Sample Code

Synchronizing documents in the iCloud environment

Manage documents across multiple devices to create a seamless editing and collaboration experience.

Download

iOS 14.0+ | iPadOS 14.0+ | Xcode 12.4+

Overview

As technology advances, more and more people generate their digital assets from different devices, and expect those assets to synchronize seamlessly. To support such use cases, apps need to discover the assets, as well as their changes, from all the devices, and present the user with a consistent view.

This sample demonstrates how to discover and synchronize documents in the iCloud environment, and manage them to achieve high performance and a low memory footprint. These *documents* can be the digital assets or any user data.

The sample also demonstrates how to publish an iCloud container to iCloud Drive so that the user can access the container's `Documents` folder from other apps. Additionally, it shows how to support the Open-in-Place feature, which allows the user to launch an app by tapping a document in Files, and then edit it directly.

Configure the sample code project

Before building the sample, perform the following steps in Xcode:

1. In the General pane of the `SimpleiCloudDocument` target, update the Bundle Identifier field with a new identifier.

2. In the Signing & Capabilities pane, select the applicable team from the Team drop-down menu to let Xcode automatically manage the provisioning profile. See Assign a project to a team for details.

3. Make sure the iCloud capability is present and the iCloud Documents box is in a selected state, then select the iCloud container with your bundle identifier from step 1 from the Containers list. If the container doesn't exist, click the Add button (+), enter the container name (iCloud.<bundle identifier>), and click OK to let Xcode create the container and associate it with the app.

4. If you prefer to use a different container, select it from the Containers list, specify the container identifier when creating the `MetadataProvider` instance in the `viewDidLoad` method of the `MainViewController` class. An iCloud container identifier is case-sensitive and must begin with "`iCloud.`".

5. Find the `NSUbiquitousContainers` entry in the `Info.plist` file, and change the iCloud container identifier there as well.

Before running the sample on a device, configure the device as follows:

1. Log in with an Apple ID. For documents to synchronize across devices, the Apple ID must be the same on all devices.

2. Choose Settings > Apple ID > iCloud, and turn on iCloud Drive, if it is off.

3. Prepare some pictures in the Photo Library to use in the sample.

Publish an iCloud container to iCloud Drive

Publishing an iCloud container to iCloud Drive makes the container's `Documents` folder appear in iCloud Drive so the user can access the folder from other apps. Follow these steps to publish a container:

1. Provide the container's metadata by adding an `NSUbiquitousContainers` entry to the `Info.plist` file like the example code below demonstrates.

2. Increase the bundle version by changing the Build field in the General pane of the Xcode target, or the `CFBundleVersion` entry in the `Info.plist` file. The new value must be larger than the previous value when using the `compare(_:options:range:)` method with the `numeric` option to compare, and must only contain numeric (0 – 9) and period (.) characters. The system only updates an app's iCloud container metadata when detecting a new version, so perform this step every time the metadata changes.

3. Make sure the `Documents` folder exists in the iCloud container and has at least one document.

The `NSUbiquitousContainers` entry of the sample is as follows:

```
    <key>NSUbiquitousContainers</key>
    <dict>
        <key>iCloud.com.example.apple-samplecode.SimpleiCloudDocument</key>
        <dict>
            <key>NSUbiquitousContainerIsDocumentScopePublic</key>
            <true/>
            <key>NSUbiquitousContainerName</key>
            <string>SimpleiCloudDocument</string>
            <key>NSUbiquitousContainerSupportedFolderLevels</key>
            <string>ANY</string>
        </dict>
    </dict>
```

Support Open-in-Place

The Open-in-Place feature allows the user to launch an app by tapping a document of the type the app owns. After opening it, the app can change the document directly without copying it to the app's sandbox container. Follow these steps to implement the feature:

1. Declare and export a document type for the app by adding the `CFBundleDocumentTypes` and `UTExportedTypeDeclarations` `Info.plist` entries. Make sure the type conforms to at least `public.content` and `public.data` in the `UTTypeConformsTo` entry so that the other system components, like `UIActivityViewController`, recognize it.

2. Add the `LSSupportsOpeningDocumentsInPlace` key to the `Info.plist` file, and set the value to YES.

3. Implement the `scene(_:openURLContexts:)` method of the `UISceneDelegate` protocol to accept the document.

Apps need to wrap the code that accesses the passed-in URL with the `startAccessingSecurityScopedResource` and `stopAccessingSecurityScopedResource` methods if the URL is outside of their sandbox. This sample doesn't explicitly do that because it accesses the URL via `UIDocument`, which handles security-scoped bookmarks automatically.

Discover documents in an iCloud container

iOS apps use `NSMetadataQuery` rather than file system APIs to discover documents in an iCloud container. When an app creates an iCloud document on one device, iCloud first synchronizes the document metadata to the other devices to tell them about the existence of the document. Then, depending on the device types, iCloud may or may not continue to synchronize the document data. For iOS devices, iCloud doesn't synchronize the document data until an app asks (either explicitly or implicitly). When an iOS app receives a notification that a new document exists, the document data may not physically exist on the local file system, so it isn't discoverable with file system APIs.

To watch the metadata changes in the iCloud container, the sample creates an `NSMetadataQuery` object. It uses the following code to configure and start the query to gather the changes of documents that are in the iCloud container and have an `.sicd` extension name.

```
metadataQuery.notificationBatchingInterval = 1
metadataQuery.searchScopes = [NSMetadataQueryUbiquitousDataScope, NSMetadataQueryUbi
metadataQuery.predicate = NSPredicate(format: "%K LIKE %@", NSMetadataItemFSNameKey,
metadataQuery.sortDescriptors = [NSSortDescriptor(key: NSMetadataItemFSNameKey, asce
metadataQuery.start()
```

A query has two phases when gathering the metadata: the initial phase that collects all currently matching results, and a second phase that gathers live updates. It posts an `NSMetadataQueryDidFinishGathering` notification when it finishes the first phase, and an `NSMetadataQueryDidUpdate` notification each time an update occurs. To avoid potential conflicts with the system, disable the query update when accessing the results, and enable it after finishing the access, as the following example shows:

```
func metadataItemList() -> [MetadataItem] {
    var result = [MetadataItem]()
    metadataQuery.disableUpdates()
    if let metadatItems = metadataQuery.results as? [NSMetadataItem] {
        result = metadataItemList(from: metadatItems)
    }
    metadataQuery.enableUpdates()
    return result
}
```

Manage a large data set

Documents in this sample could potentially contain many images, and the images might be large. To load image data only when necessary, and release the data immediately after using it, the `Document` class provides public methods for directly accessing the images in the document package. As an example, the following method retrieves a full image asynchronously:

```
func retrieveImageAsynchronously(with imageName: String, completionHandler: @escapir
    performAsynchronousFileAccess {
        let imageFileURL = self.fileURL.appendingPathComponent(imageName)
        let fileCoordinator = NSFileCoordinator(filePresenter: self)
        fileCoordinator.coordinate(readingItemAt: imageFileURL, options: .withoutCha
            if let imageData = try? Data(contentsOf: newURL), let image = UIImage(da
                completionHandler(image)
            } else {
                completionHandler(nil)
            }
        }
    }
}
```

When directly manipulating the files in the document package, the sample calls the `performAsynchronousFileAccess(_:)` method to serialize the file access in the background queue, and uses `NSFileCoordinator` to coordinate the reading or writing.

Likewise, to avoid the default implementation that loads image data to `FileWrapper` objects and keeps it in memory, the sample overrides the `save(to:for:completionHandler:)` method to directly remove or add image files when updating a document.

```
override func save(to url: URL, for saveOperation: UIDocument.SaveOperation, complet
    if saveOperation != .forCreating {
        print("\(#function)")
        return performAsynchronousFileAccess {
            let fileCoordinator = NSFileCoordinator(filePresenter: self)
            fileCoordinator.coordinate(writingItemAt: self.fileURL, options: .forMe
                let success = self.fulfillUnsavedChanges()
                self.fileModificationDate = Date()
                if let completionHandler = completionHandler {
                    DispatchQueue.main.async {
                        completionHandler(success)
                    }
                }
            }
        }
    }
    super.save(to: url, for: saveOperation, completionHandler: completionHandler)
}
```

Resolve version conflicts

In the iCloud environment, the user can edit a document from different devices. Depending on networking conditions and the timing of synchronization, that may trigger version conflicts. Apps that provide support for iCloud documents need to resolve these conflicts, and remove the obsolete versions so they don't consume the user's iCloud storage.

To create a document conflict with the sample:

1. Run the sample on two iOS devices with Internet connections that use the same Apple ID to log in to iCloud.
2. Create a document with several images on one device, and watch the document synchronize with the other device.
3. Turn on Airplane mode on both devices to disconnect them from the Internet.
4. Change the document on both devices by adding some images on one device, removing some images from the other device, and then saving the changes.
5. Turn off Airplane mode on both devices at the same time to connect them back to the Internet.
6. If no conflict occurs, repeat steps 3–5. After detecting a conflict, the sample enables the `Conflicts` item on the toolbar, and changes its color to red, so users can resolve the conflict.

Handling version conflicts in document-based apps is straightforward because `UIDocument` does most of the heavy lifting. When a conflict occurs, `UIDocument` detects it and posts a `stateChangedNotification` notification, which apps can observe and then implement their conflict resolution strategy.

The sample resolves a conflict by selecting the version that has the most recent `modificationDate` and removing all others. It uses file coordination to assess the version information of an iCloud document to avoid potential conflicts with the system.

```
private func resolveConflictsAsynchronously(document: Document, completionHandler: (
    DispatchQueue.global().async {
        NSFileCoordinator().coordinate(writingItemAt: document.fileURL,
                                       options: .contentIndependentMetadataOnly, err
            let shouldRevert = self.pickLatestVersion(for: newURL)
            completionHandler?(shouldRevert)
        }
    }
}
```

See Also

Documents

`class UIDocument`

An abstract base class for managing discrete portions of your app’s data.

`class UIManagedDocument`

A managed document object that integrates with Core Data.