Article

Dispatching intents to handlers

Provide SiriKit with an intent handler capable of handling a specific intent.

Overview

When a user makes a request of your app as an intent, SiriKit needs a handler that conforms to the corresponding intent handling protocol. Each intent object has an associated protocol based on the name of the intent. For example, to handle an `INSendMessageIntent`, provide an instance of a type that conforms to the `INSendMessageIntentHandling` protocol. The protocol defines the methods that your handler implements to resolve any intent parameters and to let SiriKit know how your app handled the intent.

Provide a handler in your intents app extension

An intents app extension helps you respond to a person's request quickly, without loading the entire app. For intents you support in an extension, SiriKit loads your intents app extension and creates an instance of your `INExtension` subclass. Implement the `handler(for:)` method on your extension object to provide SiriKit with the handlers that you use to handle specific intents.

Provide a handler in your app

If you support an intent directly in your app, the system asks your app delegate for the handler. In a macOS app that uses AppKit, implement `application(_:handlerFor:)` on your `NSApplicationDelegate`. In an iOS app (or an app built with Mac Catalyst), implement `application(_:handlerFor:)` on your `UIApplicationDelegate`. If you're using SwiftUI, use `UIApplicationDelegateAdaptor` or `NSApplicationDelegateAdaptor` to include an app delegate in your app.

Important

On iOS (and Mac Catalyst), the system only invokes the delegate method in apps that support multiple scenes. For more information, see Specifying the scenes your app supports.

Determine which handler type to return

You can define a separate type for handling each intent class to keep your code organized. When the system asks your app delegate or Intents app extension for an intent handler, check the type of the intent you receive to determine which type of handler to return. Return a valid new handler that conforms to the relevant protocol for all intents that your app or extension supports.

The following code listing shows logic for an app or extension that supports four different intents. The code handles three message intents with one class, `MyMessageHandler`, which implements `INSendMessageIntentHandling`, `INSearchForMessagesIntentHandling`, and `INSetMessageAttributeIntentHandling`. The code provides a separate handler, `MyCallHandler`, which implements `INStartCallIntentHandling`. After checking the type of the provided intent object, the method creates and returns an instance of the type capable of handling that intent.

```
// Inside your implementation of handler(for:) or application(_:handlerFor:),
// use the type of the intent to determine which handler to provide.
switch intent {
    case is INSendMessageIntent,
           is INSearchForMessagesIntent,
           is INSetMessageAttributeIntent:
        return MyMessageHandler()

    case is INStartCallIntent:
        return MyCallHandler()
    default:
        // SiriKit doesn't call this method with intents you don't support.
        return nil
}
```

Handle an intent in multiple versions of iOS

In iOS 14 and later, you can resolve, confirm, and handle long-running or memory-intensive intents directly in your app. In iOS 13 and earlier, SiriKit required you resolve and confirm all intents in an Intents app extension, then send long-running intents to your app for the final handling step. To continue supporting these intents on earlier versions of iOS, you can implement the methods to resolve and confirm the intent in both your app delegate and Intents app extension. When your extension confirms the intent, respond with a `continueInApp` or `handleInApp` response code so the system sends the intent to the app to handle the intent.

See Also

Articles

Adding User Interactivity with Siri Shortcuts and the Shortcuts App

Add custom intents and parameters to help users interact more quickly and effectively with Siri and the Shortcuts app.

Defining Relevant Shortcuts for the Siri Watch Face

Inform Siri when your app's shortcuts may be useful to the user.

Deleting Donated Shortcuts

Remove your donations from Siri.

Improving Siri Media Interactions and App Selection

Fine-tune voice controls and improve Siri Suggestions by sharing app capabilities, customized names, and listening habits with the system.

Improving interactions between Siri and your messaging app

Donate app-specific content, use Siri's contact suggestions, and adopt the latest platform features to create a more consistent messaging experience.

Registering Custom Vocabulary with SiriKit

Register your app's custom terminology, and provide sample phrases for how to use your app with Siri.

Confirming the Details of an Intent

Perform final validation of the intent parameters and verify that your services are ready to fulfill the intent.

Handling an Intent

Fulfill the intent and provide feedback to SiriKit about what you did.

Resolving the Parameters of an Intent

Validate the parameters of an intent and make sure that you have the information you need to continue.

Generating a List of Ride Options

Generate ride options for Maps to display to the user.

Handling the Ride-Booking Intents

Support the different intent-handling sequences for booking rides with Shortcuts or Maps.

Donating Reservations

Inform Siri of reservations made from your app.

Specifying Synonyms for Your App Name

Provide alternative names for your app that are more familiar or easier for users to speak.

Intent Phrases

The keys that you include in your global vocabulary file to show how users engage your app from Siri.

Localizing Your Vocabulary for Chinese Dialects

Apply emphasis markers to your pronunciation tips to assist Siri with Chinese dialects.