Article

Making a configurable widget

Give people the option to customize their widgets by adding a custom app intent to your project.

Overview

To make the most relevant information easily accessible to people, widgets can provide customizable properties. For example, a person can select a specific stock for a stock quote widget, or enter a tracking number for a package-delivery widget. Widgets define customizable properties by using app intents, the same mechanism that Siri Suggestions and Siri Shortcuts use for customizing those interactions.

To add configurable properties to your widget:

1. Add custom app intent types that conform to `WidgetConfigurationIntent` to define the configurable properties to your Xcode project.
2. Specify an `AppIntentTimelineProvider` as your widget's timeline provider to incorporate the person's choices into your timeline entries.
3. Add code to your custom app intent types to provide the data if their properties rely on dynamic data.

If your app already supports Siri Suggestions or Siri Shortcuts and you have a custom app intent, you've already done most of the work already. Otherwise, consider leveraging the work you do for your widget to add support for Siri Suggestions or Siri Shortcuts. For more information on how to get the most from app intents, see App Intents.

> **Note**
> Prior to iOS 17, iPadOS 17, and macOS 14, configurable widgets used SiriKit Intents. For information on migrating your configurable widgets from the SiriKit Intents to the App Intents framework, see Migrating widgets from SiriKit Intents to App Intents.

Add a custom app intent to your project

To show the character's information, the person needs a way to select the character. The following code shows how to define a custom app intent to represent the choice the person makes:

```
struct SelectCharacterIntent: WidgetConfigurationIntent {
    static var title: LocalizedStringResource = "Select Character"
    static var description = IntentDescription("Selects the character

    @Parameter(title: "Character")
    var character: CharacterDetail

    init(character: CharacterDetail) {
        self.character = character
    }

    init() {
    }
}
```

The static `title` property describes the action the intent enables the person to take. Use a title case string that combines a verb with a noun. Set the static `description` to a localized, human-readable string that describes the intent.

To add parameters to the intent, add one or more `@Parameter` property wrappers. WidgetKit uses the parameter type information to automatically create the user interface for setting the parameter. For example, if the type is `String`, the person enters a string value. If the type is an `Int`, they use a number pad. For a parameter that is a predefined, static, list of values, define a custom type that conforms to `EnumSubset`.

> **Note**
> The order of the parameters in the intent determines the order in which they appear when a person edits your widget.

In the example above, the parameter uses a custom `CharacterDetail` type the app defines to represent a character in the game. To use a custom type as an app intent parameter, it must conform to `AppEntity`. To implement the `Character Detail` parameter type, the game-status widget uses a structure that exists in the game's project. This structure defines a list of available characters and their details, as follows:

```
struct CharacterDetail: AppEntity {
    let id: String
    let avatar: String
    let healthLevel: Double
    let heroType: String
    let isAvailable = true

    static var typeDisplayRepresentation: TypeDisplayRepresentation =
    static var defaultQuery = CharacterQuery()

    var displayRepresentation: DisplayRepresentation {
        DisplayRepresentation(title: "\(avatar) \(id)")
    }

    static let allCharacters: [CharacterDetail] = [
        CharacterDetail(id: "Power Panda", avatar: "🐼", healthLevel:
        CharacterDetail(id: "Unipony", avatar: "🦄", healthLevel: 0.6
        CharacterDetail(id: "Spouty", avatar: "🤽", healthLevel: 0.83
    ]
}
```

Because characters might vary from game to game, the intent generates the list dynamically at runtime. WidgetKit uses the app entity's `defaultQuery` property to access the dynamic values, as described below.

If your widget includes nonoptional parameters, you must supply a default value. For types such as `String`, `Int`, or enumerations that use `AppEnum`, one option is to supply a default value as follows:

```
@Parameter(title: "Title", default: "A Default Title")
var title: String
```

A second option is to use a query type that implements `defaultResult()`, as shown in the next section.

For custom intents with parameters that conform to `AppEntity`, implement initializer methods to provide default values for the nonoptional parameters, such as the `init(character:)` method in the code for `SelectCharacterIntent` shown above. In your timeline provider's `placeholder()` method, use one of these initializer methods to initialize the app intent that you pass to the timeline entry. These methods enable you to customize the placeholder with values that might be different from the default, if needed.

Implement a query to provide dynamic values

Some of the tasks that an `EntityQuery` performs include:

- Mapping `AppEntity` identifiers to the corresponding entity instances.
- Providing a list of suggested values when a person edits a widget.
- Specifying a default value for a parameter.

When a person edits a widget with a custom intent that provides dynamic values, the system invokes the query object's `suggestedEntities()` method to get the list of possible choices.

In the entity query, the result is an array of all the `CharacterDetail` types available.

```
struct CharacterQuery: EntityQuery {
    func entities(for identifiers: [CharacterDetail.ID]) async throws
        CharacterDetail.allCharacters.filter { identifiers.contains(0
    }

    func suggestedEntities() async throws -> [CharacterDetail] {
        CharacterDetail.allCharacters.filter { $0.isAvailable }
    }

    func defaultResult() async -> CharacterDetail? {
        try? await suggestedEntities().first
    }
}
```

With the configuration of the custom app intent done, a person can edit the widget to select a specific character to display.

After the person edits the widget and selects a character, the next step is to incorporate that choice into the widget's display.

Handle customized values in your widget

To support configurable properties, a widget uses the `AppIntentTimeline Provider` configuration. For example, the character-details widget defines its configuration as follows:

```
struct CharacterDetailsWidget: Widget {
    var body: some WidgetConfiguration {
        AppIntentConfiguration(
            kind: kind,
            intent: SelectCharacterIntent.self,
            provider: CharacterDetailProvider()) { entry in
            CharacterDetailView(entry: entry)
        }
        .configurationDisplayName("Character Details")
        .description("Displays a character's health and other details
        .supportedFamilies([.systemSmall, .systemMedium, .systemLarge
    }
}
```

The `SelectCharacterIntent` parameter determines the customizable properties for the widget. The configuration uses the `CharacterDetailProvider` to manage the timeline events for the widget. For more information about timeline providers, see Keeping a widget up to date.

After a person edits the widget, WidgetKit passes the customized values to the provider when requesting timeline entries. You typically include relevant details from the intent in the timeline entries the provider generates. In the following example, the provider passes the `defaultQuery` to look up the `Character Details` using the character's `id` in the intent, and then creates a timeline with an entry containing the character's detail.

```
struct CharacterDetailProvider: AppIntentTimelineProvider {
    func placeholder(in context: Context) -> SelectCharacterIntent {
        // Create the timeline and return it. The .never reload polic
        // that the containing app uses WidgetCenter methods to reloa
        // widget's timeline when the details change.
        let entry = CharacterDetailEntry(date: Date(), detail: config
        let timeline = Timeline(entries: [entry], policy: .never)
        return timeline
    }
}
```

When you include the customized values in the timeline entry, your widget's view can display the appropriate content.

Access customized values in your app

When a person taps on a widget to open your app, WidgetKit passes the customized values to your app in a `URLContext`. In your app's code that handles the user activity, such as `onContinueUserActivity(_:perform:)` for a SwiftUI app or `scene(_:continue:)` for a UIKit app, use the `widget ConfigurationIntent(fr)` method to access the widget's intent.

To access the intent of any widget that the user has installed, use `getCurrent Configuration(_:)` to fetch the `WidgetInfo` objects. Iterate over the `WidgetInfo` objects and call `widget(ofConfigurationIntent:of:)`.

Offer configurable widgets and complications on Apple Watch

Like widgets in iOS and macOS, watch complications use app intents to display user-configurable data, and implementing configurable complications and widgets in watchOS works the same as in iOS or macOS. However, you have a choice whether to offer a user-configurable complication or widget or allow people to configure it themselves.

In your `AppIntentTimelineProvider` code, implement the `recommendations()` and return:

- An array of `AppIntentRecommendation` objects you create using your custom app intents to offer a preconfigured complication or widget
- An empty array (`return []`) to let people configure the complication or widget

> **Note**
> watchOS 11 and older don't have an interface for configuring widgets or complications. If you support older watchOS versions, offer preconfigured complications and widgets.

If you offer a preconfigured complication or widget, and your app receives new data that's relevant to your recommended widget configurations, invalidate the now outdated recommendations by calling `invalidateConfiguration Recommendations()`. This invalidation tells WidgetKit to get new recommendations for your preconfigured complications and widgets. When you invalidate the recommendations for preconfigured complications, make sure you return updated `AppIntentRecommendation` objects in the `recommendations()` callback.

See Also

Configurable widgets

Migrating widgets from SiriKit Intents to App Intents
Configure your widgets for backward compatibility.

`struct AppIntentConfiguration`
An object describing the content of a widget that uses a custom intent to provide user-configuration options.

`struct WidgetInfo`
A structure that contains information about user-configured widgets.