Sample Code

Highlighting app features with TipKit

Bring attention to new features in your app by using tips.

Download

iOS 18.0+ | iPadOS 18.0+ | Xcode 16.0+

Overview

With TipKit, you can teach people about a new feature in your app, or show them ways to accomplish a task faster. This sample guides you through the different types of tips you can add to your app, along with ways to control where and when they appear.

Each example highlights a different feature of the TipKit framework. The TipView and PopoverView examples show you how to add two different styles of tips to your app. Action buttons demonstrate how to direct people to information or options. The Parameters, Events, and Options examples show you various ways to control when your tips appear. And the Combining rules example shows you how to combine several conditions for when your tips display.

Related session from WWDC23

Session 10229: Make features discoverable with TipKit

Define tip content

The `Tip` protocol defines the text and imagery that tips use to describe what the features does, as well as the rules for when the tip appears. Each example defines a structure that conforms to the `Tip` protocol, and sets the properties that define the tip content.

```
struct InlineTip: Tip {
    var title: Text {
        Text("Save as a Favorite")
    }

    var message: Text? {
        Text("Your favorite backyards always appear at the top of the list.")
    }

    var image: Image? {
        Image(systemName: "star")
    }
}
```

Load tips when the app starts

Before tips display, they must load into the app. The sample initializes and loads all tips the app uses by calling `configure(_:)`. The best practice is to call this once per app session, for example, in the `init()` method of your app.

```
@main
struct TipKitExamples: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }

    init() {
        do {
            // Configure and load all tips in the app.
            try Tips.configure()
        }
        catch {
            print("Error initializing tips: \(error)")
        }
    }
}
```

> **Note**
>
> The sample project is currently configured to reset tip state each time the sample runs. To turn this off, and see how tips behave in production, comment out `try Tips.resetDatastore()` in the `setupTipsForTesting` method of `TipKitExamplesApp`.

Embed a tip directly in the view using inline tips

Use this style of tip whenever possible to avoid covering UI elements that people may want to interact with. Inline tips embed themselves directly into the view containing the feature they highlight. This style of tip causes the underlying view to alter its layout to accommodate the tip's appearance and dismissal.

To add an inline `TipView` to a view:

1. Create an instance of the tip content in the view containing the feature.
2. Place an instance of a `TipView` next to the feature to highlight, passing in the tip content, as well as an optional arrow pointing toward the feature.
3. Invalidate the tip after a person uses the feature. This is done programmatically by calling `invalidate(reason:)` on the tip content instance along with the reason you invalidated the tip. People can also click the close button in the upper right-hand corner of the tip view to dismiss the tip.

```
struct InlineView: View {
    // Create an instance of your tip content.
    let inlineTip = InlineTip()

    var body: some View {
        VStack(spacing: 20) {
            Text("A TipView embeds itself directly in the view. Make this style of t

            // Place your tip near the feature you want to highlight.
            TipView(inlineTip, arrowEdge: .bottom)
            Button {
                // Invalidate the tip when someone uses the feature.
                inlineTip.invalidate(reason: .actionPerformed)
            } label: {
                Label("Favorite", systemImage: "star")
            }

            Text("To dismiss the tip, tap the close button in the upper right-hand c
```

```
            Spacer()
        }
        .padding()
        .navigationTitle("Inline tip view")
    }
}
```

Display a tip on top of the underlying view using popover tips

Use this style of tip view if adjusting the underlying layout is undesirable or people don't need to interact with any controls the tip obscures. Popover view modifiers display the tip view on top of the current screen layout while obscuring UI elements underneath. If you use a popover, consider excluding imagery because the tip already points to the feature being highlighted.

To display a popover tip:

1. Define an instance of your tip content.
2. Place the tip on the feature to highlight using the `popoverTip(_:arrowEdge:action:)` view modifier, passing in the tip content.
3. Invalidate the tip after someone uses the feature, passing in an instance of `Invalidation Reason`.

```
struct PopoverView: View {
    // Create an instance of your tip content.
    let highlightTip = HighlightTip()

    var body: some View {
        // ...
        Image(systemName: "wand.and.stars")
            .imageScale(.large)
            // Place the tip on the feature to highlight.
            .popoverTip(highlightTip)
            .onTapGesture {
                // Invalidate the tip when someone uses the feature.
                tip.invalidate(reason: .actionPerformed)
            }
        // ...
        }
    }
}
```

Direct people to information or options with buttons

Buttons present people with additional information and options before they decide whether to use a new feature. For example, consider using buttons to redirect people to an area where they can learn more about what the given feature does. Or consider sending people to a settings screen if adjustments are necessary before a new feature can be used.

To add a button to a tip, the sample code defines actions using the `Action` type in the `actions` property of the tip content.

```
struct PasswordResetTip: Tip {
    var title: Text {
        Text("Need Help?")
    }

    var message: Text? {
        Text("Do you need help logging in to your account?")
    }

    var image: Image? {
        Image(systemName: "lock.shield")
    }

    var actions: [Action] {
        // Define a reset password button.
        Action(id: "reset-password", title: "Reset Password")
        // Define a FAQ button.
        Action(id: "faq", title: "View our FAQ")
    }
}
```

The sample code first checks the `action.id` of the action callback to see which button the user tapped. It then redirects users to either a password reset screen or a FAQ screen appropriately.

```
struct PasswordResetView: View {
    @Environment(\.openURL) var openURL

    // Create an instance of your tip content.
    let passwordResetTip = PasswordResetTip()

    var body: some View {
        VStack(spacing: 20) {
```

```
            Text("Use action buttons to link to more options. In this example, two a

            // Place your tip near the feature you want to highlight.
            TipView(passwordResetTip, arrowEdge: .bottom) { action in
                // Define the closure that executes when someone presses the reset b
                if action.id == "reset-password", let url = URL(string: "https://ifc
                    openURL(url) { accepted in
                        print(accepted ? "Success Reset" : "Failure")
                    }
                }

                // Define the closure that executes when someone presses the FAQ but
                if action.id == "faq", let url = URL(string: "https://appleid.apple.
                    openURL(url) { accepted in
                        print(accepted ? "Success FAQ" : "Failure")
                    }
                }
            }
            Button("Login") {
                // Perform login action.
            }
            Spacer()
        }
        .padding()
        .navigationTitle("Password reset")
    }
}
```

Display tips based on app state using parameters

This sample demonstrates how to use app state to display a tip. Define a variable in your tip structure to represent the app state to track. Wrap it in a `Parameter` property wrapper, and then define a `Rule` for when it displays using the #`Rule` macro in the `rules` property of the structure defining the tip. This sample only displays the tip when someone taps the button and logs in.

```
struct ParameterRuleTip: Tip {
    // Define the app state you want to track.
    @Parameter
    static var isLoggedIn: Bool = false

    var rules: [Rule] {
        // Define a rule based on the app state.
```

```
        #Rule(Self.$isLoggedIn) {
            // Set the conditions for when the tip displays.
            $0 == true
        }
    }
    // ...
}
```

> **Note**
>
> If you define no rules within a tip content structure, all tips display until dismissed or they exceed the threshold of their display frequency.

With this rule defined, the sample triggers a change of state when someone taps the button.

```
Button("Tap") {
    // Trigger a change in app state to make the tip appear or disappear.
    ParameterRuleTip.isLoggedIn.toggle()
}
```

This updates the view and displays the tip accordingly.

```
struct ParameterView: View {
    // Create an instance of your tip content.
    let parameterRuleTip = ParameterRuleTip()

    var body: some View {
        VStack(spacing: 20) {
            // ...
            // Place your tip near the feature you want to highlight.
            TipView(parameterRuleTip, arrowEdge: .bottom)
            Image(systemName: "photo.on.rectangle")
                .imageScale(.large)

            Button("Tap") {
                // Trigger a change in app state to make the tip appear or disappear.
                ParameterRuleTip.isLoggedIn.toggle()
            }
            // ...
        }
    }
```

```
}
```

> **Note**
>
> A tip dismissed by the user won't appear again until its datastore is reset. See the section on overriding tip eligibility rules for instructions on how to reset a tip's state.

The `Parameter` property wrapper also support types that conforms to the `Codable` and `Sendable` protocol. In this example, the tip displays if the plants tip has more than two favorites, and one of the favorites is a string with the value "Rose".

```
struct FavoritePlantTip: Tip {
    // Define a custom value type to store a list of plant names.
    struct FavoritePlants: Codable, Sendable {
        var plants: Set<String> = []

        var arrayValue: [String] {
            Array(plants)
        }

        mutating func setPlants(_ newValue: [String]) {
            plants = Set(newValue)
        }
    }

    // Reset to default value the first time it is referenced.
    @Parameter(.transient)
    static var favoritePlants: FavoritePlants = FavoritePlants(plants: ["Sunflower",

    var title: Text {
        Text("Explore Favorite Plants")
    }

    var message: Text? {
        Text("Discover your favorite plants and flowers.")
    }

    var image: Image? {
        Image(systemName: "leaf.fill")
    }

    // Tip will only display when there are 3 or more favorite plants and Rose has b
```

```
    var rules: [Rule] {
        // Display if more than two favorite plants are added.
        #Rule(FavoritePlantTip.$favoritePlants) {
            $0.plants.count >= 3
        }

        // Display if "Rose" is added as a favorite.
        #Rule(FavoritePlantTip.$favoritePlants) {
            $0.plants.contains("Rose")
        }
    }
}
```

Display tips based on user actions with events

To track and display tips based on user actions, define a `Event` constant with an `id` in your tip structure representing the user interaction you want to track. Then define a `Rule` for when the tip displays using a `#Rule` macro referencing the event to track. Set the conditions for when the tip displays within the `#Rule` macro closure. In this sample, the tip displays when the event occurs three or more times.

```
struct EventRuleTip: Tip {
    // Define the user interaction you want to track.
    static let didTriggerControlEvent = Event(id: "didTriggerControlEvent")

    var rules: [Rule] {
        // Define a rule based on the user interaction state.
        #Rule(Self.didTriggerControlEvent) {
            // Set the conditions for when the tip displays.
            $0.donations.count >= 3
        }
    }
    // ...
}
```

To trigger the event, the sample presents a button that displays the tip when tapped three times.

```
struct EventView: View {
    var body: some View {
        List {
            NavigationLink("Basic Event Tip") {
```

```
                EventRuleView()
            }

            NavigationLink("Event Tip using custom data type") {
                FoodDetailView()
            }
        }
        .navigationTitle("Event rules")
    }
}
```

Creating an event with an associated donation value

You can also create a `Event` property wrapper with associated donated values. These values hold additional information relevant to the tip itself.

The example below donates the `viewedSpecificFood` event with an associated donation value when the user action occurs. In this sample, the tip displays if the following rules are satisfied:

- The `viewedDetailView` event occurs at least once.
- The `viewedSpecificFood` event occurs with at least three distinct items.
- The `viewedSpecificFood` event occurs more than four times within the last hour.

```
struct FoodItem: View {
    @Binding
    var food: FoodEventTip.Item

    var body: some View {
        Button {
            food.isFavorite.toggle()

            // Donate to the event when the user action occurs.
            FoodEventTip.viewedSpecificFood.sendDonation(food)
        } label: {
            VStack {
                Text(food.name)
                Image(systemName: food.isFavorite ? "heart.fill" : "heart")
                    .foregroundColor(food.isFavorite ? .red : .primary)
            }
        }
    }
}
```

The example below creates a display rule for `FoodEventTip` based on the viewedSpecificFood event.

```
struct FoodEventTip: Tip {
    struct Item: Codable, Sendable {
        var name: String
        var isFavorite: Bool
    }

    // Event triggered when a user views a specific food item.
    static let viewedSpecificFood: Tips.Event<Item> = Tips.Event(id: "viewed-specifi

    static let viewedDetailView = Tips.Event(id: "FoodDetailViewDidOpen")

    var title: Text {
        Text("Save as a Favorite")
    }

    var message: Text? {
        Text("Tap on the button to favorite an item.")
    }

    var image: Image? {
        Image(systemName: "fork.knife")
    }

    var rules: [Rule] {
        #Rule(FoodEventTip.viewedDetailView) {
            // This rule checks if the user donated to the `FoodDetailViewDidOpen` e
            $0.donations.count >= 1
        }
        #Rule(FoodEventTip.viewedSpecificFood) {
            // The events donated must contain at least three distinct items with di
            // This ensures the user explored a variety of options before showing th
            $0.donations.smallestSubset(groupedBy: \.name).count > 1
        }
        #Rule(FoodEventTip.viewedSpecificFood) {
            // This rule checks if the user has donated to the `viewedSpecificFood`
            // within the last hour for favorited items.
            $0.donations.donatedWithin(.hour)
                .filter({ $0.isFavorite == true }).count > 4
        }
```

```
    }

    var options: [Option] {
        // Show this tip once.
        MaxDisplayCount(1)
    }
}
```

Control display frequency using options

Options control the frequency with which tips display. Define a `Option` in the `options` property of your tip structure and set the frequency for the tip display there.

```
struct OptionTip: Tip {
    // ...
    var options: [Option] {
        // Show this tip once.
        MaxDisplayCount(1)
    }
}
```

Then present the tip in a view. The first time the view appears, the tip displays once. Afterward, the tip no longer appears until reset or the app reinstalls on the device. See the section on overriding tip eligibility rules for instructions on how to reset a tip's state.

Combine parameters, events, and options for more complex interactions

To create more complex rules, combine parameters, events, and options to make tips appear. This sample defines one parameter-based rule to only show a tip when someone logs in, and another event-based rule that only displays a tip when someone enters the view three times. These rules logically AND together in the rules property of the tip structure.

```
struct ComboTip: Tip {
    // Define the app state you want to track.
    @Parameter
    static var isLoggedIn: Bool = false

    // Define the user interaction you want to track.
    static let enteredView = Event(id: "enteredView")
```

```
    // ...
    var rules: [Rule] {
        // Note: These rules AND together.
        // Define a parameter-based rule tracking app state.
        #Rule(Self.$isLoggedIn) {
            $0 == true
        }

        // Define an event-based rule tracking user state.
        #Rule(Self.enteredView) {
            $0.donations.count >= 3
        }
    }
}
```

The tip displays when the following two rules are met:

1. The user is logged in (triggered by tapping the Login button).

2. The user enters the view three times (triggered by navigating in and out of the view).

The sample tracks the number of times the page appears by donating to the event-based rule in the `onAppear(perform:)` view modifier.

```
.onAppear {
    // Donate to the event each time the view appears.
    ComboTip.enteredView.sendDonation()
}
```

Then, when the user logs in and the view appears three times, the tip displays.

```
struct ComboView: View {
    // Create an instance of your tip content.
    let comboTip = ComboTip()

    var body: some View {
        VStack(spacing: 20) {
            Text("You can combine parameters, events, and options to support more co

            // Place your tip near the feature you want to highlight.
            TipView(comboTip, arrowEdge: .bottom)
            Image(systemName: "star")
```

```
            Button(ContentView.isLoggedIn ? "Logout" : "Login") {
                ContentView.isLoggedIn.toggle()
            }

            Text("Entered view: \(ComboTip.enteredView.donations.count + 1) times")
            Text("For example, to make this tip appear, navigate in and out of the s
            Spacer()
        }
        .onAppear {
            // Donate to the event each time the view appears.
            ComboTip.enteredView.sendDonation()
        }
        .padding()
        .navigationTitle("Combining rules")
    }
}
```

Override eligibility rules to test tip appearance and display

When a tip is no longer eligible for display, it stops appearing in the app. While this might make sense in production, it can make testing tip appearance and placement challenging during development.

To make tip testing easier, the sample code uses various testing APIs to override a tip's eligibility rules, and hide or display tips regardless of their previous eligibility or status.

```
// Show all defined tips in the app.
Tips.showAllTipsForTesting()

// Show some tips, but not all.
Tips.showTipsForTesting([tip1, tip2, tip3])

// Hide all tips defined in the app.
Tips.hideAllTipsForTesting()

// Purge all TipKit-related data.
try Tips.resetDatastore()
```