Sample Code

Bringing Photos picker to your SwiftUI app

Select media assets by using a Photos picker view that SwiftUI provides.

Download

iOS 16.0+ | iPadOS 16.0+ | macOS 13.0+ | watchOS 9.0+ | Xcode 14.0+

Overview

This sample shows how to use the SwiftUI Photos picker to browse and select a photo from your photo library. The app displays an interface that allows you to fill in customer profile details, and includes a button to select a photo from the Photos library. The sample explores how to retrieve a SwiftUI image by using `Transferable` — a new SwiftUI protocol you use to move data.

> **Note**
>
> This sample code project is associated with WWDC22 session 10023: What's new in the Photos picker.

Configure the Sample Code Project

Before you run the sample code project in Xcode, ensure you're using iOS 16 or later, watchOS 9 or later, macOS 13 or later.

You must use a physical device when building the watchOS target in Xcode.

Add a Photos picker view

To display the picker, the sample adds a `PhotosPicker` view as an overlay to a profile image. The view provides a label that describes the action of choosing an item from the photo library. The

sample displays assets that match the image type. In iOS 16 or later, `PHPickerFilter` contains new filters for `bursts`, `cinematicVideos`, and `depthEffectPhotos`.

```
CircularProfileImage(imageState: viewModel.imageState)
    .overlay(alignment: .bottomTrailing) {
        PhotosPicker(selection: $viewModel.imageSelection,
                     matching: .images,
                     photoLibrary: .shared()) {
            Image(systemName: "pencil.circle.fill")
                .symbolRenderingMode(.multicolor)
                .font(.system(size: 30))
                .foregroundColor(.accentColor)
        }
        .buttonStyle(.borderless)
    }
```

Load an image from the selection

The sample contains an `PhotosPickerItem` that contains the selection. The selection only contains a placeholder object. Some large files may take a long time to download, so the sample shows an inline loading indicator instead of an indicator that blocks execution.

```
@Published var imageSelection: PhotosPickerItem? = nil {
    didSet {
        if let imageSelection {
            let progress = loadTransferable(from: imageSelection)
            imageState = .loading(progress)
        } else {
            imageState = .empty
        }
    }
}
```

To load the asset data, `PhotosPickerItem` adopts `Transferable`. The sample attempts to retrieve the SwiftUI `Image` from the item. A failure can occur when the system attempts to retrieve the data. For example, if the picker tries to download data from iCloud Photos without a network connection.

```
private func loadTransferable(from imageSelection: PhotosPickerItem) -> Progress {
    return imageSelection.loadTransferable(type: ProfileImage.self) { result in
```

```
        DispatchQueue.main.async {
            guard imageSelection == self.imageSelection else {
                print("Failed to get the selected item.")
                return
            }
            switch result {
            case .success(let profileImage?):
                self.imageState = .success(profileImage.image)
            case .success(nil):
                self.imageState = .empty
            case .failure(let error):
                self.imageState = .failure(error)
            }
        }
    }
}
```

In advanced data transfer cases, an app can control the type of data to load by defining a custom model object that conforms to the `Transferable` protocol. The sample creates a model `ProfileImage` to handle loading a `DataRepresentation` of an image, and converts it to a `UIImage` or `NSImage`.

```
struct ProfileImage: Transferable {
    let image: Image

    static var transferRepresentation: some TransferRepresentation {
        DataRepresentation(importedContentType: .image) { data in
        #if canImport(AppKit)
            guard let nsImage = NSImage(data: data) else {
                throw TransferError.importFailed
            }
            let image = Image(nsImage: nsImage)
            return ProfileImage(image: image)
        #elseif canImport(UIKit)
            guard let uiImage = UIImage(data: data) else {
                throw TransferError.importFailed
            }
            let image = Image(uiImage: uiImage)
            return ProfileImage(image: image)
        #else
            throw TransferError.importFailed
        #endif
        }
```

```
    }
}
```

When handling many items at the same time, or large assets, use `FileRepresentation` to load assets as files and reduce memory usage. When loading assets as files, copy them to an app directory and remove them when they're no longer needed.

See Also

Sample code

{} Browsing and Modifying Photo Albums

Help users organize their photos into albums and browse photo collections in a grid-based layout using PhotoKit.

{} Selecting Photos and Videos in iOS

Improve the user experience of finding and selecting assets by using the Photos picker.

{} Implementing an inline Photos picker

Embed a system-provided, half-height Photos picker into your app's view.

{} Creating a Slideshow Project Extension for Photos

Augment the macOS Photos app with extensions that support project creation.