Article

Populating SwiftUI menus with adaptive controls

Improve your app by populating menus with controls and organizing your content intuitively.

Overview

Menus are versatile components you can populate adaptively and use to organize commands, actions, or items in your app.

In tight layouts or smaller devices, menus optimize space by displaying options on demand. Use menus to conceal complex interface options when actions can be logically grouped. You have options for configuring your menus, with various controls like `Button`, `Toggle`, `Slider`, `Divider`, and more. This adaptability ensures that your menus remain flexible and succinct while supporting complex use cases.

> **Note**
>
> While the code for creating a menu in SwiftUI is largely the same across platforms, the system may display the menu differently depending on the device.

Plan the structure of your menus

Make your menus simple and flexible, able to adapt to various interfaces, such as regular and compact size classes on iOS and iPadOS and across macOS, tvOS, and visionOS.

A menu consists of three components:

- Label: A view that describes the purpose of the menu.

- Content: A closure that uses a `ViewBuilder` to define the items inside the menu.
- Primary action: An optional closure that performs an action when someone clicks or taps the menu, instead of the default primary action of opening the menu. When provided, opening the menu becomes the secondary action, such as opening after a long press gesture instead of a tap.

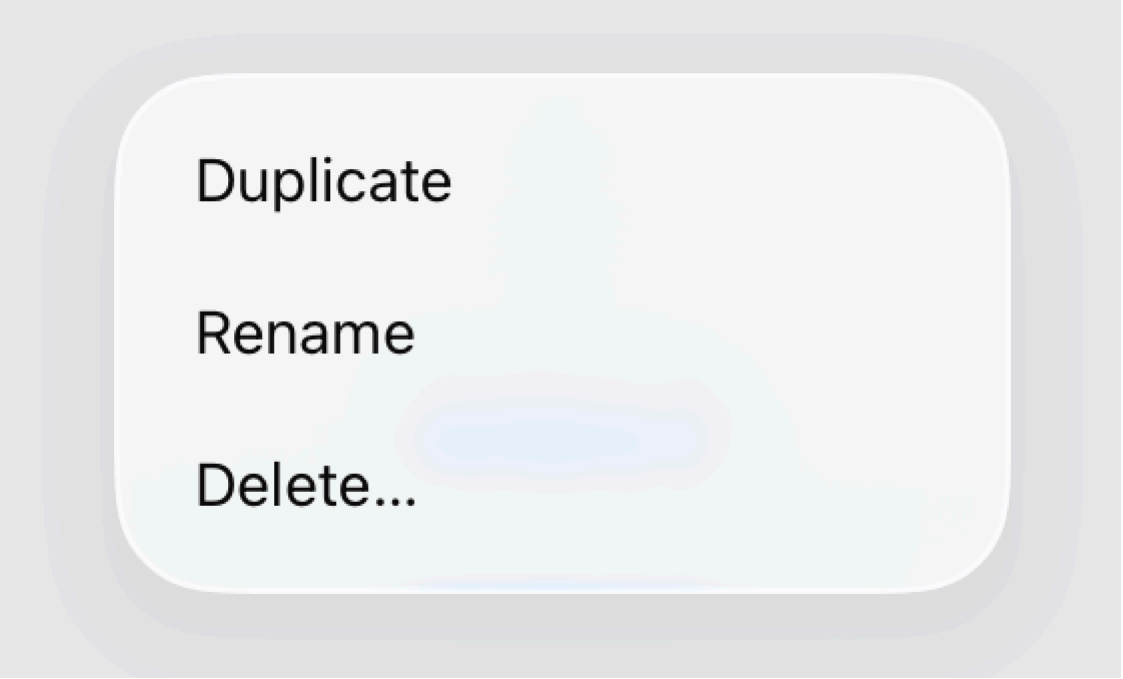
For design guidance, see Human Interface Guidelines > Menus.

Populate your menus

A well-declared SwiftUI `Menu` resembles its ultimate rendered appearance: the contents of the menu visually adapt to the purpose of each element. For example, inserting a `Button` in the menu's closure renders an actionable menu item, while inserting a `Menu` creates a submenu item.

To render a menu item that performs a given action closure, use the `Button` control:

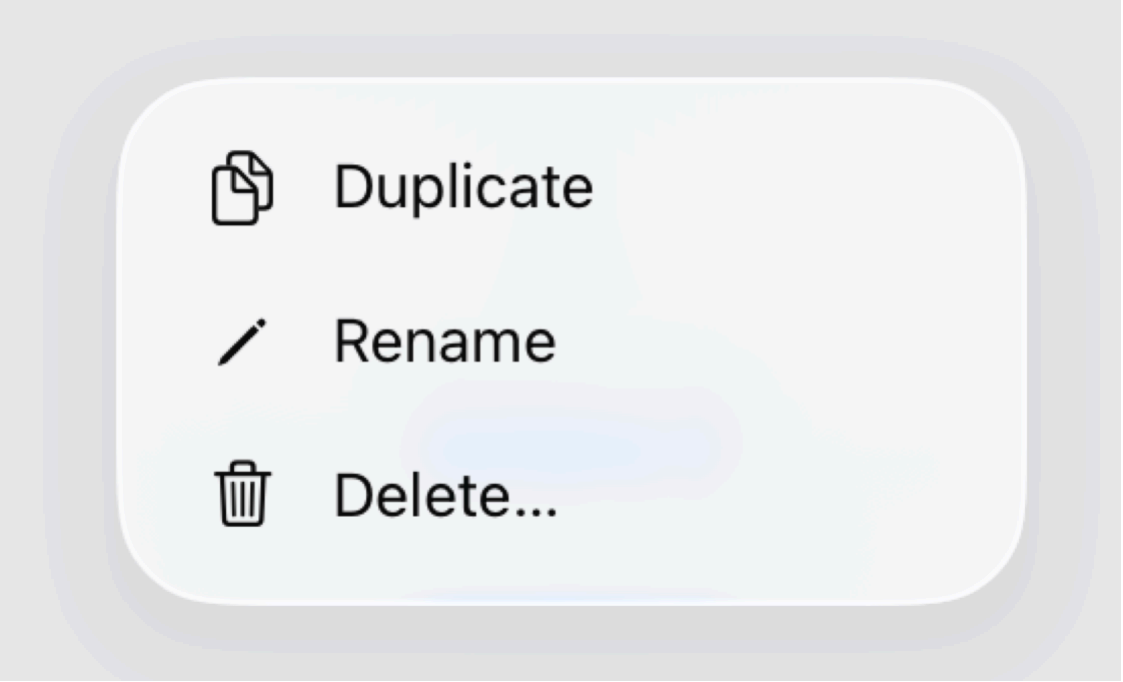
```
Menu("Actions") {
    Button("Duplicate") {
        // Duplicate action.
    }
    Button("Rename") {
        // Rename action.
    }
    Button("Delete…") {
        // Delete action.
    }
}
```



> **Note**
>
> SwiftUI controls and views are adaptive, and represent functionality and meaning, as well as visual representation. When you open a menu, the menu items appear in a context-appropriate order depending on the platform. For more information, see `menuOrder(_:)`.

To show a symbol next to the menu item title, use the `init(_:systemImage:action:)` initializer:

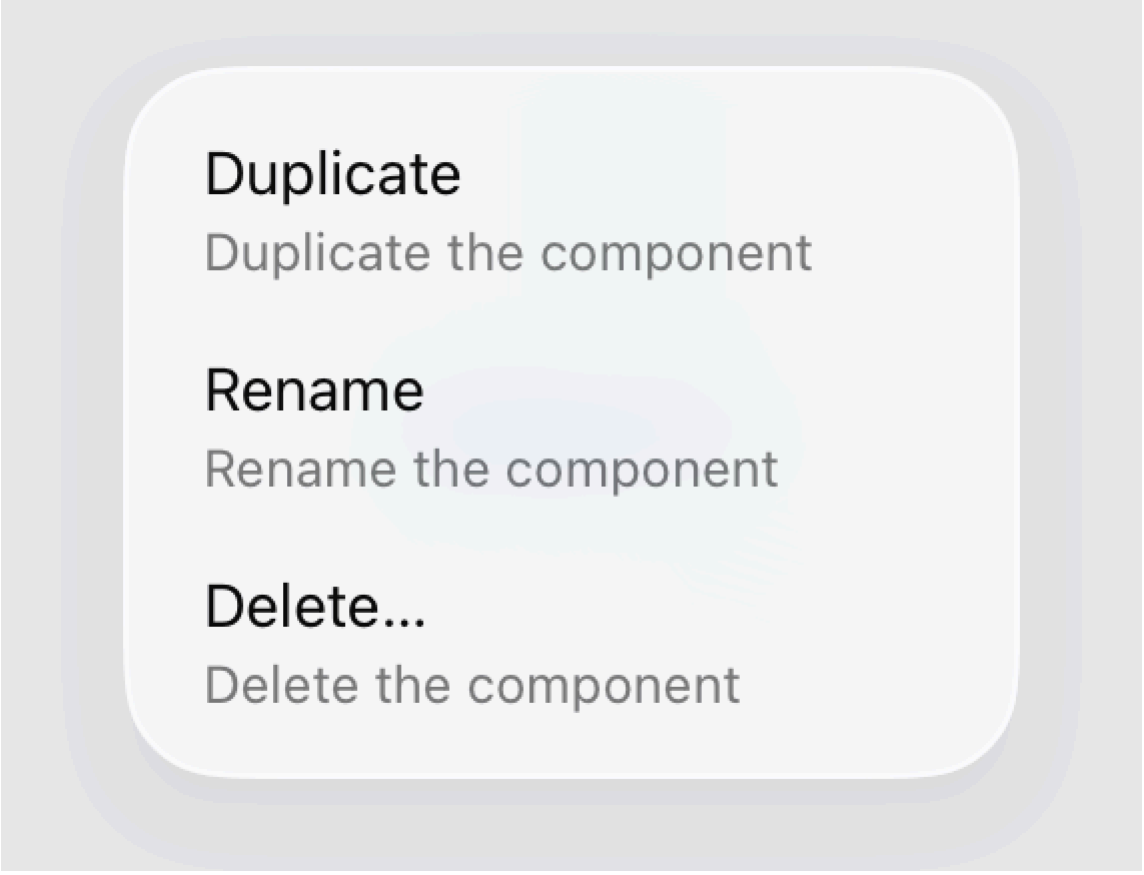
```
Menu("Actions") {
    Button("Duplicate", systemImage: "doc.on.doc") {
        // Duplicate action.
    }
    Button("Rename", systemImage: "pencil") {
        // Rename action.
    }
    Button("Delete…", systemImage: "trash") {
        // Delete action.
    }
}
```



You can also construct menu actions by adding the label closure initializers on `Button`. This method provides more flexibility for your subtitles.

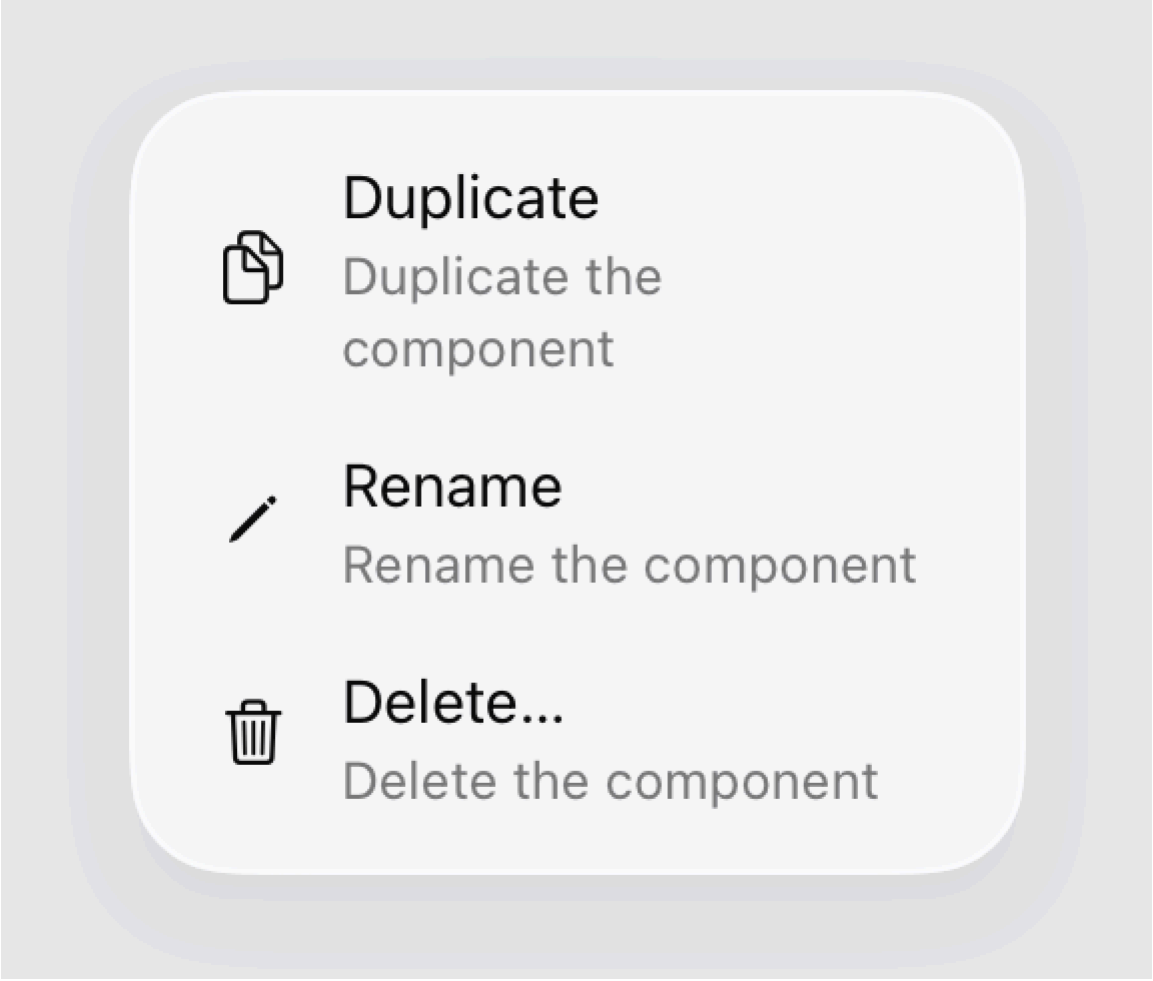
To add a title and subtitle to a menu item, populate the control's label closure with two `Text` views, in which the first text represents the title, and the second represents the subtitle. The following example shows this hierarchical style applied to the views:

```
Menu("Actions") {
    Button {
        // Duplicate action.
    } label: {
        Text("Duplicate")
        Text("Duplicate the component")
    }
    Button {
        // Rename action.
    } label: {
        Text("Rename")
        Text("Rename the component")
    }
    Button {
        // Delete action.
    } label: {
        Text("Delete…")
        Text("Delete the component")
    }
}
```



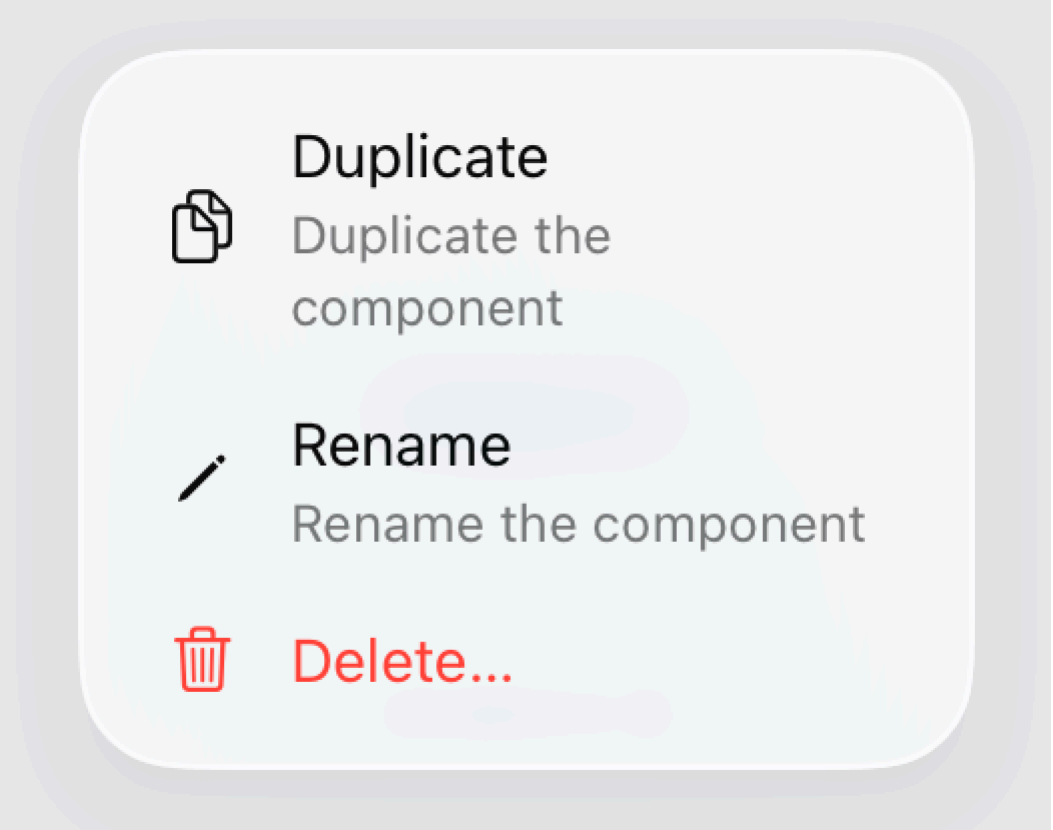
You can insert an icon by replacing the first `Text` with a `Label`:

```
Menu("Actions") {
    Button {
        // Duplicate action.
    } label: {
        Label("Duplicate", systemImage: "doc.on.doc")
        Text("Duplicate the component")
    }
    Button {
        // Rename action.
    } label: {
        Label("Rename", systemImage: "pencil")
        Text("Rename the component")
    }
    Button {
        // Delete action.
    } label: {
        Label("Delete…", systemImage: "trash")
        Text("Delete the component")
    }
}
```



Add a visual warning cue to menu items that are destructive by nature. Add a `destructive` role to `Button` to tint the menu item red. Use `destructive` only for actions that require caution.

```
Menu("Actions") {
    // ...

    Button("Delete…", systemImage: "trash", role: .destructive) {
        // Delete action.
    }
}
```



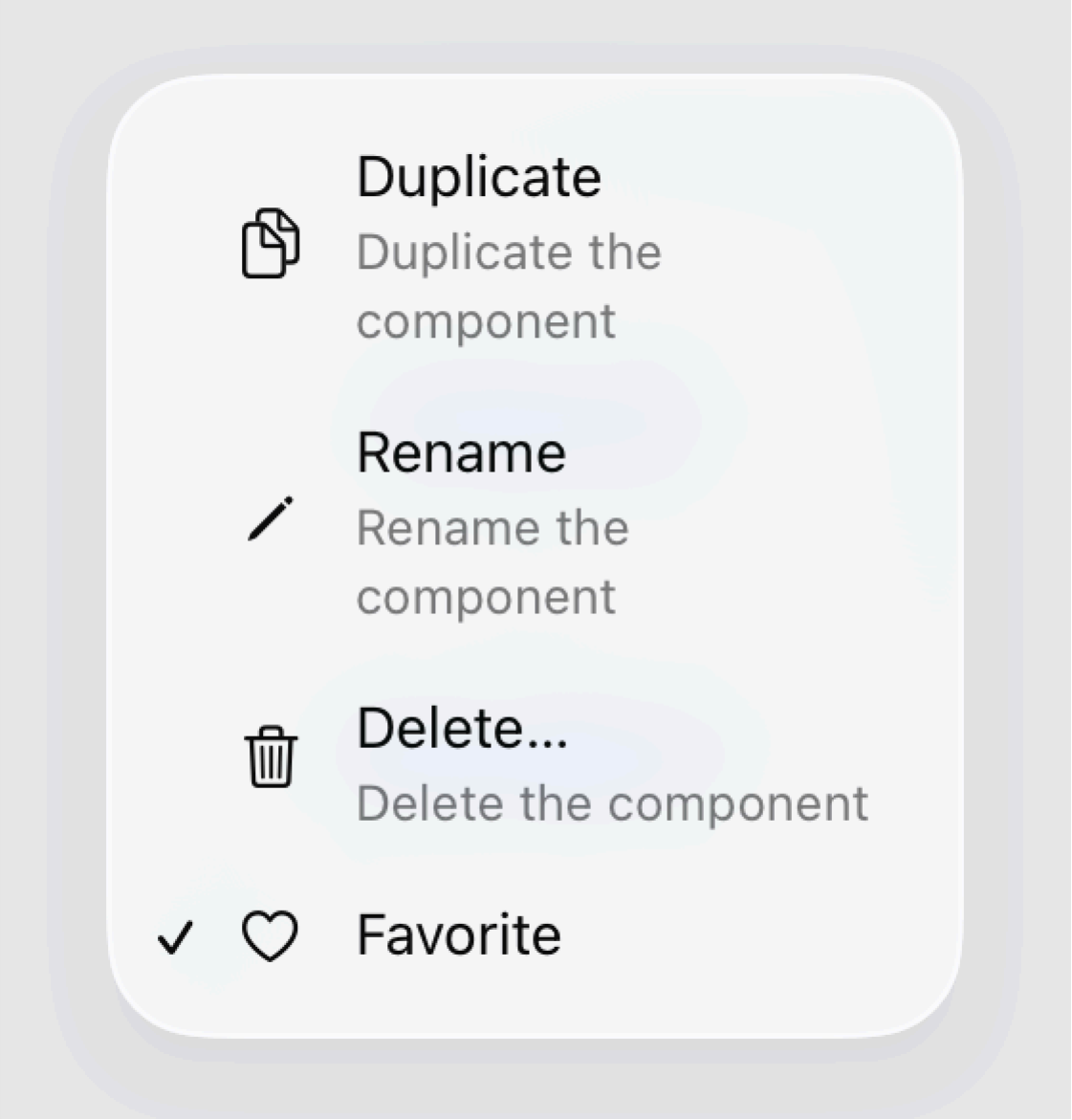
On macOS, menu items constructed with a `Label` render without an icon by default. Use the `titleAndIcon` style to override the system behavior and explicitly render an icon for the menu items.

```
Menu("Actions") {
    // ...
}
.labelStyle(.titleAndIcon)
```

Menus are also great for representing toggled items. To render a toggled menu item, you can add a `Toggle` to the menu's content.

Because SwiftUI controls adapt to their context, a `Toggle` in a menu automatically appears with a checkmark indicating its on or off state.

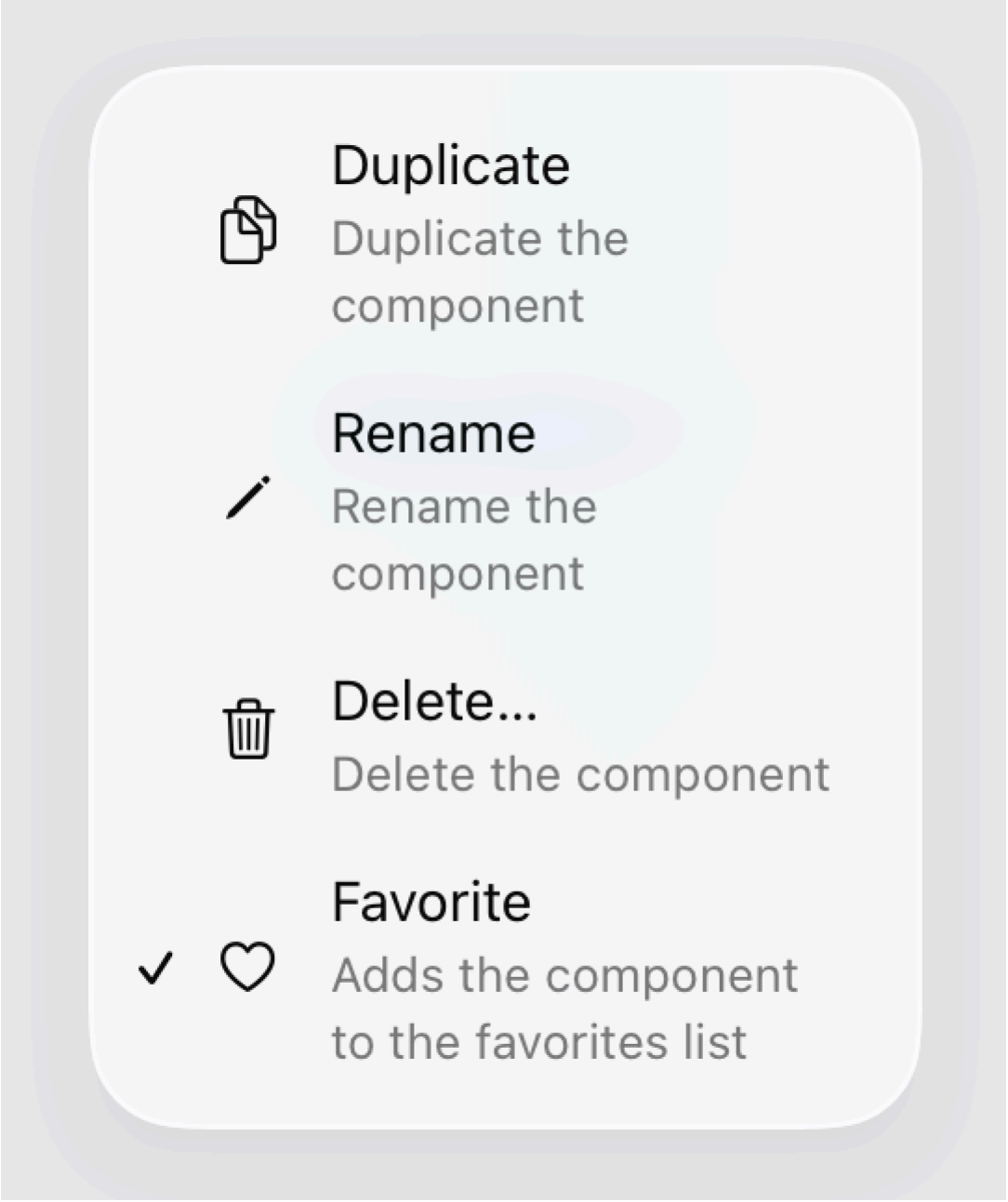
```
Menu("Actions") {
    // ...

    Toggle(
        "Favorite",
        systemImage: "suit.heart",
        isOn: $isFavorite)
}
```



Just like `Button`, initialize a `Toggle` with a label closure for more flexibility.

```
Menu("Actions") {
    // ...
```

```
    Toggle(isOn: $isFavorite) {
        Label("Favorite", systemImage: "suit.heart")
        Text("Adds the component to the favorites list")
    }
}
```



Use a `Picker` within a menu to let people choose from a list of options:

```
enum Flavor: String, CaseIterable, Identifiable {
    case chocolate, vanilla, strawberry
    var id: Self { self }
}

@State private var selectedFlavor: Flavor = .chocolate

var body: some View {
```

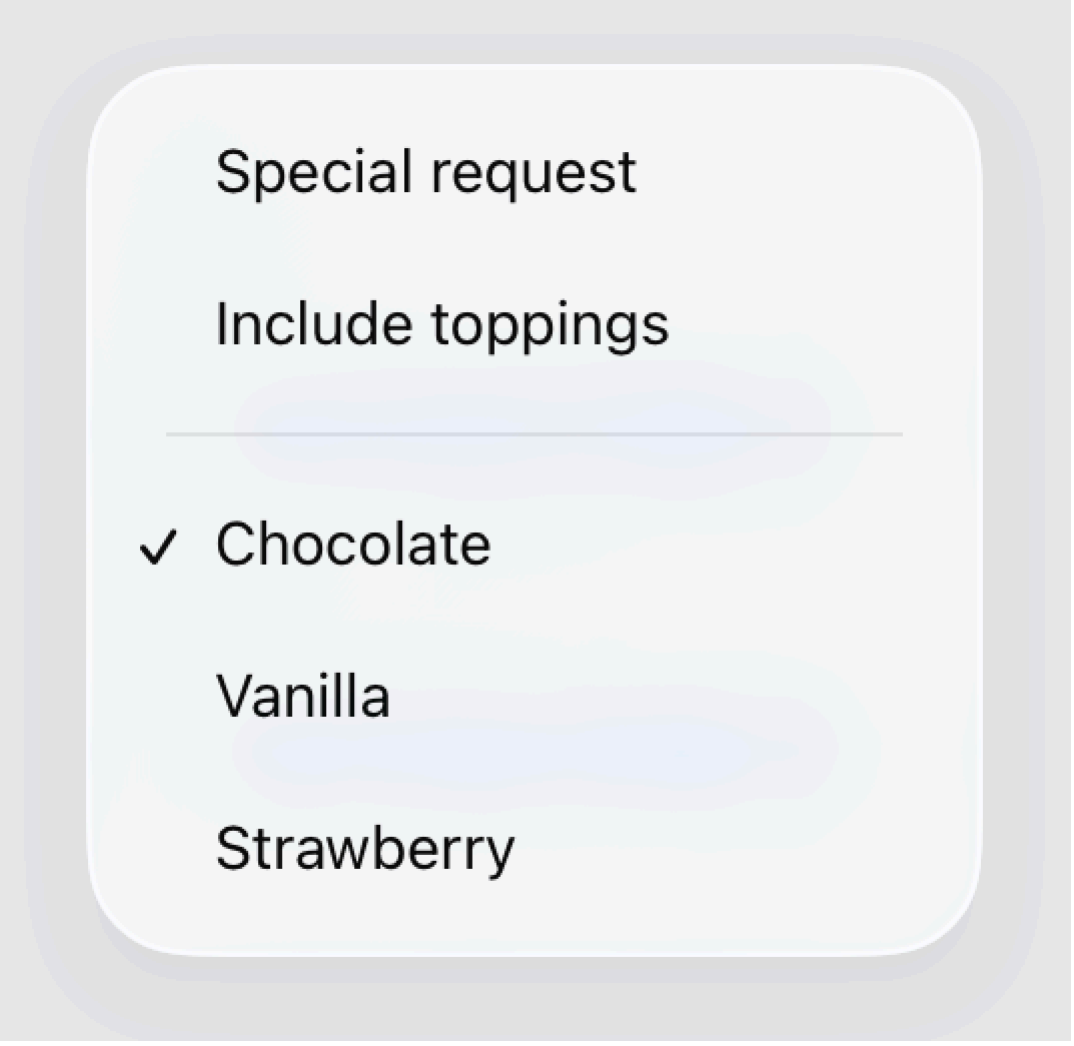
```
    Picker("Flavor", selection: $selectedFlavor) {
        ForEach(Flavor.allCases) { flavor in
            Text(flavor.rawValue.capitalized)
                .tag(flavor)
        }
    }
}
```

This example embeds a picker within a menu, displaying multiple selectable items. Although you can select several options, only one item is active at any given time. The selected item, identified with a checkmark, indicates the current selection.

Adding a picker to a menu creates a more convenient and customized layout than several individual toggles. A picker provides a single interface to manage multiple options, ensuring a person can only select one item at a time. Multiple toggles might be more appropriate when your content doesn't require mutual exclusivity.

```
enum Flavor: String, CaseIterable, Identifiable {
    case chocolate, vanilla, strawberry
    var id: Self { self }
}

@State private var selectedFlavor: Flavor = .chocolate
@State private var includesToppings: Bool = false

var body: some View {
    Menu("Ice Cream Order") {
        Button("Special request") {
```

```
            // Create a special request.
        }

        Toggle("Include toppings", isOn: $includesToppings)

        Picker("Flavor", selection: $selectedFlavor) {
            ForEach(Flavor.allCases) { flavor in
                Text(flavor.rawValue.capitalized)
                    .tag(flavor)
            }
        }
    }
```



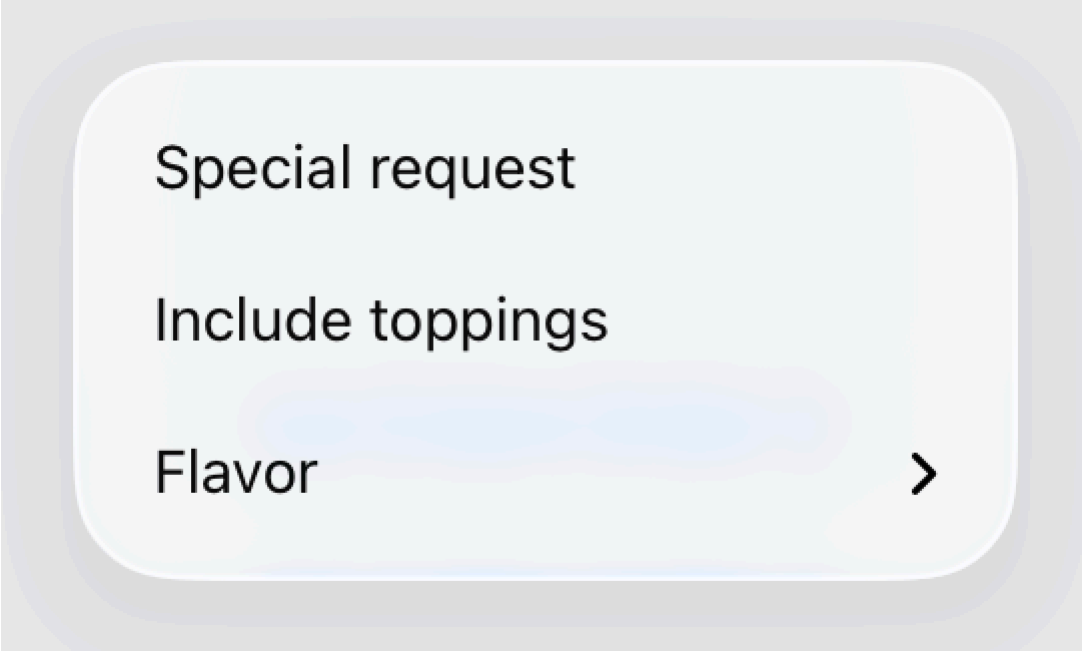
You can choose picker styles such as `inline`, `menu`, and `palette`.

Apply style to menu pickers

By default, picker options in menus appear inline. SwiftUI implicitly applies the `inline` style, allowing you to select options without navigating away from the current view. The inline style works well for settings or configurations that require immediate context.

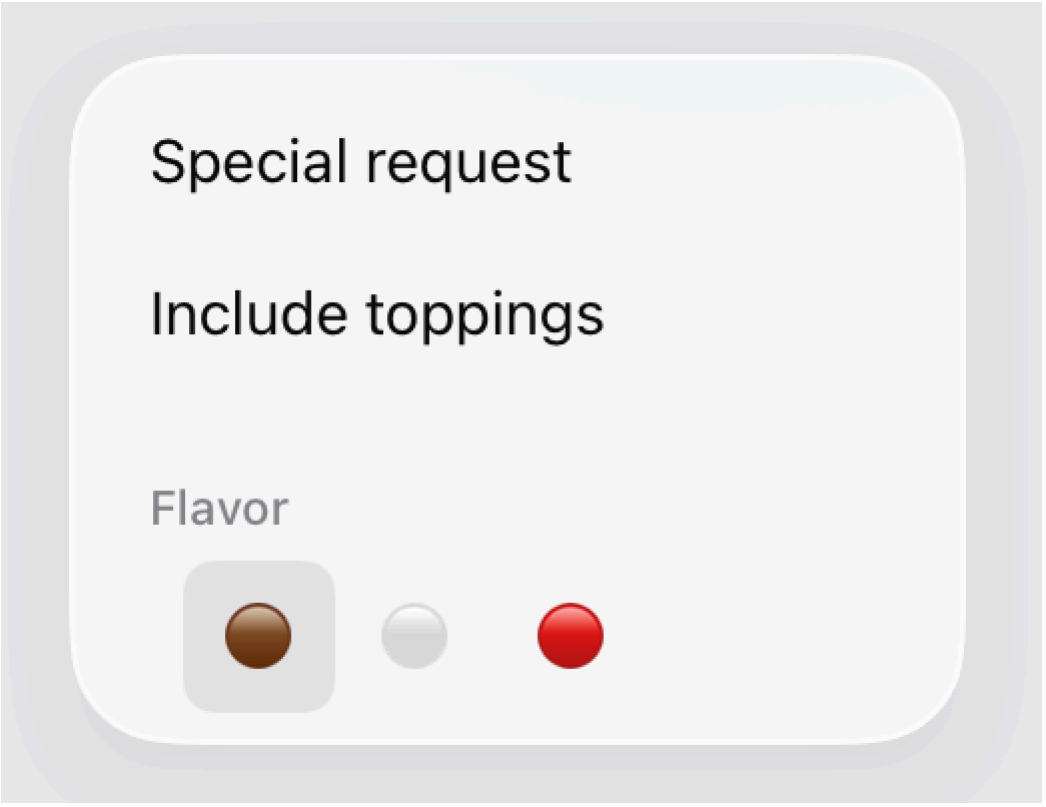
When you apply the `menu` style to a picker within a menu, it transforms into a submenu, presenting options in a hierarchical manner. This style helps organize complex menus with categorized options.

```
enum Flavor: String, CaseIterable, Identifiable {
    case chocolate, vanilla, strawberry
    var id: Self { self }
}

@State private var selectedFlavor: Flavor = .chocolate
@State private var includesToppings: Bool = false

var body: some View {
    Menu("Ice Cream Order") {
        Button("Special request") {
            // Create a special request.
        }

        Toggle("Include toppings", isOn: $includesToppings)

        Picker("Flavor", selection: $selectedFlavor) {
            ForEach(Flavor.allCases) { flavor in
                Text(flavor.rawValue.capitalized)
                    .tag(flavor)
            }
        }
        .pickerStyle(.menu)
    }
}
```



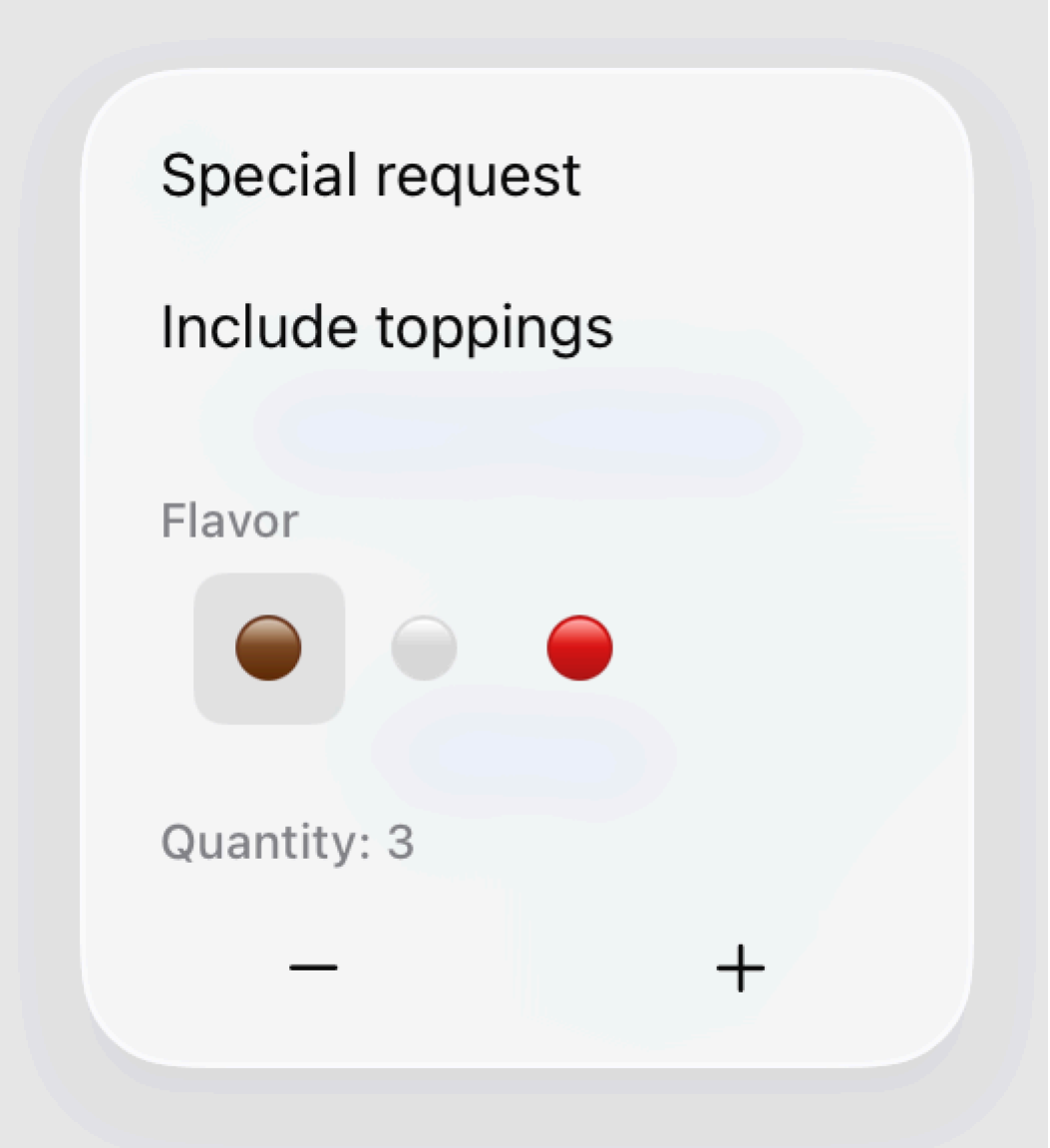
Palette pickers work best in compact scenarios in which someone chooses from a set of symbols. Palette pickers minimize icons, and turn into a horizontal scroll if there's limited space.

```
enum Flavor: String, CaseIterable, Identifiable {
    case chocolate, vanilla, strawberry
    var id: Self { self }
}

@State private var selectedFlavor: Flavor = .chocolate
@State private var includesToppings: Bool = false

var body: some View {
    Menu("Ice Cream Order 3") {
        Button("Special request") {
            // Create a special request.
        }
        Toggle("Include toppings", isOn: $includesToppings)
        Picker("Flavor", selection: $selectedFlavor) {
            Text("🟤")
                .tag(Flavor.chocolate)
            Text("⚪️")
                .tag(Flavor.vanilla)
            Text("🔴")
                .tag(Flavor.strawberry)
        }
        .pickerStyle(.palette)
    }
}
```



Menus can also handle numerical values with sliders and steppers.

```
@State private var quantity: Int = 1

Menu("Actions") {
    // ...

    Stepper(value: $quantity) {
        Text("Quantity: \(quantity)")
    }
}
```



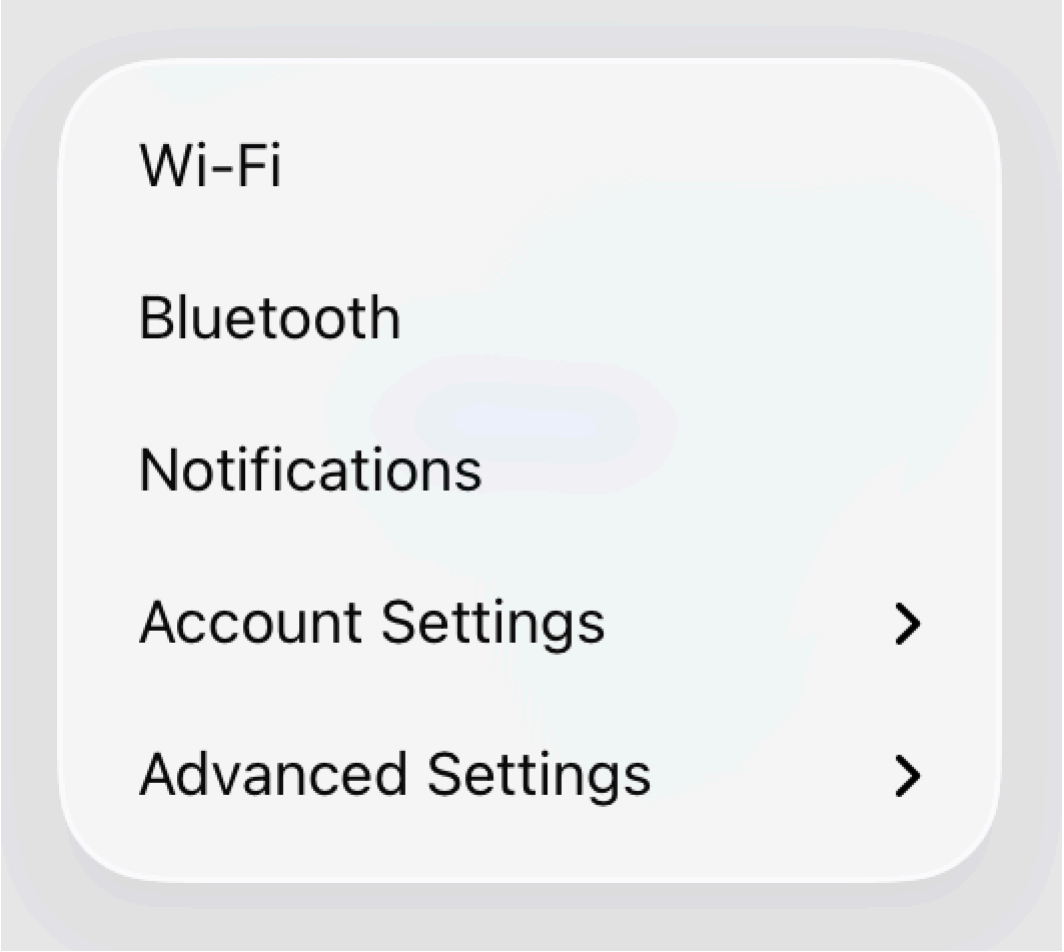
Group menu items

SwiftUI provides multiple ways to group items within menus, including submenus, sections, and dividers.

Submenus group items hierarchically, hiding content until needed. A submenu keeps the main menu uncluttered, while providing access to additional options when necessary:

```
Menu("General Settings") {
    // The General Settings submenu.
    Button("Wi-Fi") { openWiFiSettings() }
    Button("Bluetooth") { openBluetoothSettings() }
    Button("Notifications") { openNotificationSettings() }

    // The Account Settings submenu.
    Menu("Account Settings") {
        Button("Profile") { openProfileSettings() }
        Button("Security") { openSecuritySettings() }
```

```
        Button("Privacy") { openPrivacySettings() }
    }

    // The Advanced Settings submenu.
    Menu("Advanced Settings") {
        Button("Developer Options") { openDeveloperOptions() }
        Button("System Update") { openSystemUpdate() }
        Button("Backup & Restore") { openBackupRestore() }
    }
}
```



In the example above, the Settings menu populates with two submenus, grouping related and less-prominent settings actions.

You can also organize items with sections. The `Section` view groups items while keeping all elements visible, often with section headers for clarity. This style is useful for organizing related items within the root-level menu, providing clear separation and context for each group.

```
Menu("Settings") {
    // The General Settings submenu.
    Section("General Settings") {
        Button("Wi-Fi") { openWiFiSettings() }
        Button("Bluetooth") { openBluetoothSettings() }
```

```
        Button("Notifications") { openNotificationSettings() }
    }

    // The Account Settings submenu.
    Section("Account Settings") {
        Button("Profile") { openProfileSettings() }
        Button("Security") { openSecuritySettings() }
        Button("Privacy") { openPrivacySettings() }
    }

    // The Advanced Settings submenu.
    Section("Advanced Settings") {
        Button("Developer Options") { openDeveloperOptions() }
        Button("System Update") { openSystemUpdate() }
        Button("Backup & Restore") { openBackupRestore() }
    }
}
```

Display compact menu items

When you want to display a few related actions in a single row within a menu, consider using a `ControlGroup`. This method provides a compact, horizontally-grouped layout of up to four items.

```
Menu("Edit") {
    ControlGroup {
        Button {
            // Undo action
        } label: {
            Label("Undo", systemImage: "arrow.uturn.backward")
        }

        Button {
            // Redo action
        } label: {
            Label("Redo", systemImage: "arrow.uturn.forward")
        }

        Button {
            // Copy action
        } label: {
```

```
            Label("Copy", systemImage: "doc.on.doc")
        }
    }

    Divider()

    // Additional menu items here...
}
```

While submenus and sections are containers that group items, the `Divider` view provides a simple way to visually separate items within a menu. Unlike `Section`, `Divider` isn't a container, but serves as a visual break that divides groups of items to organize and group like-commands for improved usability and uniformity across apps.

Modify content behavior

Beyond populating a menu's content, SwiftUI also offers a set of APIs to modify the default behavior of menu items.

SwiftUI offers a set of APIs to modify the default behavior of menu items. On iOS and iPadOS, the system rearranges menu items by default so the first items in a menu appear closest to the viewer's point of interaction. To override this behavior and keep items in the order you define, use the `menuOrder(_:)` modifier:

```
Menu("Settings", systemImage: "ellipsis.circle") {
    Button("Select") {
        // Select folders
    }
    Button("New Folder") {
        // Create folder
    }
    Picker("Appearance", selection: $appearance) {
        Label("Icons", systemImage: "square.grid.2x2").tag(Appearance.icons)
        Label("List", systemImage: "list.bullet").tag(Appearance.list)
    }
}
.menuOrder(.fixed)
```

> **Note**
>
> On macOS, menu items typically follow the standard macOS ordering rules, and don't reorder for proximity.

By default, menus dismiss as soon as someone clicks or taps an item. If you want the person to make multiple selections, or repeat an action without reopening the menu, override this behavior with the `menuActionDismissBehavior(_:)` modifier on specific items.

The following code demonstrates:

- Increase and decrease actions that disable menu dismissal, letting someone click or tap them repeatedly to adjust the font size without re-opening the menu each time.
- A reset action that reverts the font to a default size. Because the action doesn't disable the dismissal, the menu closes after resetting.

```
Menu("Font size") {
    Button(action: increase) {
        Label("Increase", systemImage: "plus.magnifyingglass")
    }
    .menuActionDismissBehavior(.disabled)
    Button("Reset", action: reset)
    Button(action: decrease) {
        Label("Decrease", systemImage: "minus.magnifyingglass")
    }
    .menuActionDismissBehavior(.disabled)
}
```

See Also

Creating a menu

`struct Menu`

A control for presenting a menu of actions.

`func menuStyle<S>(S) -> some View`

Sets the style for menus within this view.