Article

Responding to invocations

Add code to respond to invocations and offer a focused launch experience.

Overview

A great launch experience that helps the user complete a task quickly is the key to a successful App Clip. Upon launch, your App Clip needs to be aware of the user's context and update its UI accordingly. For example, an App Clip might provide functionality to order donuts from a local donut shop with multiple physical locations. Instead of making the user select a location before displaying the donut menu, the App Clip recognizes the user's context and updates its UI accordingly. The user doesn't have to select a location, and can complete their task — ordering donuts — with fewer taps.

To enable the App Clip to respond to the user's context upon launch, the system makes the *invocation URL* available to the App Clip. Configuring the launch experience of your App Clip and choosing invocation URLs are key tasks when creating an App Clip. However, configuring invocation URLs in App Store Connect isn't enough to provide a streamlined launch experience. You also need to add code to your App Clip that handles the invocation URL and updates its UI.

Leveraging the invocation URL

Both the App Clip and the full app need to respond to the invocation URL and update their UI to help the user quickly complete their task at hand. Consider the example of a donut shop with multiple physical locations. The user shouldn't have to select the location that matches their context before they can order pastries. Instead, the App Clip can make use of the invocation URL to update its UI to display the shop's menu right away. For example, use an invocation URL with additional URL parameters that help the App Clip recognize the user's context and update its UI:

1. If you don't use the default App Clip link that App Store Connect generates for a default experience, add entries for each domain that launches the App Clip to the `Associated`

    `Domains Entitlement`. For the example described above, you add `example.com`. You typically do this when you add an App Clip target to your Xcode project.

2. Use invocation URLs that contain additional parameters; for example, `https://example.com/location1`, `https://example.com/location2`, and so on.
3. On launch, respond to the URLs by persisting any unsaved data in case the user switches from one location to another. Then, update the UI to match the new location.

If you configure a generic URL, both your App Clip and your full app must always be able to handle the URL, even if you only intend to use it as a prefix for your actual invocation URLs. For example, you can register `https://example.com/` as part of an advanced App Store experience. For the actual invocations — such as in your App Clip Codes — you can use URLs like `https://example.com/location1` or `https://example.com/location2`. However, your App Clip and full app must also be able to handle `https://example.com`.

For additional information on associating your App Clip with your website and configuring experiences, refer to Associating your App Clip with your website and Configuring App Clip experiences.

Access the invocation URL

To respond to an invocation, you need to access the invocation URL so your App Clip can tailor itself to the experience. To access the URL, query the `NSUserActivity` object the system passes to the App Clip upon launch. If a user installs the full app, it replaces the App Clip, and the system launches it with each invocation. To offer the same functionality and provide an equivalent user experience as the App Clip, the full app also needs to handle all invocation URLs. In most cases, it makes sense to share the code that handles your URL between full app and App Clip, and also have the App Clip store state information in a shared container. This way, the full app can access any data stored by the App Clip, and, upon launch, take the user to the part of the app that best matches their context.

To access the `NSUserActivity` object on launch:

- If you use the SwiftUI lifecycle, apply the `onContinueUserActivity(_:perform:)` modifier. For additional information on implementing lifecycle callbacks for your SwiftUI app, refer to Restoring your app’s state with SwiftUI.
- If you use a scene-based lifecycle, implement the `scene(_:willConnectTo:options:)` and `scene(_:willContinueUserActivityWithType:)` callbacks. On the first launch, the system calls the `scene(_:willConnectTo:options:)` callback. If the app or App Clip is suspended in memory and the user launches it, the system calls `scene(_:willContinueUserActivityWithType:)`.
- If you use the `UIApplicationDelegate` object to respond to lifecycle events, be sure to implement the `application(_:continue:restorationHandler:)` callback. Note that

you don't have access to the NSUserActivity object in application(_:didFinish LaunchingWithOptions:).

On launch, confirm that the invocation is of type NSUserActivityTypeBrowsingWeb, then access the URL that the system passes to the App Clip. The following code shows a function that extracts components from the invocation URL:

```
func respondTo(_ activity: NSUserActivity?) {

    // Guard against faulty data.
    guard let activity, activity.activityType == NSUserActivityTypeBrowsingWeb else
    let incomingURL = activity.webpageURL
    guard let components = NSURLComponents(url: incomingURL, resolvingAgainstBaseURL

    // Update the user interface based on URL components passed to the App Clip or t
}
```

For more information about responding to lifecycle events, refer to Managing your app's life cycle. For general information on handling links, refer to Supporting universal links in your app.

To learn more about sharing data between the App Clip and the full app, refer to Sharing data between your App Clip and your full app.

Ensure your code handles all invocations

A fast, consistent user experience that fits the user's context is key to the App Clip user experience, and it's why designing and configuring your App Clip experiences is so important. As a result, spending time to write resilient code, handling all possible invocation URLs, and testing the code is key. Make sure you guard against faulty data and handle the following scenarios:

- Invocations from the Maps app and location-based suggestions from Siri Suggestions use the URL you register for an App Clip experience as the invocation URL.
- Invocations from Messages app or your website use the site's URL as the invocation URL.
- If a user returns to a previously launched App Clip from the App Library or Spotlight, the App Clip uses the invocation URL that it previously used to launch the App Clip.
- If a user returns to a previously launched App Clip from a notification or the App Switcher, the App Clip launches without an invocation URL. To address this case, save the state of your App Clip before the user leaves it, and restore the saved state if the invocation URL isn't available upon launch.

For more information on verifying invocation URLs, refer to Testing the launch experience of your App Clip.

See Also

Launch

Associating your App Clip with your website

Enable the system to verify your App Clip to support invocations from your website and devices running iOS 16.3 or earlier.

Supporting invocations from your website and the Messages app

Display a Smart App Banner and the App Clip card on your website that people tap to launch your App Clip, and add support for invocations from the Messages app.

Confirming a person's physical location

Add code to quickly confirm a person's physical location while respecting their privacy.

Launching another app's App Clip from your app

Enable people to launch another app's App Clip from your app with App Clip links and offer a rich preview of it with the Link Presentation framework.

`class APActivationPayload`

Information that's passed to an App Clip on launch.

`NSAppClip`

A collection of keys that an App Clip uses to get additional capabilities.