Article

Adding interactivity to widgets and Live Activities

Include buttons or toggles in a widget or Live Activity to offer app functionality without launching the app.

Overview

Widgets and Live Activities can include buttons and toggles to offer specific app functionality without launching the app. For example, a Reminders widget allows people to mark a task as completed with a toggle. On a locked device, buttons and toggles are inactive and the system doesn't perform actions unless a person authenticates and unlocks their device.

When you create a widget or Live Activity, think about how people interact with it. Then, make your app's most important actions available directly in a widget or Live Activity by adding buttons or toggles.

Important

An interaction with a button or toggle should do more than open the app. If you want to offer an interaction that opens the app, use `Link` and `widgetURL(_:)` as described in Linking to specific app scenes from your widget or Live Activity.

For design guidance, refer to Human Interface Guidelines > Widgets.

Related session from WWDC23

Session 10027: Bring widgets to new places

Review widgets that support interactive views

Widgets of the following sizes can include buttons and toggles:

- `WidgetFamily.systemSmall`
- `WidgetFamily.systemMedium`
- `WidgetFamily.systemLarge`
- `WidgetFamily.systemExtraLarge`
- `WidgetFamily.systemExtraLargePortrait`
- `WidgetFamily.accessoryCircular` on iPhone and iPad
- `WidgetFamily.accessoryRectangular` on iPhone and iPad

Live Activities can include buttons or toggles in the expanded and the Lock Screen presentation.

Support interactive widgets in CarPlay

People configure small system widgets to appear in CarPlay in the Widgets screen. To fit the context of widgets on a vehicle's built-in display, WidgetKit adjusts their functionality to match the CarPlay context. For additional information, refer to Adding StandBy and CarPlay support to your widget.

Understand the role of app intents

Widgets offer direct interaction with your app using the App Intents framework and SwiftUI. For example, the large widget of the Emoji Rangers: Supporting Live Activities, interactivity, and animations sample code project includes a button to boost the hero's healing power.

To create widgets and watch complications, you add a widget extension to your project. As a result, any code for widgets and watch complications runs in an independent process that's separate from your app. Based on your timeline entries, the system archives a view representation and only renders the views using these representations when applicable.

As a result of the timeline mechanism and of rendering in a separate process, the system can't run your code or update data bindings at the time it renders your widget. This is where the App Intents framework comes into play. App intents allow you to expose actions of your app to the system and enable it to perform the actions when needed — for example, when a person interacts with a button or a toggle in a widget.

> **Note**
>
> Live Activities don't use a timeline mechanism to update their content. However, they use WidgetKit and a widget extension with a similar cycle of view archiving and decoding. As a result, you add buttons and toggles to a Live Activity in the same way as do for your widgets, as described below.

Add an app intent that performs the action

Buttons and toggles you add to your widgets and Live Activities use functionality that you expose to the system by adopting the App Intents framework. Before you add a button or toggle, make the app functionality available to the system using an app intent:

1. For a widget, create a new structure that adopts the `AppIntent` protocol and add it to your app target. For a Live Activity interactive, adopt the `LiveActivityIntent` protocol. If the interaction starts or pauses media playback, adopt the `AudioPlaybackIntent` protocol.
2. Implement the protocol's requirements.
3. Define input parameters that your action needs using the `@Parameter` property wrapper and make sure their type conforms to the `AppEntity` protocol. Make sure input parameters have assigned values because, unlike app intents you define for system functionality like Siri, widgets don't resolve parameters for app intents.
4. In the protocol's required `perform()` function, add code for the action you want to make available to the widget.

For example, the Emoji Rangers: Supporting Live Activities, interactivity, and animations sample code project includes a button in its large widget that people click or touch to give the hero a healing boost. The following code snippet shows its app intent implementation:

```
struct SuperCharge: AppIntent {

    static var title: LocalizedStringResource = "Emoji Ranger SuperCharger"
    static var description = IntentDescription("All heroes get instant 100% health.'

    func perform() async throws -> some IntentResult {
        EmojiRanger.superchargeHeros()
        return .result()
    }
}
```

If you adopt the `LiveActivityIntent` or `AudioPlaybackIntent` protocol, the system runs the app intent in the app's process. Make sure to add your custom app intent to your app target.

If you adopt the `AppIntent` protocol, add your custom app intent to your widget extension target and your app target. Adding it to the app target in addition to the widget extension target allows you to reuse the button or toggle you add to the widget in your app.

Implement the perform function

When a person interacts with a button or toggle in your widget, the system runs the underlying app intent's `perform()` function. The `perform()` function is asynchronous, and you can take full advantage of Swift concurrency. To learn more about concurrency, see The Swift Programming Language > Concurrency and Concurrency. Per default, the system runs the app intent in the same process as the widget extension. However, if the app intent's `openAppWhenRun` property is `true`, or if the intent conforms to `AudioPlaybackIntent`, `ForegroundContinuableIntent`, `LiveActivityIntent`, or `PushToTalkTransmissionIntent`, the system performs the app intent in the app's process.

When you return from the `perform()` function, the system reloads the widget's timeline using its timeline provider. Use this as an opportunity to update your widget with the result of the interaction. Make sure any code that's necessary for the timeline update runs before you return from `perform()`. For example, use the `await` keyword while you store updated information in your database, and upon completion, `return` from `perform()`. Then, the system reloads the widget's timeline and your timeline provider loads the updated data from your database.

> **Note**
>
> Interactions with a toggle or button always guarantee a timeline reload.

Additionally, note that the `perform()` function is marked as `throws`. Be sure to handle errors instead of rethrowing them, and update your app, widget, and Live Activity as needed. For example, update a widget's interface to tell a person that it displays outdated information if it can't load new data.

> **Tip**
>
> With the App Intents framework, you can extend your app's custom functionality to support system-level services like Siri and the Shortcuts app. Additionally, you use app intents to make your widget configurable. By making your widget interactive, you automatically support system services for that interaction. To learn more about the App Intents framework, see App Intents and Making actions and content discoverable and widely available. To learn more about configurable widgets, see Making a configurable widget.

Add a button

After you've implemented the app intent, add a `Button` to your widget using one of the initializers that take an app intent, for example, `init(_:intent:)`.

> **Tip**
>
> You don't have to create a `Button` implementation solely for use in a widget or Live Activity. Reuse the code that adds a button to a widget — including the app intent implementation — in your app.

The following example shows how the large Emoji Rangers: Supporting Live Activities, interactivity, and animations widget includes a button to give the heroes a healing boost. Note how the sample checks for the presence of iOS 17 before adding the button.

```
struct EmojiRangerWidgetEntryView: View {
    var entry: SimpleEntry

    @Environment(\.widgetFamily) var family

    @ViewBuilder
    var body: some View {
        switch family {

        case .systemLarge:
            VStack {
                HStack(alignment: .top) {
                    AvatarView(entry.hero)
                        .foregroundStyle(.white)
                    Text(entry.hero.bio)
                        .foregroundStyle(.white)
                }
                .padding()
                if #available(iOS 17.0, *) {
                    HStack(alignment: .top) {
                        Button(intent: SuperCharge()) {
                            Image(systemName: "bolt.fill")
                        }
                    }
                    .tint(.white)
                    .padding()
```

```
                }
            }
            .containerBackground(for: .widget) {
                Color.gameBackgroundColor
            }
            .widgetURL(entry.hero.url)

        // Code for other widget sizes.
    }
}
```

After you've added a button, review views in your widget or Live Activity that change their data when a person interacts with the button. For example, a button press could cause several texts in the widget to reload. This reload may take some time, for example for interactions with an iPhone widget on Mac. To let a person know that the widget is waiting to update its content, add the `invalidatableContent(_:)` view modifier to views that receive updated data. Use this modifier judiciously; don't annotate every single view that might change, and use it on views that are important to a person.

> **Note**
>
> A button indicates that a person interacted with it but it doesn't remain in a pressed state. As a result, use `Toggle` to indicate on/off functionality.

Add a toggle

Add a `Toggle` to your view using one of the initializers that take an app intent, for example, `init(isOn:intent:label:)`.

The following example shows how a view of a task manager widget could add a toggle. Note that the initializer's `isOn` parameter receives a Boolean value instead of a `Binding<Bool>` value.

```
struct TodoItemView: View {
    var todo: Todo

    var body: some View {
        Toggle(isOn: todo.complete, intent: ToggleTodoIntent(todo.id)) {
            Text(todo.body)
        }
        .toggleStyle(TodoToggleStyle())
    }
```

```
}
```

> **Note**
>
> If you define your own `ToggleStyle`, check the `isOn` property and add the correct appearance for on and off states.

The `perform()` function runs code asynchronously and may take time to complete, for example, when a person performs an interaction on an iPhone widget on Mac. `Toggle` updates its appearance optimistically and indicates its new state immediately — without waiting for the result of the performed action.

To accurately reflect changed state for a person's action with a `Toggle`, update the widget with the result of the code you run in the app intent's `perform()` function. For example, if a person taps a toggle in a task manager widget to complete a task, the toggle immediately shows the task as complete — even though the logic in `perform()` still runs. To make sure the widget only shows a task that was actually completed, the `perform()` implementation can start the following sequence of actions:

1. Mark the task as completed in a database the widget shares with the app.
2. Run code to synchronize the database with your server.
3. Refresh the widget's timeline to show the correct state for the task when your server acknowledges the successful synchronization. The `Toggle` should show the correct state and not its earlier optimistic prediction.
4. In case of an error or server timeout, update the widget to reflect that the task isn't completed. For example, set the toggle's `isOn` property to `false` and show text in the widget that there's a synchronization error.

Review interactions in iPhone widgets on Mac

In the context of iPhone widgets on Mac, it's important to use the `invalidatableContent(_:)` view modifier on views if you add a `Button` to a widget and also understand the optimistic behavior of a `Toggle`. People can put iPhone widgets on their Mac desktop and in Notification Center. When a person interacts with the iPhone widget on Mac, the system sends the interaction to iPhone. On iPhone, the system performs the intent, generates a new timeline, and then sends the updated timeline to the iPhone widget on Mac. This process may take extra time. As a result, marking views that await updated data and making sure a toggle reflects synchronized data are especially important.

See Also

Interactivity

- Animating data updates in widgets and Live Activities

  Use SwiftUI animations to indicate data updates in your widgets and Live Activities.

- Linking to specific app scenes from your widget or Live Activity

  Add deep links to your widgets and Live Activities that enable people to open a specific scene in your app.