Sample Code

Synchronizing files using file provider extensions

Make remote files available in macOS and iOS, and synchronize their states by using file provider extensions.

Download

iOS 18.0+ | iPadOS 18.0+ | Mac Catalyst 18.0+ | macOS 15.0+ | Xcode 16.0+

Overview

More and more people use cloud storage to store and share digital assets. Developers can seamlessly integrate their cloud storage service with macOS and iOS by creating file provider extensions to expose items on their server as files or folders in macOS and iOS, and synchronize their states.

This sample code project implements file provider extensions that sync content between a file storage server and both macOS and iOS, so the files on the server are available on the local system and accessible through user interactions and system-provided APIs. It also demonstrates how to synchronize the states of the file provider items between the server and the systems, and how to enhance the user experience using decorations and custom actions. It includes a macOS app and an iOS app that implement a management interface for users to log in to the server and try out other features. The macOS app implements a simplified file storage server with HTTP-based communication to act as the backend of the file provider extensions.

Configure the sample code project

To build the sample app, perform the following steps in Xcode:

1. Set the developer team for all targets to let Xcode automatically manage the provisioning profile. See Assign a project to a team for details.

2. Replace the App Group container identifier `group.com.example.apple-samplecode.FruitBasket` with one specific to your team for the entire project. The identifier points to an App Group container that the apps and extensions use to share data. It occurs in the entitlements for the `FruitBasket`, `Provider`, `Action`, and `FruitBasket-iOS` targets, and in `Defaults.swift` and `StandaloneServer.swift`. You can search `group.com.example.apple-samplecode.FruitBasket` in Xcode's Find navigator and change all the occurrences (except the ones in this Readme file). See Configuring App Groups for more details.

To try out the macOS app:

1. Run the `FruitBasket` target on your Mac.

2. Add a domain by clicking Add Domain in `FruitBasket.app`, giving it a name, and clicking Save.

3. Find the domain from the sidebar of Finder, and start to use it by dragging a file onto the location.

To try out the sample app in iOS:

1. Launch the macOS app first. The macOS app includes a server, which the iOS app requires to function.

2. Run the `FruitBasket-iOS` target on your iOS device.

3. Tap the Settings button (the gear icon) in the iOS app, and select the macOS server. The macOS and iOS devices need to be on the same network for the app to find the server.

4. Tap the Add button (+) to add a domain. A new domain is in a disabled state by default.

5. Launch the Files app, find the domain in the list of locations, and then tap to enable it.

Understand the targets of the sample project

This sample project consists of the following targets:

- `FruitBasket` is the macOS app target. Users use the app to create a file provider domain and log in to the remote server from a Mac. Running the app launches the standalone file storage server that acts as the backend of the file provider extension. To be functional, both the macOS and the iOS file provider extensions require the server to be running on the same network.

- `FruitBasket-iOS` is the iOS app target. Users use the app to create a file provider domain and log in to the remote server for an iOS device.

- `Provider` is the file provider extension target. It uses `Provider.entitlement` when building with the macOS app, and uses `Provider-iOS.entitlement` with the iOS app. The apps embed the extension using the Embed Without Signing setting. The target contains only an empty `main.swift`, which acts as the entry point for the file provider extensions. The concrete implementation of the file provider extensions is in the `Extension` target. Separating the implementation into a different target allows unit testing. Developers can't run unit tests against an app extension directly.
- `Extension` creates `extension.framework`, which implements all the file provider extension logics, and works in both macOS and iOS.
- `Action` creates a macOS file provider UI extension that handles document conflicts and account authentication.
- `Server` creates a simplified file storage server that acts as the backend of the file provider extensions.
- `Common` creates `common.framework`, which provides some utilities for the entire project.

Manage file provider domains

A file provider domain (NSFileProviderDomain) represents a location on the sidebar of Finder (macOS) or Files (iOS). A file provider extension can have multiple domains to provide different item sets and apply per-domain account authentication. macOS and iOS create one extension instance for each domain, so all the domains work independently.

A file provider extension by default doesn't have a domain, and so doesn't show up in the location list of Finder or Files before users add one. In the sample, users add a domain by tapping the Add button (+) in `FruitBasket-iOS.app` (or Add Domain in `FruitBasket.app`), which triggers the following code:

```
func addDomain(displayName: String, accountIdentifier: String) {

    let domain = NSFileProviderDomain(identifier: NSFileProviderDomainIdentifier(raw
    NSFileProviderManager.add(domain) { _ in }
}
```

The domain name needs to be unique. Adding a domain with a name that conflicts with an existing domain triggers an NSFileWriteFileExistsError error.

Users can delete a domain by selecting it and tapping the Remove Domain button, which eventually triggers remove(_:completionHandler:) or remove(_:mode:completionHandler:). In iOS, deleting a domain deletes all the files. In macOS, apps can keep downloaded files or files that

contain unsynchronized local changes by specifying a removal mode (NSFileProviderManager.DomainRemovalMode).

Enumerate file provider items

After users add a domain, the system immediately asks the file provider extension for an enumerator (an object that implements NSFileProviderEnumerator) by calling enumerator(for:request:), and then triggers enumerations to request the items.

File provider extensions need to implement `enumerator(for:request:)` to provide the enumerator the system requests based on the `containerItemIdentifier` parameter, which can be .rootContainer, .workingSet, .trashContainer, or a provider-supplied item identifier. All the enumerators need to support the initial enumeration by implementing enumerateItems(for:startingAt:), which reports the appropriate items based on the starting page (`startingAt`). Working set enumerators need to support change enumeration by implementing enumerateChanges(for:from:), which reports the appropriate changes based on the sync anchor (`syncAnchor`). If the system queries the working set enumerator with a sync anchor that's already aged out, the enumerator needs to report NSFileProviderError.Code.syncAnchorExpired so the system restarts the sync operation from the beginning.

The system can request an enumerator and trigger an enumeration whenever necessary. For example, when a user browses a folder for the first time, the system requests an enumerator with the folder's item identifier and triggers an enumeration for the folder. When getting a `PushKit` notification, it requests an enumerator with `.workingSet` (if there isn't a cached one) and triggers a working set enumeration.

This sample implements the enumeration logic in the following class:

```
class ItemEnumerator: NSObject, NSFileProviderEnumerator
```

File provider extensions need to define a working set that contains the items of particular interest to the users. See Defining Your File Provider's Content for more information. The system indexes the items the working set enumerator provides so they are searchable.

In this sample, the working set is the same as the item set in the root container (which includes every item under the root container, recursively). The extensions don't maintain a local copy of the working set. When the system requests items, the extensions make a JSON call to retrieve the relevant information from the server, and pass the result to the system. Real-world file provider extensions can define a working set smaller than the root container item set, and use a different enumerator for the working set enumeration, if desired.

This sample doesn't track the materialized items in the local system. Real-world file provider extensions can do that and report the items to the server so the server can maintain the

materialized item set and avoid pushing the changes on items irrelevant to the devices. See Synchronizing the File Provider Extension for more details.

Synchronize files between the device and the server

The state of a file provider item can change on both the device side and the server side. When that happens, the file provider extension needs to synchronize the state.

When a change happens on the device side, the system notifies the file provider extension by calling the NSFileProviderReplicatedExtension methods with the changes, and the extension passes the changes to the server to synchronize the states. Because of the involvement of networking requests, all the methods are asynchronous, have a `completionHandler`, and return a Progress object to the system for progress tracking or cancellation. The extension needs to call the completion handler to report the result back to the system. For more information about syncing files, see Sync files to the cloud with FileProvider on macOS and Bring desktop class sync to iOS with FileProvider.

This sample implements all the `NSFileProviderReplicatedExtension` methods in `Extension.swift`. The following code example shows the method that the system calls to notify the file provider about the changes on an item's content or metadata from the device:

```
public func modifyItem(_ item: NSFileProviderItem, baseVersion version: NSFileProvid
                       contents newContents: URL?, options: NSFileProviderModifyItem
                       completionHandler: @escaping (NSFileProviderItem?, NSFileProv
```

The method makes a JSON call to pass the changes to the server. If the server can't fulfill all the changes, this method reports the fields associated with the unsynchronized changes back to the system using the `remainingFields`. If the item content changes on the server side, it asks the system to refetch the content by setting `shouldFetchContent` to `true`.

When users or system APIs access a dataless item (a file that has the metadata only), the system calls the method in the code example below to tell the file provider to fetch the file content:

```
public func fetchContents(for itemIdentifier: NSFileProviderItemIdentifier,
                          version requestedVersion: NSFileProviderItemVersion?,
                          request: NSFileProviderRequest,
                          completionHandler: @escaping (URL?, NSFileProviderItem?, E
```

This method similarly makes a JSON call to download the file content, and returns a `Progress` object with a `cancellationHandler` that calls the `completionHandler`.

To work with the POSIX read operations that read only part of a file, the file provider extensions in this sample support partial download by implementing the following NSFileProviderPartialContentFetching method:

```
public func fetchPartialContents(for itemIdentifier: NSFileProviderItemIdentifier,
                                 version requestedVersion: NSFileProviderItemVersion?,
                                 request: NSFileProviderRequest,
                                 minimalRange range: NSRange,
                                 aligningTo alignment: Int,
                                 options: NSFileProviderFetchContentsOptions,
                                 completionHandler:
                                 @escaping (URL?, NSFileProviderItem?, NSRange,
                                            NSFileProviderMaterializationFlags,
                                            Error?) -> Void) -> Progress
```

When changes happen on the server side, there are two ways to notify the file provider extensions:

- The server sends a PushKit notification of `fileProvider` push type with the item identifier. (The only valid container in a push payload is "`NSFileProviderWorkingSetContainerItemIdentifier`".) When receiving the notification, the local system (macOS or iOS) triggers an enumeration so the file provider extensions can fetch the changes from the server. See Using push notifications to signal changes and Sending Push Notifications Using Command-Line Tools for more information.
- The app detects that there are pending server-side changes (by using remote push notifications or polling, for example), and triggers an enumeration by calling signalEnumerator(for:completionHandler:). See Setting Up a Remote Notification Server for more information.

In this sample, the server doesn't implement the `PushKit` support (to reduce the complexity of the sample code configuration). It posts an `.itemsChanged` notification using DistributedNotificationCenter when changes happen. The macOS app, which runs on the same computer as the server, observes the notification, and triggers an enumeration when getting one. The iOS app provides the Signal domain button in the domain detail view for users to trigger an enumeration manually.

Enhance file providers with decorations, custom actions, and file provider UI extensions

File provider extensions can provide decorations and custom actions for an item. Finder and Files display the decorations on the item, and present the custom actions as contextual menu items when users Control-click or long press the item.

To provide decorations for a file provider item, the file provider extensions in this sample do the following:

- Declare the item decorations using the `NSExtension` > `NSFileProviderDecorations` entry in the `Info.plist` file in the `Provider` target, which includes adding the `Identifier`, `Label`, `Category`, and `BadgeImageType` keys for each decoration.
- Implement NSFileProviderItemDecorating in the file provider item class NSFileProviderItem to return the decoration identifiers (NSFileProviderItemDecorationIdentifier) of the current item.

The following code example returns the appropriate decorations based on the item state:

```
var decorations: [NSFileProviderItemDecorationIdentifier]?
```

Similarly, the file provider extensions do the following to provide custom actions:

- Declare the custom actions using the `NSExtension` > `NSExtensionFileProviderActions` entry in the `Info.plist` file in the `Provider` target, which includes adding the NSExtensionFileProviderActionIdentifier, NSExtensionFileProviderActionName, NSExtensionFileProviderActionActivationRule, and `Comment` keys for each action. See Adding Actions to the Context Menu for information about using `NSExtensionFileProviderActionActivationRule` to enable or disable a custom action based on the current context.
- Implement NSFileProviderCustomAction in the file provider extension class to perform the actions when users select the contextual menu items.

The following code example performs the actions:

```
public func performAction(identifier actionIdentifier: NSFileProviderExtensionActior
                          onItemsWithIdentifiers itemIdentifiers: [NSFileProviderIte
                          completionHandler: @escaping (Error?) -> Void) -> Progress
```

When performing an action, file provider extensions don't present any additional UI elements. If necessary, apps can present the UI when users trigger a custom action by creating a file provider UI extension. See Adding Actions to the Context Menu for more details.

This sample implements a file provider UI extension to handle document conflicts and account authentication. For conflict handling, the file provider UI extension creates a custom action named *Other versions* (see the `NSExtension` > `NSExtensionFileProviderActions` entry in the `Info.plist` file in the `Action` target), and presents the conflict view controller (`ConflictViewController.swift`) when users trigger the action.

The system automatically presents the principal view controller (NSExtensionPrincipalClass) of the file provider UI extension for the NSFileProviderError.Code.notAuthenticated error. When that

happens, the file provider UI extension in this sample presents the account authentication view controller (`AuthenticationViewController`), as the following code example shows:

```
public override func prepare(forError error: Error) {
    prepare(AuthenticationViewController())
}
```

Handle errors elegantly

The File Provider framework defines the common errors in FSFileProviderError.Code. If an error happens when running a method that the system calls, file provider extensions need to report that by passing the error to the completion handler.

Among the predefined errors, the following four are resolvable: .notAuthenticated, .serverUnreachable, .insufficientQuota, .cannotSynchronize.

When encountering a resolvable error, the system throttles the operation until something (most likely the app or extension) calls signalErrorResolved(:completionHandler:) to signal that the user or the server resolves the error. Any other error, including crashes of the extension process, is transient, and causes the system to retry the modification.

The file provider UI extension in this sample calls the method with `.notAuthenticated` when the user resolves the authentication issue.

```
try await manager.signalErrorResolved(NSFileProviderError(.notAuthenticated))
```

See Also

Essentials

Synchronizing the File Provider Extension

Keep the local and remote copies of your File Provider extension's content in sync.

Setting the Finder Sidebar Icon

Specify a standard or custom symbol as a sidebar icon.