Sample Code

Add Functionality to Finder with Action Extensions

Implement Action Extensions to provide quick access to commonly used features of your app.

Download

macOS 10.14+ | Xcode 11.3+

Overview

Action Extensions can let users access features of your app directly from macOS. For example, if your app contains the functionality to perform image processing, an Action Extension could allow users to process their images directly from Finder. This sample code project creates three Action Extensions, two for processing images and one for processing text. One extension also presents a user interface and all three are available from both the Touch Bar and Finder Preview pane.

The project consists of four targets:

- A macOS app target that has no functionality, but would normally contain your main application.
- `ThumbnailAction`, an Action Extension target that takes images of any type as inputs and outputs thumbnail versions of them. This extension preserves the original input files and creates thumbnails as new files.
- `RemoveOpacityAction`, an Action Extension target that takes PNG images as inputs and replaces them with opaque versions of the same images. During processing, this extension prompts the user for a color to draw as the background behind the transparent image.
- `UppercaseAction`, an Action Extension target that takes text as input and converts it to be uppercase.

The extensions in this sample all accept file data from Finder, process it, and output the data back to Finder.

By default, new Action Extensions are not enabled in macOS. After running this sample code project, enable the extensions by selecting the “Create Thumbnail”, “Make PNG Opaque”, and “Convert Text to Upper Case” checkboxes in the Actions, Finder, and Touch Bar categories of the Extensions System Preferences.

Accept File Data

The system makes Action Extensions available depending on the type of data that the user has selected in Finder. For example, if the user selects a PNG file, Action Extensions that support images are shown. Each of the extensions in this sample operate on different kinds of input content:

- `ThumbnailAction` operates on *any* kind of image.
- `RemoveOpacityAction` operates specifically on PNG images.
- `UppercaseAction` operates on text.

The `NSExtensionActivationRule` key in the `Info.plist` of the associated target determines the conditions under which extensions are available. The value for this key can either be a dictionary, or a string.

`ThumbnailAction` and `UppercaseAction` are both configured with dictionary values. `ThumbnailAction` specifies that it accepts image data with the `NSExtensionActivationSupportsImageWithMaxCount` key and `UppercaseAction` specifies that it accepts text data with the `NSExtensionActivationSupportsText` key.

For more information on valid dictionary keys for `NSExtensionActivationRule`, see the App Extension Keys reference.

`RemoveOpacityAction` uses a `NSPredicate` query as a string value for the `NSExtensionActivationRule`. The query in this sample project filters images to only be valid if they are in PNG format.

```
SUBQUERY (
    extensionItems,
    $extensionItem,
    SUBQUERY (
        $extensionItem.attachments,
        $attachment,
        ANY $attachment.registeredTypeIdentifiers UTI-CONFORMS-TO "public.png"
    ).@count == $extensionItem.attachments.@count
```

For more information on defining extension activation rules, see the App Extension Programming Guide.

Process Input Attachments

When a user invokes an Action Extension, the system instantiates the `NSExtensionPrincipalClass` specified in the `Info.plist` and asks it to handle the request through the `NSExtensionRequestHandling` protocol.

The system calls the `beginRequest(with:)` method and passes in an `NSExtensionContext`. To access the input files, use the `attachments` property of the `NSExtensionItem` contained in the `inputItems` array.

Once processing completes, pass output files back to the system by creating new `NSItemProvider` objects and registering them with `registerFileRepresentation(forTypeIdentifier:fileOptions:visibility:loadHandler:)`.

```
let itemProvider = NSItemProvider()
itemProvider.registerFileRepresentation(
    forTypeIdentifier: kUTTypePNG as String, fileOptions: [.openInPlace],
    visibility: .all, loadHandler: { completionHandler in
        guard let sourceImage = NSImage(contentsOf: sourceUrl) else { return nil }
        let thumbnailImage = sourceImage.thumbnailImage
        let thumbnailUrl = self.thumbnailUrl(for: sourceUrl)
        thumbnailImage.savePNGToDisk(at: thumbnailUrl)
        completionHandler(thumbnailUrl, false, nil)
        return nil
    }
)
```

To obtain the correct directory path for writing output files, use `url(for:in:appropriateFor:create:)`, passing the `itemReplacementDirectory` constant.

```
let itemReplacementDirectory = try FileManager.default.url(
    for: .itemReplacementDirectory, in: .userDomainMask,
    appropriateFor: URL(fileURLWithPath: NSHomeDirectory()), create: true)
let thumbnailFilename = sourceUrl.deletingPathExtension().lastPathComponent + " Thum
```

Finally, signal that the action is complete by calling `completeRequest(returningItems:completionHandler:)`.

> **Note**
>
> If you wish to preserve the input files instead of replacing them, it is important that `NSItemProvider` objects for the original input files are also included in the returned array of item providers. The `ThumbnailAction` target in this sample code project demonstrates passing back both the original input files as well as new thumbnail files.

Process Text Data

The `UppercaseAction` extension accepts any text, either supplied as attachments in the same way as `ThumbnailAction` and `RemoveOpacityAction`, or from other sources such as a standard `NSTextField` control. Check the `attributedContentText` property for input from text fields.

If `attributedContentText` is empty, check for input in the `attachments` property of `NSExtensionItem`.

```
let outputItem = NSExtensionItem()
outputItem.attributedContentText = NSAttributedString(string: inputContent.string.up
context.completeRequest(returningItems: [outputItem], completionHandler: nil)
```

Present a User Interface

If the action requires user interaction then the extension may present a user interface. Set the `NSExtensionPointIdentifier` key in the `Info.plist` to `com.apple.ui-services` and ensure that the principal class is a subclass of `NSViewController`. If the extension does not require user interaction, specify `com.apple.services` and have the principal class conform to `NSExtensionRequestHandling`.

> **Note**
>
> When creating a new Action Extension target, Xcode prompts you to decide if your extension will present a user interface. Based on that choice, Xcode sets appropriate initial values for `NSExtensionPointIdentifier` and `NSExtensionPrincipalClass`.

Extensions that present a user interface show their view controller's view immediately after the extension calls `beginRequest(with:)` and should input files with the `extensionContext` property of `NSViewController`.

Once the extension's user interface has completed any required user interaction for the task, process the files in the normal way by calling `completeRequest(returningItems:completionHandler:)`.

To cancel the task, use the `cancelRequest(withError:)`. Pass a `NSUserCancelledError` if the user canceled the task, or an error caused the failure.

```
let cancelError = NSError(domain: NSCocoaErrorDomain, code: NSUserCancelledError, us
context.cancelRequest(withError: cancelError)
```

Support the Touch Bar and Finder Preview Pane

Users may also access Action Extensions from the Touch Bar and from the Quick Actions menu in Finder's Preview pane. Set both `NSExtensionServiceAllowsFinderPreviewItem` and `NSExtensionServiceAllowsTouchBarItem` to YES in the `Info.plist` to support the Touch Bar and Preview Pane.

Each extension must also have a template icon for use in the Quick Actions menu and the Finder Preview pane. Specify the icon with the `NSExtensionServiceTouchBarIconName` key of the `Info.plist` for each target. For more information, see the Human Interface Guidelines. The color of the Touch Bar button can also be customized using the `NSExtensionServiceTouchBarBezelColorName` key. By default, this is a `TouchBarBezel` color but can be any color in the asset catalog for the target.

For information on how to add Quick Actions to the Touch Bar see the Automator User Guide.

See Also

Quick Actions

`NSExtensionServiceAllowsFinderPreviewItem`

A Boolean value indicating whether the extension appears in the Finder Preview pane and Quick Actions menu.

`NSExtensionServiceFinderPreviewLabel`

A name for display when the extension appears in the Finder Preview pane and Quick Actions menu.

`NSExtensionServiceFinderPreviewIconName`

The name of an icon for display when the extension appears in the Finder Preview pane and Quick Actions menu.

`NSExtensionServiceAllowsTouchBarItem`

A Boolean value indicating whether the extension appears as a Quick Action in the Touch Bar.

`NSExtensionServiceTouchBarLabel`

A name for display when the extension appears as a Quick Action in the Touch Bar.

`NSExtensionServiceTouchBarIconName`

The name of an icon for display when the extension appears as a Quick Action in the Touch Bar

`NSExtensionServiceTouchBarBezelColorName`

The color to use for the bezel around the extension when it appears as a Quick Action in the Touch Bar.