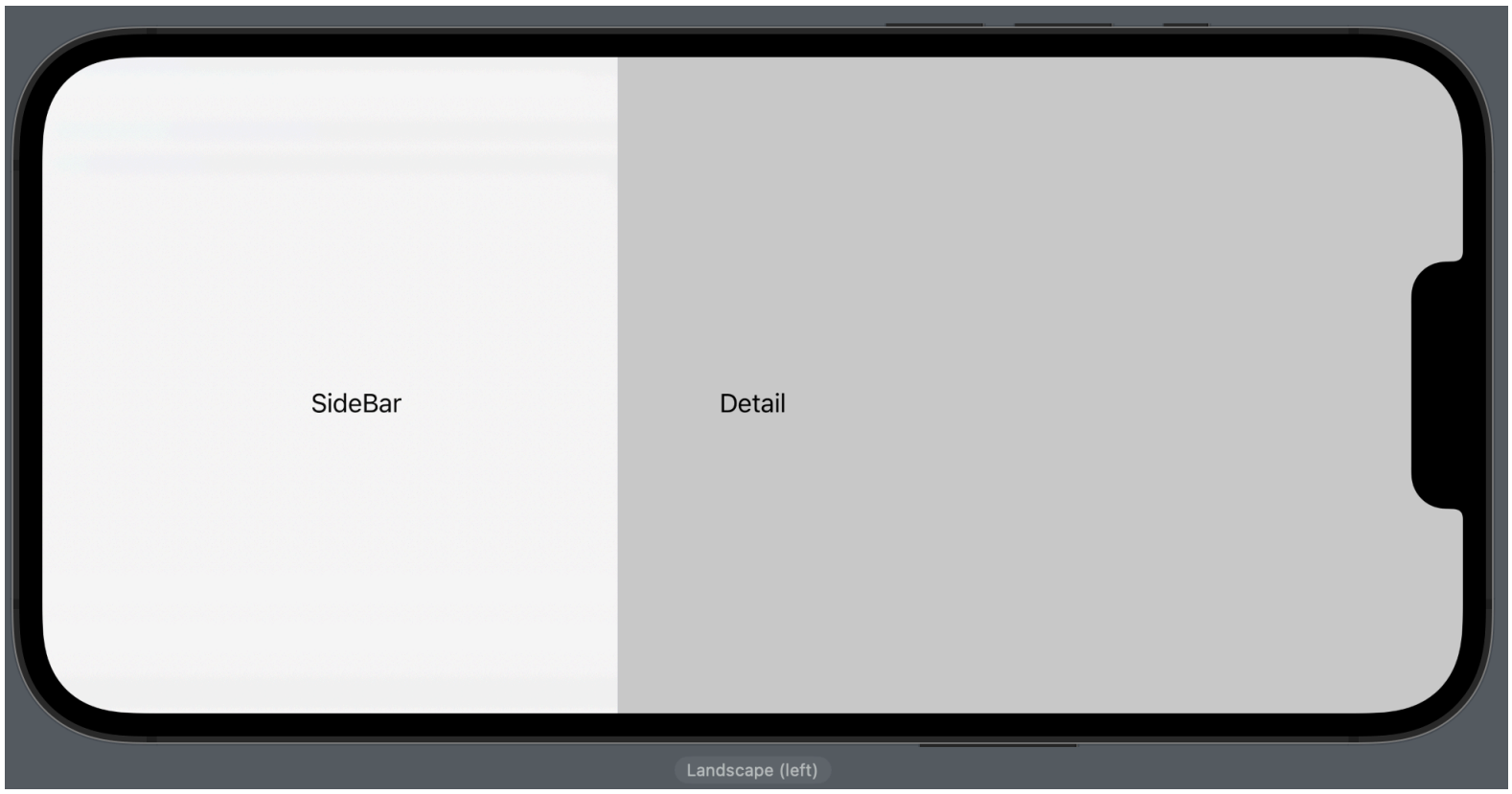

Figure 1: Basic setup of NavigationSplitView

The sidebar is used to display a list of rows and the detail view is for used for displaying the details of the selected content from the sidebar. **Listing 17** shows how to populate the sidebar with a list of genre.

```
enum Genre: String, Hashable, CaseIterable {
    case action = "Action"
    case horror = "Horror"
    case fiction = "Fiction"
    case kids = "Kids"
}


struct ContentView: View {

    @State private var selectedGenre: Genre?


    var body: some View {
        NavigationSplitView {
            List(Genre.allCases, id: \.self, selection: $selectedGenre)
{ genre in
                NavigationLink(genre.rawValue, value: genre)
            }
        } detail: {

           Text(“Detail”)
        }
    }
}
```

Listing 17: Populating list of genre in sidebar

The List is passed a $selectedGenre binding, which is populated when a new genre is selected. It also automatically highlights the selected row, giving it a nice visual effect as shown in **Figure 2**.

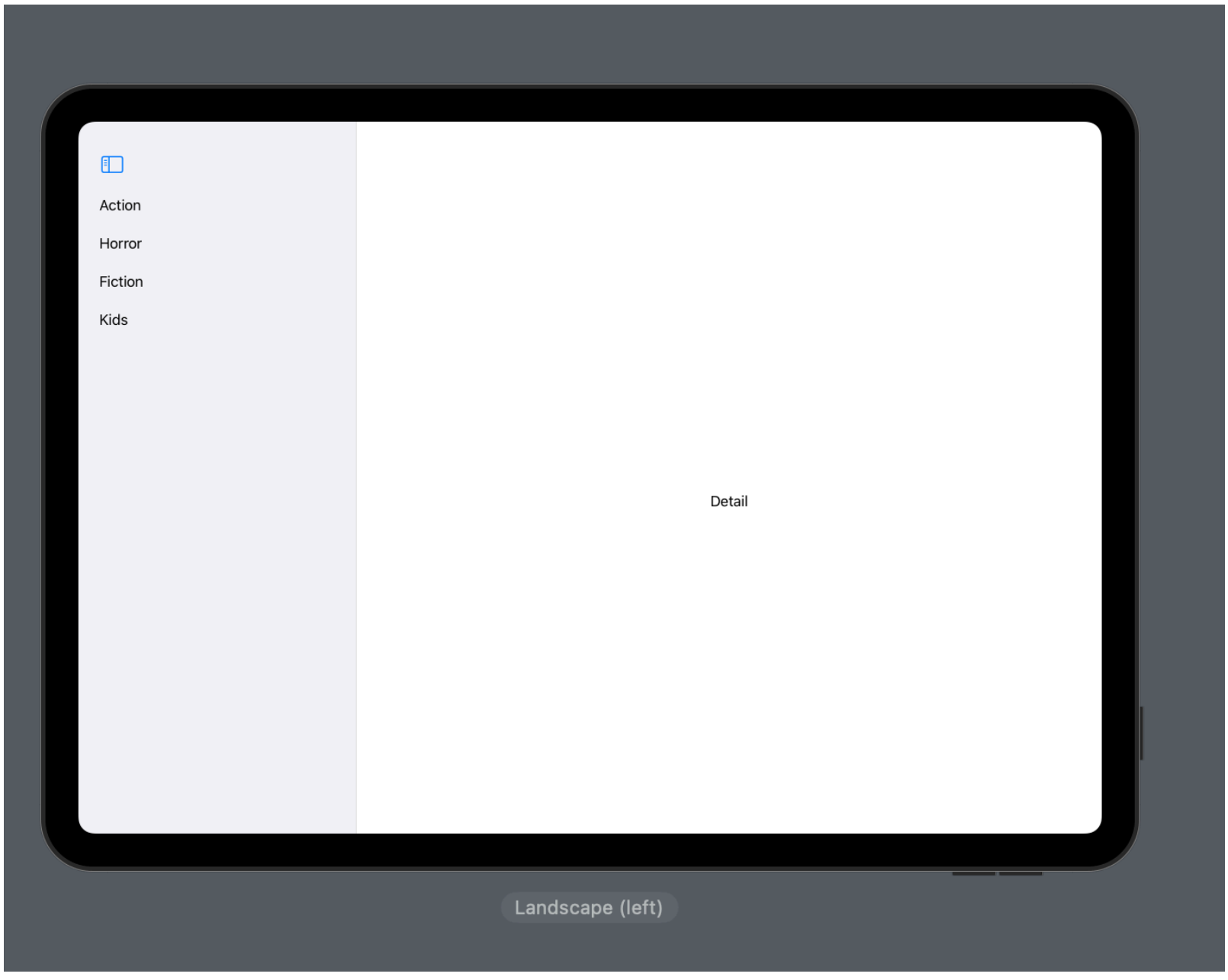

Figure 2: List selection binding in sidebar

We can update our detail view to display the movies based on the selected genre. **Listing 18** shows the implementation of the detail view.

```
struct ContentView: View {

    @State private var selectedGenre: Genre?

    let movies = [Movie(name: "Superman", genre: .action), Movie(name: "28
Days Later", genre: .horror), Movie(name: "World War Z", genre: .horror),
Movie(name: "Finding Nemo", genre: .kids)]
```

```
    let columns: [GridItem] = [.init(.fixed(400)), .init(.fixed(400))]

    var body: some View {
        NavigationSplitView {
            List(Genre.allCases, id: \.self, selection: $selectedGenre)
{ genre in
                NavigationLink(genre.rawValue, value: genre)
            }
        } detail: {

            let filteredMovies = movies.filter { $0.genre == selectedGenre }

            LazyVGrid(columns: columns) {
                ForEach(filteredMovies, id: \.name) { movie in
                    Text(movie.name)
                        .frame(width: 200, height: 200)
                        .foregroundColor(.white)
                        .background(content: {
                            Color.gray
                        })
                }
            }
        }
    }
}
```

Listing 18: Detail view showing movies based on genre selection

The result is shown in **Figure 3**.

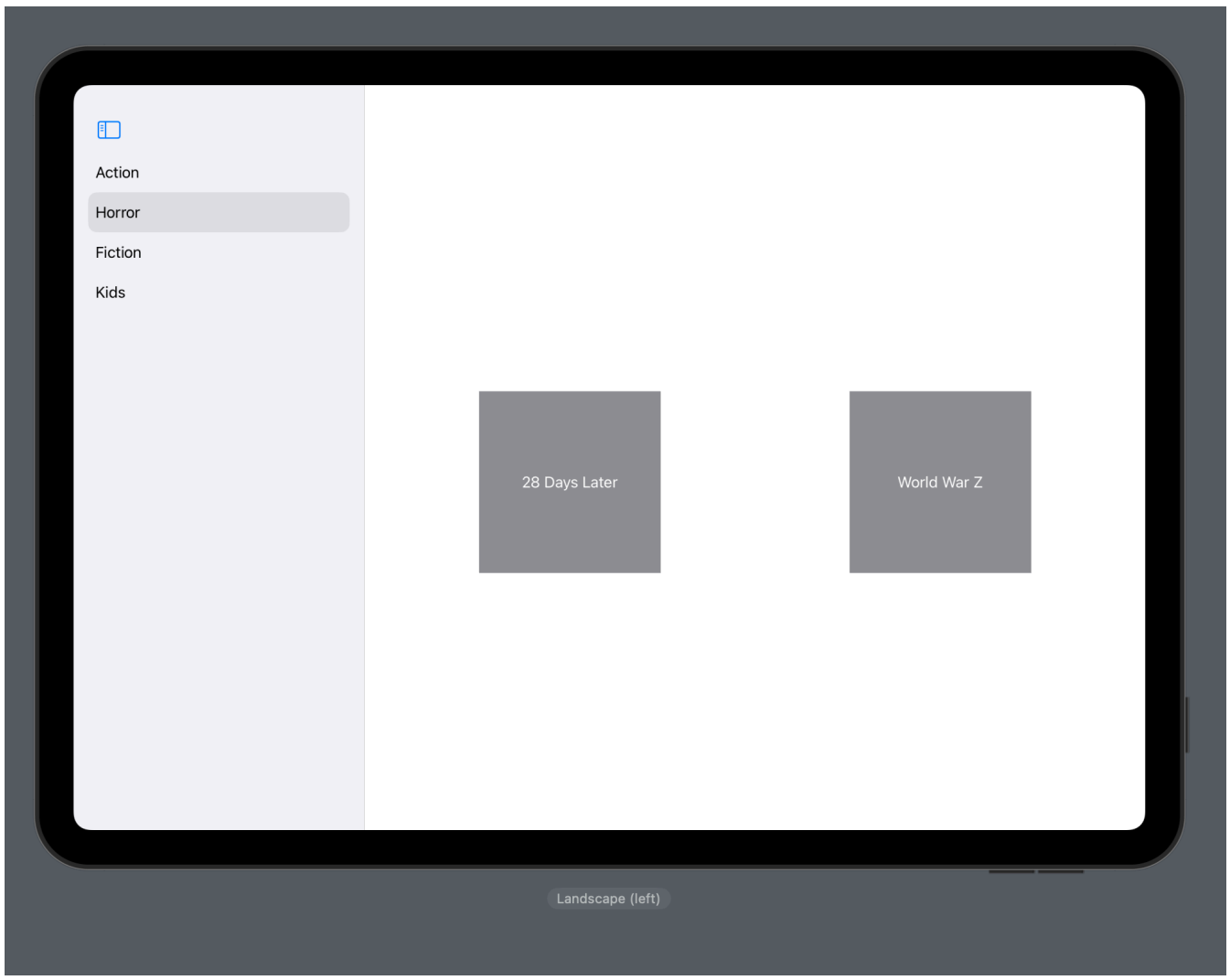

Figure 3: Detail view showing movies based on genre selection

Once the genre is selected, we filter the movies based on the selected genre and then display it in a LazyVGrid.

NavigationSplitView is also capable of displaying three columns. This includes sidebar, content and detail. The basic idea is the same, sidebar column will display a list of genres. When the user select a particular genre then the content column (middle column) will display all the movies based on the selected genre. And finally, when the user selects the movie then we display a the details about the movie in the detail column.

The implementation is shown in **Listing 19**.

```
struct ContentView: View {

    @State private var selectedGenre: Genre?
    @State private var selectedMovie: Movie?

    let movies = [Movie(name: "Superman", genre: .action), Movie(name: "28
Days Later", genre: .horror), Movie(name: "World War Z", genre: .horror),
Movie(name: "Finding Nemo", genre: .kids)]

    let columns: [GridItem] = [.init(.fixed(400)), .init(.fixed(400))]

    var body: some View {

        NavigationSplitView {
            List(Genre.allCases, id: \.self, selection: $selectedGenre)
{ genre in
                NavigationLink(genre.rawValue, value: genre)
            }.navigationTitle("Genres")
        } content: {

            let filteredMovies = movies.filter { $0.genre == selectedGenre }

            List(filteredMovies, id: \.name, selection: $selectedMovie)
{ movie in
                NavigationLink(movie.name, value: movie)
            }.navigationTitle(selectedGenre?.rawValue ?? "Movies")

        } detail: {
                VStack(alignment: .center) {
                    Text(selectedMovie?.name ?? "")
                        .font(.largeTitle)
                        .navigationTitle(selectedMovie?.name ?? "")
                }.frame(maxWidth: .infinity, maxHeight: .infinity)
                    .background {
                        Color.accentColor
                    }
```

```
            }
        }
}
```

Listing 19: Three column layout in NavigationSplitView

The result is shown in **Figure 4**.

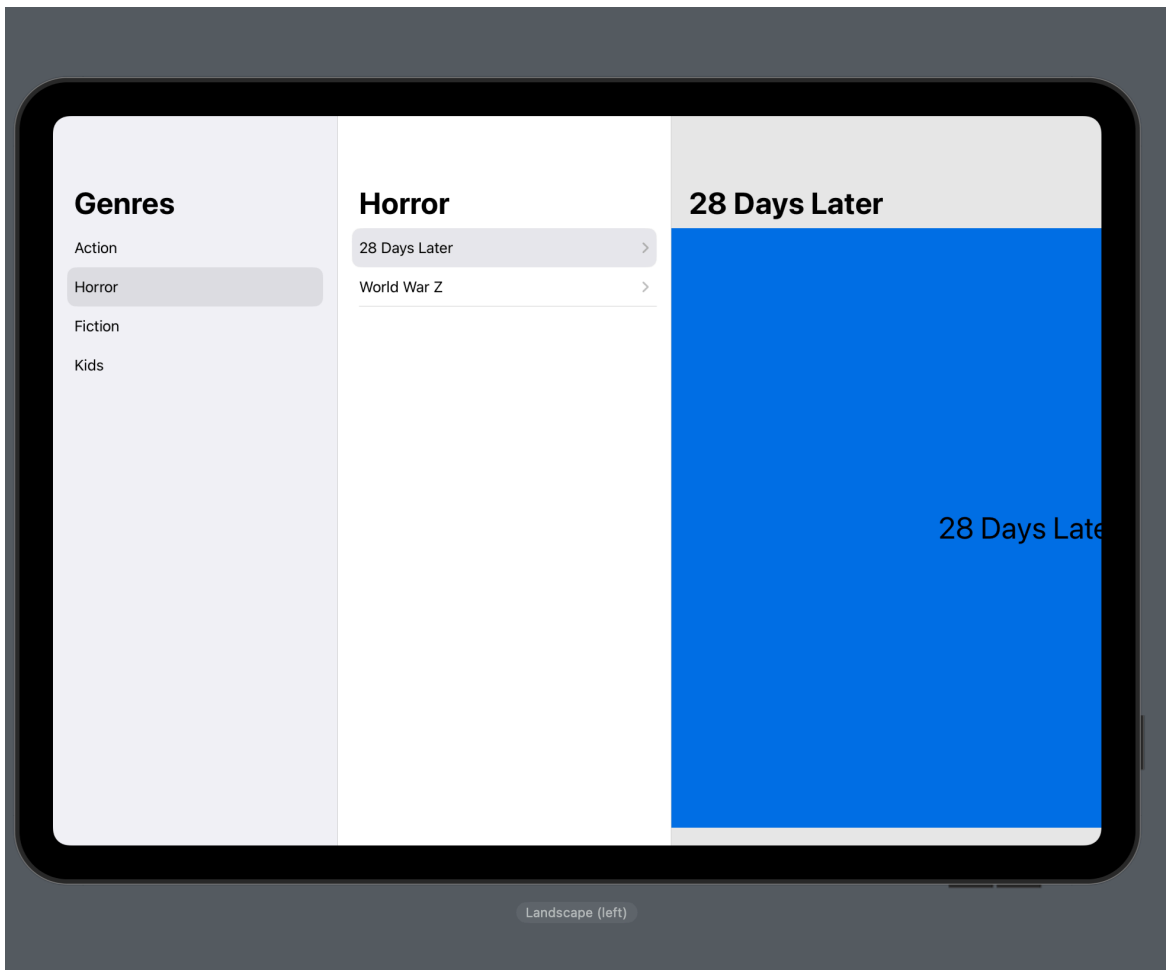

Figure 4: Three column layout in NavigationSplitView

NavigatonSplitView also allows the developer to be in complete control and you can perform programmatic navigation by simply setting the binding. This is shown in **Listing 20**.

```
func showMovieOfTheDay() {
        guard let movieOfTheDay = movies.first(where: { $0.movieOfTheDay })
else { return }
        selectedMovie = movieOfTheDay
    }
```

Listing 20: Programmatic navigation based on the selected movie

One great thing about using the NavigationSplitView with List views is that SwiftUI will automatically adapt single stack navigation on iPhone.

This is shown in **Figure 5**.

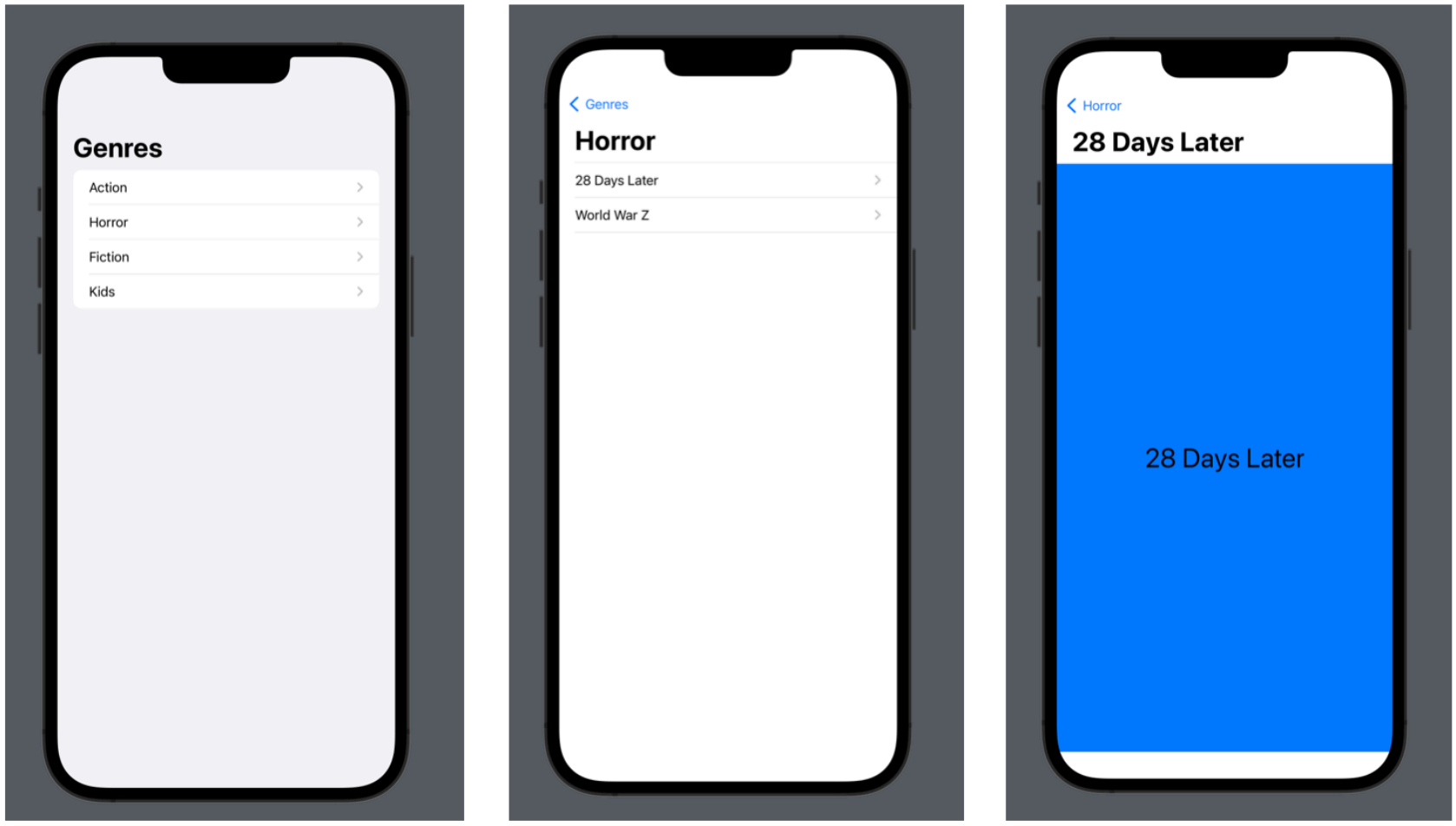

Figure 5: Single stack navigation on iPhone using NavigationSplitView and List

Nested NavigationStack in NavigationSplitView

Apart from providing the sidebar, content and detail view, NavigationSplitView can also contain NavigationStack, allowing you to create scenarios with much deeper navigation structure.

Let's start with the basic sidebar and detail application as shown in **Listing 21**.

```
struct ContentView: View {

    @State private var selection: Int?

    var body: some View {
        NavigationSplitView {
            List(1...20, id: \.self, selection: $selection) { index in
                Text("\(index)")
            }
        } detail: {
            ZStack {
                if let selection {
                    Text("\(selection)")
                }
```

```
            }
        }
    }
}
```

Listing 21: Basic sidebar and detail using NavigationSplitView

Sidebar displays a list of numbers based on the provided range. When the number is selected, detail view is updated to show the selection. This is shown in **Figure 6**.

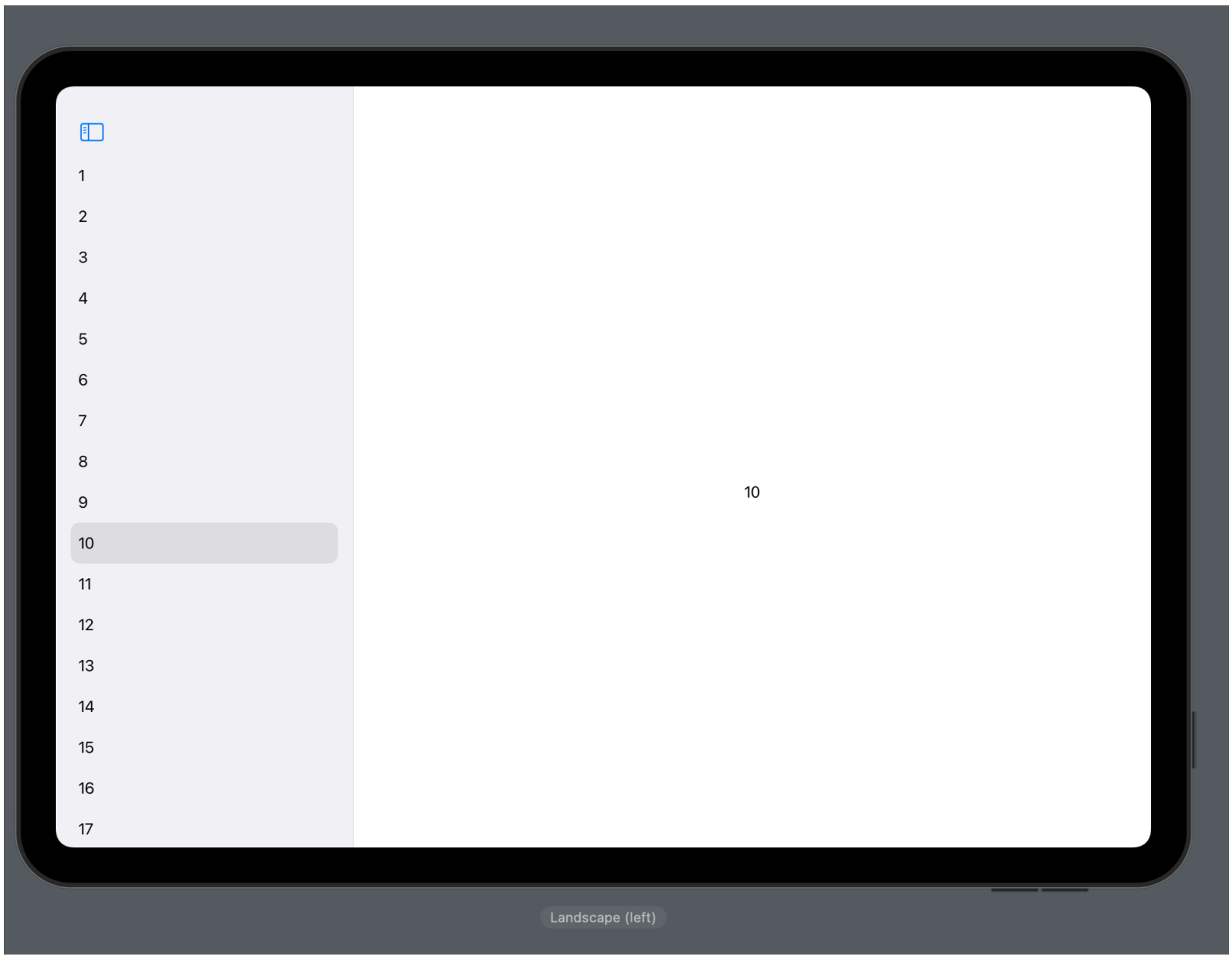

Figure 6: Basic sidebar and detail using NavigationSplitView

Let’s update our detail view to show the number of Rectangles based on our selection. The implementation is shown in **Listing 22**.

```
struct ContentView: View {

    @State private var selection: Int?
```

```
    let columns =
[GridItem(.flexible()),GridItem(.flexible()),GridItem(.flexible()),GridItem(.f
lexible())]

    var body: some View {
        NavigationSplitView {
            List(1...20, id: \.self, selection: $selection) { index in
                Text("\(index)")
            }
        } detail: {

                ZStack {
                    if let selection {
                        LazyVGrid(columns: columns, alignment: .center) {
                            ForEach(1...(selection), id: \.self) { index in
                                VStack {

                                    Rectangle()
                                        .fill(.green)

                                }.frame(width: 100, height: 100)
                            }
                        }
                        .padding(.horizontal)
                        .frame(maxWidth: .infinity, maxHeight: .infinity)
                        .background {
                            Color.yellow
                        }
                        .navigationTitle("\(selection)")
                    }
                }

        }

    }
}
```

Listing 22: Displaying rectangles based on the sidebar selection

The result is shown in **Figure 7.**

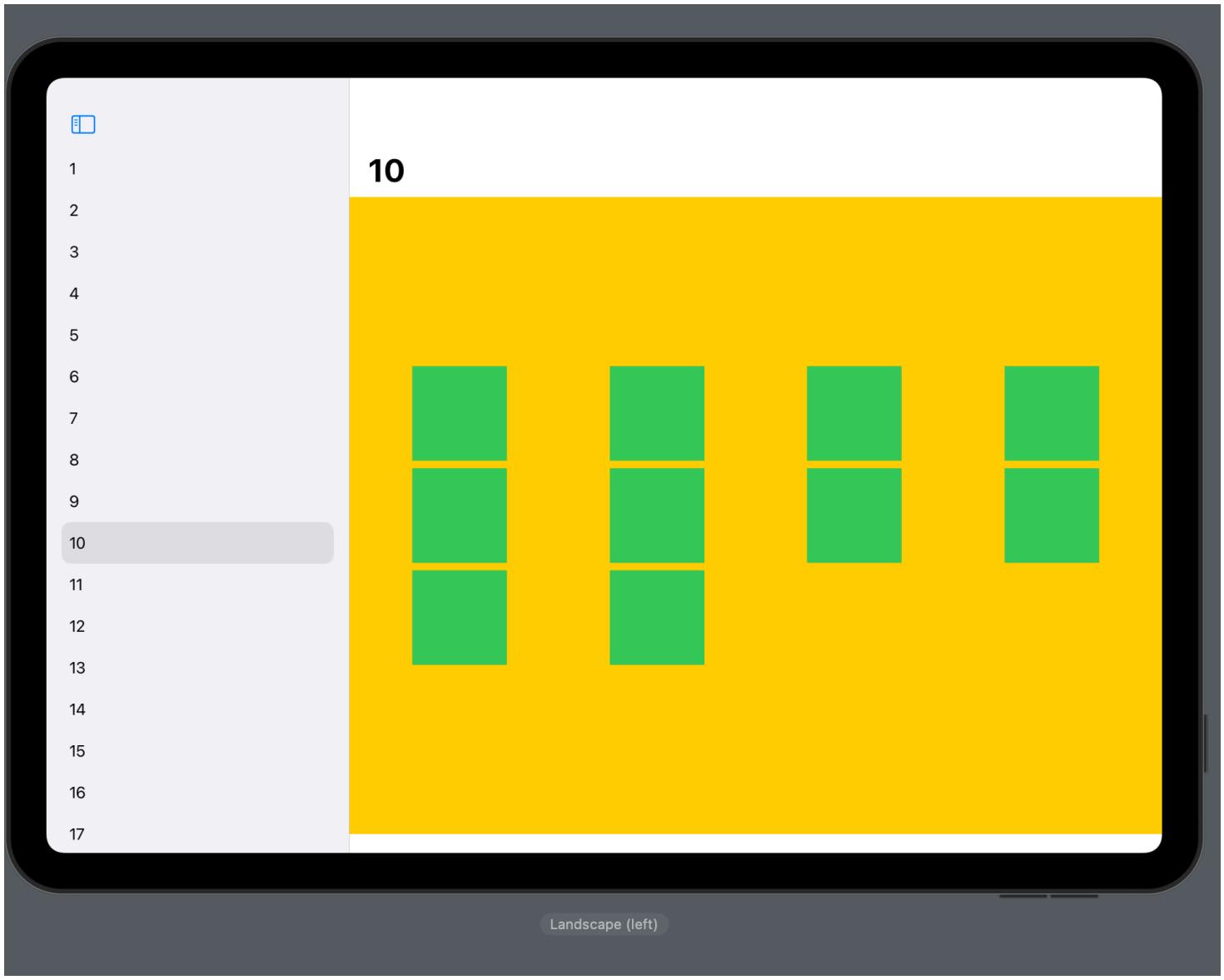

Figure 7: Displaying rectangles based on the sidebar selection

Next, we want to tap on the rectangle and perform a push navigation in our detail view. This is where we will introduce NavigationStack inside the detail view and set navigationDestination based on the type of the NavigationLink value. This is shown in **Listing 23**.

```
struct ContentView: View {

    @State private var selection: Int?
    let columns =
[GridItem(.flexible()),GridItem(.flexible()),GridItem(.flexible()),GridItem(.f
lexible())]

    var body: some View {
        NavigationSplitView {
            List(1...20, id: \.self, selection: $selection) { index in
                Text("\(index)")
            }
        } detail: {

            NavigationStack {
                ZStack {
```

```
            if let selection {
                LazyVGrid(columns: columns, alignment: .center) {
                    ForEach(1...(selection), id: \.self) { index in
                        VStack {

                            NavigationLink(value: index) {
                                VStack {
                                    Rectangle()
                                        .fill(.green)
                                    Text("\(index)")
                                        .font(.title)
                                }

                            }

                        }.frame(width: 100, height: 100)
                    }
                }
                .padding(.horizontal)
                .frame(maxWidth: .infinity, maxHeight: .infinity)
                .background {
                    Color.yellow
                }.navigationTitle("\(selection)")
            }
        }.navigationDestination(for: Int.self) { intValue in
                Text("\(intValue)")
                    .font(.system(size: 100))
        }
                }

            }

        }
    }
```

Listing 23: NavigationStack inside detail view to perform nested navigation

Now, when you tap on the NavigationLink inside the detail view you will be taken to a brand new view which will display the selected number. The push navigation will push the view into the stack and keep track of the history. When you press the back button, the view will be popped from the stack and you will return back to the screen with rectangles. This is shown in **Figure 8** and **Figure 9**.

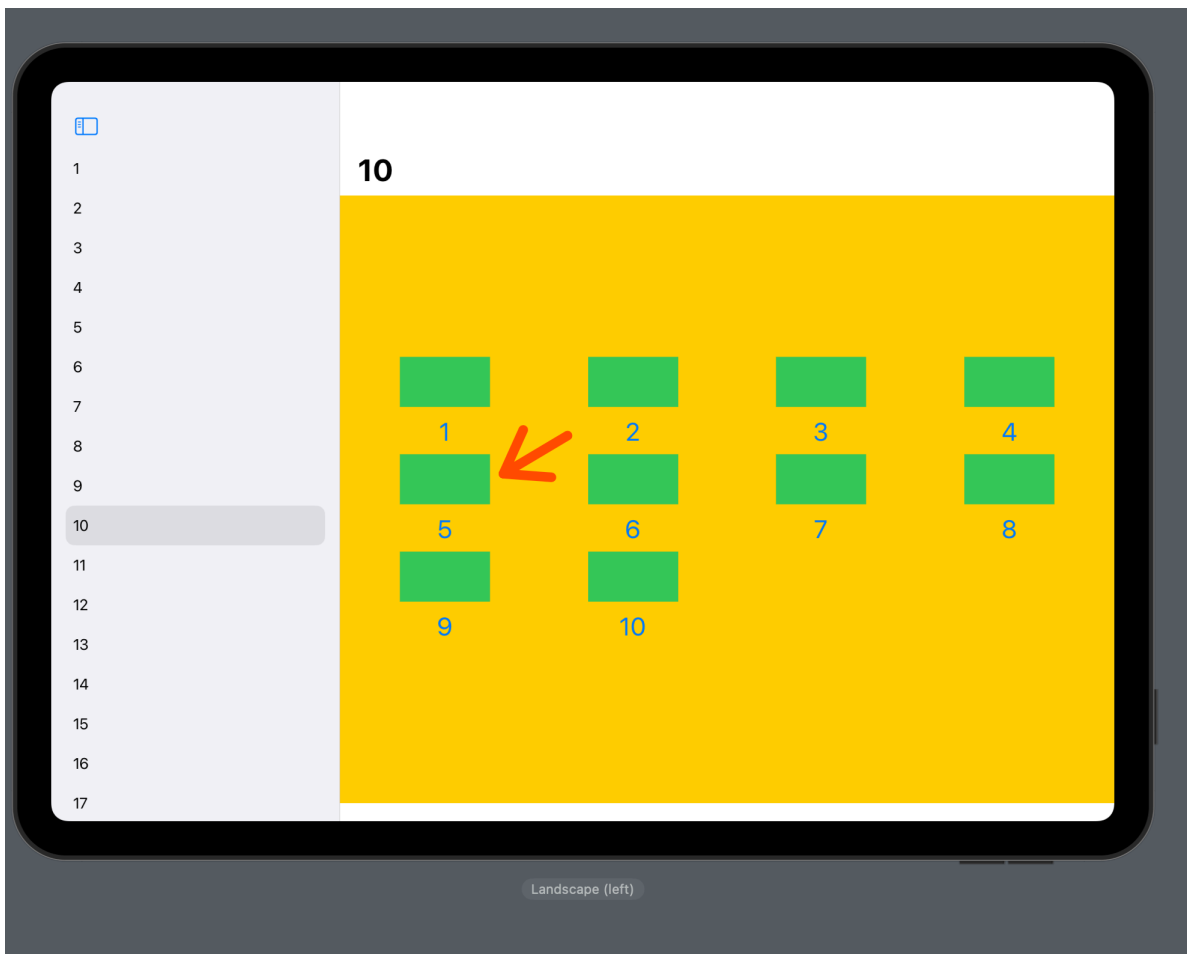

Figure 8: Tapping on the rectangle to perform the navigation

Figure 9: Displaying the selection

Conclusion

Navigation API in SwiftUI is a big improvement from its predecessor, NavigationView. Navigation API provides easy programmatic navigation based on the data type and also allows you to configure global app navigation. NavigationSplitView provides flexibility to developers to target devices with larger screen real estate, while still providing stack navigation for smaller devices.

I look forward to more advancements in Navigation API in the future and how developers integrate it to their existing apps