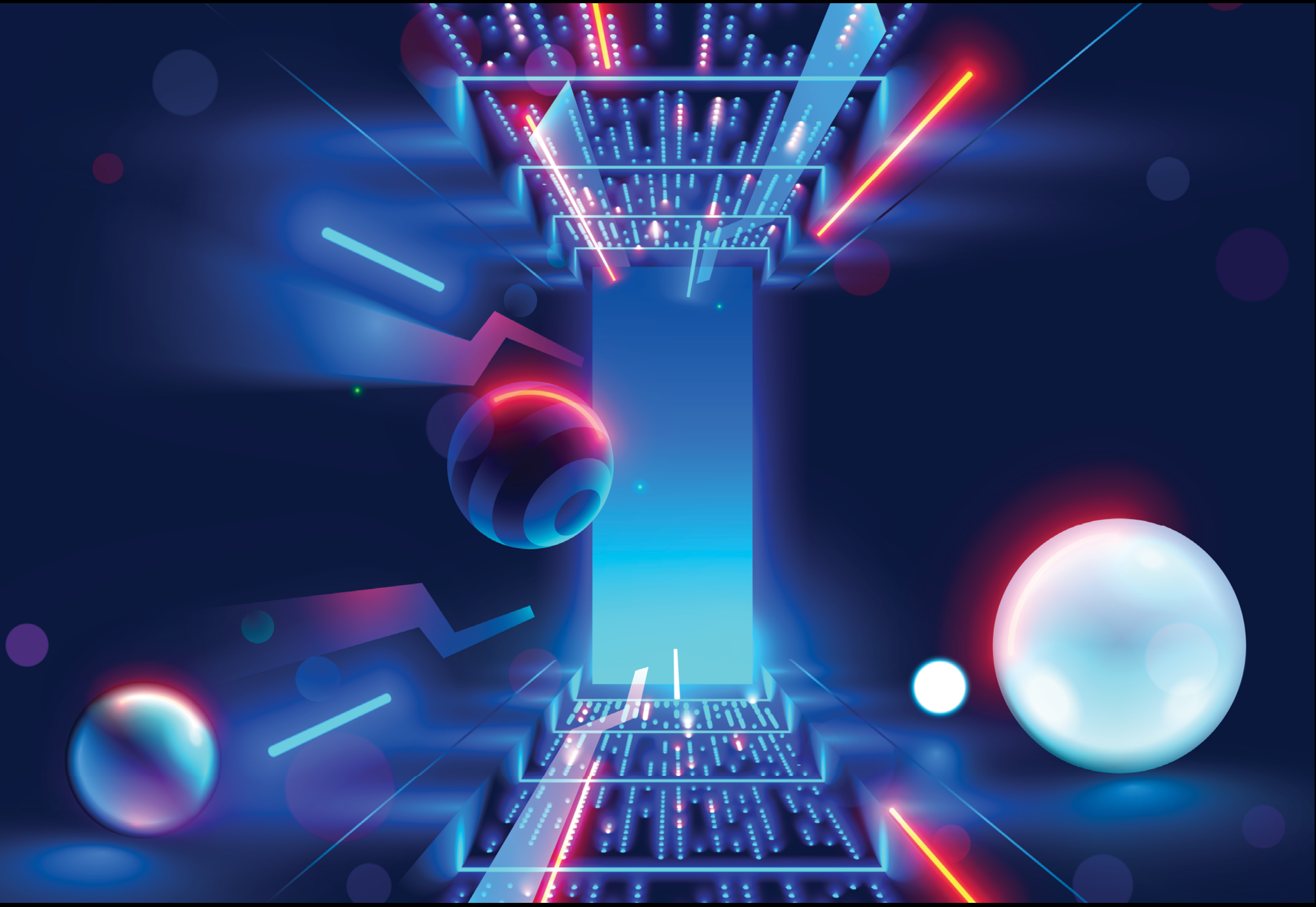
iOS 18 App Development Essentials



Payload publishing

iOS 18 App Development Essentials

iOS 18 App Development Essentials

ISBN-13: 978-1-951442-99-6

Rev: 1.0

Payload
publishing

https://www.payloadbooks.com

Chapter 40

40. A SwiftUI Custom Container Tutorial

The previous chapter explained how to create a custom container in SwiftUI. This chapter will build on this knowledge to create an app that presents a checklist interface using a custom container view.

40.1 About the Custom Container Project

The project we will create in this chapter is a checklist app that uses a custom container view. The container will support Text subviews and allow those items to be selected and deselected when tapped. Before completing the project, we will also add support for section headers. The completed app will appear as illustrated in Figure 40-1:

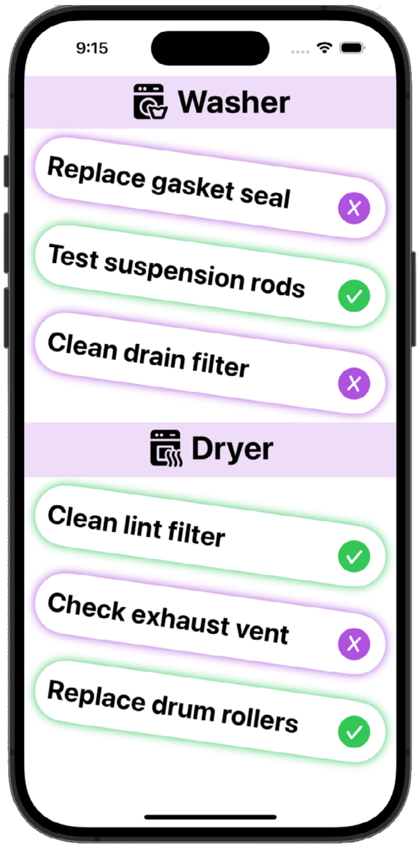


Figure 40-1

40.2 Creating the CustomContainerDemo Project

Launch Xcode and select the option to create a new Multiplatform App project named *CustomContainerDemo*.

40.3 Adding the Sample Data

Our example app simulates an appliance repair checklist consisting of a scrollable list of checkable items. The first step in the design process is adding arrays containing checklist items. Edit the *ContentView.swift* file to declare these arrays as follows:

```
import SwiftUI

let washerlist = [
    "Replace gasket seal",
    "Test suspension rods",
    "Clean drain filter"
]

let dryerlist = [
    "Clean lint filter",
    "Check exhaust vent",
    "Replace drum rollers"
]
.
.
```

40.4 Declaring the Item View

Each checklist item will be displayed using a custom view. When this item view is called, it is passed a ViewBuilder closure containing the subview to be displayed. We will enhance the item view later in the chapter, but for now, declare the basic view outline as follows:

```
struct CheckItemView<Content: View>: View {

    @ViewBuilder let content: Content

    var body: some View {
        ZStack {
            RoundedRectangle(cornerRadius: 50)
                .fill(.white)
                .padding(5)

            HStack {
                content
                    .font(.title)
                    .fontWeight(.bold)
            }
            .padding(18)
        }
        .padding(5)
        .frame(width: .infinity)
    }
}
```

40.5 Designing the Container

With an initial version of the item view added, we can start work on the custom container view. Once again, the view will be passed a ViewBuilder closure containing the subviews. Using the *ForEach(subviews:)* initializer, we can loop through the subviews, passing each to CheckItemView for rendering:

```
struct CheckList<Content: View>: View {
    @ViewBuilder var content: Content

    var body: some View {

        ScrollView(.vertical) {
            VStack(spacing: 20) {
                ForEach(subviews: content) { subview in
                    CheckItemView {
                        subview
                    }
                }
            }
        }
    }
}
```

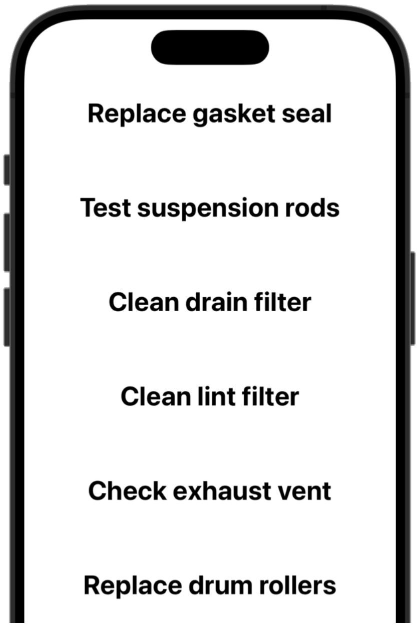
40.6 Using the Custom Container

Though more work will be necessary before the app resembles Figure 40-1, we have completed enough that the checklist should at least appear in the preview panel. To test our progress so far, modify the ContentView declaration to pass the checklist items to the container view:

```
struct ContentView: View {
    var body: some View {
        CheckList {
            ForEach(washerlist, id: \.self) { item in
                Text(item)
            }

            ForEach(dryerlist, id: \.self) { item in
                Text(item)
            }
        }
    }
}
```

When the view is rendered in the preview panel, it should appear as shown in Figure 40-2:


Replace gasket seal

Test suspension rods

Clean drain filter

Clean lint filter

Check exhaust vent

Replace drum rollers

Figure 40-2

40.7 Completing the Item View

The next step is to complete the CheckItemView implementation to enhance the appearance with colors, shading, and checkmark images and to add tap gesture recognition. We will begin by adding some state properties to store the current item state (whether the item is selected or not) and configure the corresponding color:

```
struct CheckItemView<Content: View>: View {
    @ViewBuilder let content: Content

    @State private var color = Color.green
    @State private var isEnabled:Bool = false

    var body: some View {

        let color = (self.isEnabled ? Color.green : Color.purple)

        ZStack {
.
.
```

We also need to apply a tap gesture modifier to the RoundedRectangle view to toggle the *isEnabled* state and apply the *color* state value using the *.shadow()* modifier:

```
.
.
ZStack {
    RoundedRectangle(cornerRadius: 50)
        .fill(.white)
        .shadow(color: color, radius: 5)
```

```
          .onTapGesture {
              isEnabled.toggle()
          }
          .padding(5)
.
.
```

Use the Preview panel to verify that the checklist items toggle between green and purple shadowing when tapped.

The last changes to CheckItemView involve adding checkmark images and a rotation angle. We will use images from the built-in SF Symbols library for the checkmarks and apply visual effects when the app launches and an item is tapped. Locate the HStack in the CheckItemView declaration and make the following additions:

```
.
.
HStack {
    content
        .font(.title)
        .fontWeight(.bold)
    Spacer()
    Image(systemName: (isEnabled ? "checkmark.circle.fill" : "x.circle.fill"))
        .foregroundColor(color)
        .font(.largeTitle)
        .symbolEffect(.rotate, options: .nonRepeating)
        .contentTransition(.symbolEffect(.replace))
}
.padding(18)
.
.
```

Finally, apply the *.rotationModifier()* to the ZStack container as follows to tilt each item by 8 degrees:

```
.
.
ZStack {
    RoundedRectangle(cornerRadius: 50)
.
.
            .contentTransition(.symbolEffect(.replace))
    }
    .padding(18)
}
.padding(5)
.frame(width: .infinity)
.rotationEffect(Angle(degrees: 8))
.
.
```

Refresh the Preview and test that the checkmarks perform a single rotation when the view appears and that tapping gestures toggle between checked and unchecked images. With these changes completed, the checklist view will resemble the following figure:

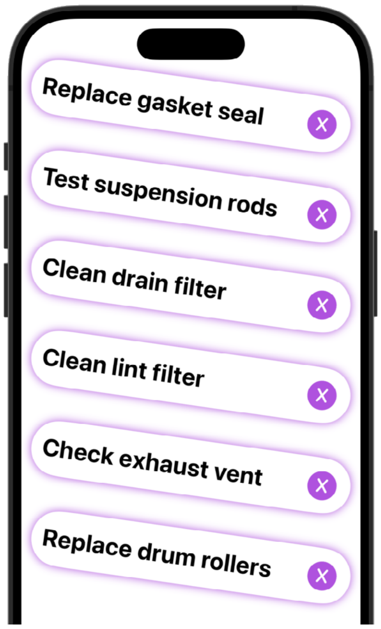


Figure 40-3

40.8 Adding Section Headers

The sole remaining task is to add section support to the container. We will begin by adding a section header view as follows:

```
struct ChecklistSectionHeader<Content: View>: View {

    @ViewBuilder var content: Content

    var body: some View {
        HStack {
            content
                .font(.largeTitle)
                .fontWeight(.bold)
        }
        .padding(5)
        .frame(maxWidth: .infinity)
        .background(Color.purple.opacity(0.2))
    }
}
```

This header view will be called from within the CheckList container view using a *ForEach(sections:)* loop as follows:

```
struct CheckList<Content: View>: View {
    @ViewBuilder var content: Content
    var body: some View {

        ScrollView(.vertical) {
            VStack(spacing: 20) {

                ForEach(sections: content) { section in

                    if !section.header.isEmpty {
                        ChecklistSectionHeader {
                            section.header
                        }
                    }

                    ForEach(subviews: section.content) { subview in
                        CheckItemView {
                            subview
                        }
                    }
                }
            }
        }
    }
}
```

As the loop executes, each section will have an optional header and a group of subviews belonging to that section. When calling CheckItemView, therefore, we have modified the *ForEach(subviews:)* construct to access each section's *content* property:

```
ForEach(subviews: section.content) { subview in
```

Modify ContentView as follows to add sections, and check the preview to confirm that the app appears as illustrated in Figure 40-1 above:

```
struct ContentView: View {
    var body: some View {
        CheckList {
            Section("\(Image(systemName: "washer.fill")) Washer") {
                ForEach(washerlist, id: \.self) { item in
                    Text(item)
                }
            }

            Section("\(Image(systemName: "dryer.fill")) Dryer") {
                ForEach(dryerlist, id: \.self) { item in
                    Text(item)
                }
            }
        }
    }
}
```

40.9 Summary

This chapter demonstrated creating and using a SwiftUI custom container view, including ViewBuilder closures, ForEach loops, and sections and headers.