Sample Code

Sharing Core Data objects between iCloud users

Use Core Data and CloudKit to synchronize data between devices of an iCloud user and share data between different iCloud users.

Download

iOS 17.4+ | iPadOS 17.4+ | macOS 14.4+ | watchOS 9.4+ | Xcode 15.3+

Overview

More and more people own multiple devices and use them for digital asset sharing and collaboration. They expect seamless data synchronization and sharing experiences with robust privacy and security features. Apps can support such use cases by implementing a data-sharing flow using Core Data and CloudKit.

This sample code project demonstrates how to use Core Data and CloudKit to share photos between iCloud users. People who share photos, called *owners*, can create a share, send an invitation, manage the permissions, and stop the sharing. People who accept the share, called *participants*, can view and edit the photos, or stop participating in the share.

Configure the sample code project

Open the sample code project in Xcode. Before building it, perform the following steps:

1. In the Signing & Capabilities pane of the `CoreDataCloudKitShare` target, set the developer team to let Xcode automatically manage the provisioning profile. See [Assign a project to a team](doc://com.apple.documentation/xcode/preparing-your-app-for-distribution#Assign-the-project-to-a-team> for details.

2. In the iCloud section, select an empty iCloud container from the Containers list. If there isn't an empty container, click the Add button (+), enter a container name, and click OK to let Xcode create the container and associate it with the app. An iCloud container identifier is case-sensitive, and must be unique and begin with "`iCloud.`".

3. Specify the same team and iCloud container for all other targets.

4. Specify the same iCloud container for the `gCloudKitContainerIdentifier` variable in `PersistenceController.swift`.

5. In the Info pane of the `CoreDataCloudKitShareOnWatch` target, change the value of the `WKCompanionAppBundleIdentifier` key to `<The iOS app bundle ID>`.

To run the sample app on a device, configure the device as follows:

1. Log in with an Apple ID. For an Apple Watch, open the Apple Watch app on the paired iPhone, log in at General > Apple ID, and confirm the Apple ID showing up in the Settings app on the watch.

2. For an iOS device, confirm that iCloud is on for the app at Settings > Apple ID > iCloud > Apps Using iCloud.

3. After running the app on the device, confirm that Allow Notifications is on for the app at Settings > Notifications. For an Apple Watch, use the Apple Watch app on the paired iPhone to confirm the settings.

For more information about the project configuration, see Setting Up Core Data with CloudKit.

Create the CloudKit schema

CloudKit apps require a schema to declare the data types they use. When apps create a record in the CloudKit development environment, CloudKit automatically creates the record type if it doesn't exist. In the production environment, CloudKit doesn't have that capability, nor does it allow removing an existing record type or field, so after finalizing the schema, developers need to deploy it to the production environment. Without this step, apps that work in the production environment, like the ones people download from the App Store or TestFlight, can't communicate with the CloudKit server. For more information, see Deploying an iCloud Container's Schema.

Apps that use `NSPersistentCloudKitContainer` can call `initializeCloudKitSchema(options:)` to create the CloudKit schema that matches their Core Data model, or keep it up to date every time their model changes. The method works by creating fake data for the record types and then deleting it, which can take some time and blocks the other CloudKit operations. Apps must not call it in the production environment or in the normal development process that doesn't include model changes.

To create the CloudKit schema for this sample app, select `InitializeCloudKitSchema` from Xcode's target menu and run the target. Having a target dedicated on CloudKit schema creation

separates the `initializeCloudKitSchema(options:)` call from the normal flow. After running the target, use CloudKit Console to ensure each Core Data entity and attribute has a CloudKit counterpart. See Reading CloudKit Records for Core Data for the mapping rules.

For apps that use the CloudKit public database, use CloudKit Console to manually add the `Queryable` index for the `recordName` field, and the `Queryable` and `Sortable` indexes for the `modifiedAt` field for all record types, including the CDMR type that Core Data generates to manage many-to-many relationships.

For more information, see Creating a Core Data Model for CloudKit.

Try out the sharing flow

To create and share a photo using the sample app, follow these steps:

1. Prepare two iOS devices, A and B, and log in to each device with a different Apple ID.
2. Use Xcode to build and run the sample app on the devices.
3. On device A, tap the Add button (+) to add a photo to the Core Data store.
4. Touch and hold the photo to display the context menu, and then tap New Share to present the sharing UI.
5. Follow the UI to send a link to the Apple ID on device B. Use iMessage if you can because it's easier to set up.
6. After receiving the link on device B, tap it to accept and open the share, which launches the sample app and shows the photo.

> **Note**
>
> It may take some time for one user to see changes from the other. CloudKit isn't for real-time synchronization. For apps that use CloudKit, the system determines when to synchronize data. This helps balance the use of system resources and achieve the best overall user experience. There is no API for apps to configure the timing for the synchronization.

To discover more features of the sample app:

- On device A, add another photo, touch and hold it, tap Add to Share, and then tap the trailing icon of the share. The photo soon appears on device B.
- On device B, touch and hold the photo, tap Participants, and then tap the Remove Me icon to stop the participation. The photo disappears.

- Tap the Manage Shares icon, and then tap a trailing icon of the share to present and use the sharing management UI.

Set up the Core Data stack

Every CloudKit container has a private database and a shared database. To mirror both of them, the sample app sets up a Core Data stack with two stores, sets one store's database scope to `.private` and the other to `.shared`, and then uses `affectedStores` or `assign(_:to:)` to specify the target store for data fetching or saving.

When setting up the store description, the sample app enables persistent history tracking and turns on remote change notifications by setting the `NSPersistentHistoryTrackingKey` and `NSPersistentStoreRemoteChangeNotificationPostOptionKey` options to `true`. Core Data relies on the persistent history to track the store changes, and the sample app updates its UI when remote changes occur.

```
privateStoreDescription.setOption(true as NSNumber, forKey: NSPersistentHistoryTrack
privateStoreDescription.setOption(true as NSNumber, forKey: NSPersistentStoreRemoteC
```

To synchronize data through CloudKit, apps need to use the same CloudKit container. The sample app explicitly specifies the same container for its iOS, macOS, and watchOS apps when setting up the CloudKit container options.

```
let cloudKitContainerOptions = NSPersistentCloudKitContainerOptions(containerIdenti
```

Share a Core Data object

Sharing a Core Data object between iCloud users includes creating a share (`CKShare`) from the owner side, accepting the sharing invitation from the participant side, and managing the share from both sides. Owners can stop their sharing or change the share permission. Participants can stop their participation. The sample app uses the following system sharing UI to implement the flow:

- It uses `ShareLink` to start a new sharing and send the invitation. (Non-SwiftUI apps use `UIActivityViewController` or `NSSharingServicePicker`.)
- It uses `UICloudSharingController` or `NSSharingService` to manage a share. (Apps that use the Shared with You framework can use `SWCollaborationView` if the UI is appropriate. For more information, see Adding shared content collaboration to your app.)

`ShareLink` requires the sharing object be `Transferable`. The `Photo` class in this sample conforms to the protocol by implementing `transferRepresentation` to provide a `CKShare`

`TransferRepresentation` instance, which is based on a new share it creates by calling `share(_:to:completion:)`.

`NSPersistentCloudKitContainer` uses CloudKit zone sharing to share objects. Each share has its own record zone on the CloudKit server. CloudKit has a limit on how many record zones a database can have. To avoid reaching the limit over time, the sample app provides an option for users to share an object by adding it to an existing share, as the following example shows:

```
func shareObject(_ unsharedObject: NSManagedObject, to existingShare: CKShare?,
                 completionHandler: ((_ share: CKShare?, _ error: Error?) -> Void)?
```

The system sharing UI may change the share and save it directly to the CloudKit server. Since iOS 16.4, iPadOS 16.4, macOS 13.3, and watchOS 9.4, `NSPersistentCloudKitContainer` automatically observes the changes and updates the share it maintains. To support earlier systems that can't upgrade to the latest versions, implement the relevant methods of `UICloudSharingControllerDelegate` or `NSCloudSharingServiceDelegate` to update the share `NSPersistentCloudKitContainer` maintains.

The sample app doesn't interact with the system sharing UI for other purposes. Apps that need to do so can create a `CKSystemSharingUIObserver` object and provide a closure for `systemSharingUIDidSaveShareBlock` and `systemSharingUIDidStopSharingBlock` to detect and react to the changes. For systems where `CKSystemSharingUIObserver` is unavailable, use `UICloudSharingControllerDelegate` or `NSCloudSharingServiceDelegate`.

`NSPersistentCloudKitContainer` doesn't support cross-share relationships. That is, it doesn't allow relating objects associated with different shares. When sharing an object, `NSPersistentCloudKitContainer` moves the entire object graph, which includes the object and all its relationships, to the share's record zone. When a participant stops participating in a share, `NSPersistentCloudKitContainer` deletes the object graph from the shared persistent store.

> **Note**
>
> For more information about Core Data and CloudKit sharing, see Build apps that share data through CloudKit and Core Data and What's new in CloudKit.

Detect relevant changes by consuming store persistent history

When importing data from CloudKit, `NSPersistentCloudKitContainer` records the changes on Core Data objects in the store's persistent history, and triggers remote change notifications

(`.NSPersistentStoreRemoteChange`) so apps can keep their state up-to-date. The sample app observes the notification and performs the following actions in the notification handler:

- Gathers the relevant history transactions (`NSPersistentHistoryTransaction`), and notifies the views when remote changes happen. The changes on shares don't generate any transactions.
- Merges the transactions to the `viewContext` of the persistent container, which triggers a SwiftUI update for the views that present photos. Views relevant to shares fetch the shares from the stores, and update the UI.
- Detects the new tags, and handles duplicates, if necessary.

To process the persistent history more effectively, the sample app:

- Maintains the token of the last transaction it consumes for each store, and uses it as the starting point of the next run.
- Maintains a transaction author, and uses it to filter the transactions irrelevant to `NSPersistentCloudKitContainer`.
- Only fetches and consumes the history of the relevant persistent store.

The following code sets up the history fetch request (`NSPersistentHistoryChangeRequest`):

```
let lastHistoryToken = historyToken(with: storeUUID)
let request = NSPersistentHistoryChangeRequest.fetchHistory(after: lastHistoryToken)
let historyFetchRequest = NSPersistentHistoryTransaction.fetchRequest!
historyFetchRequest.predicate = NSPredicate(format: "author != %@", TransactionAutho
request.fetchRequest = historyFetchRequest

if privatePersistentStore.identifier == storeUUID {
    request.affectedStores = [privatePersistentStore]
} else if sharedPersistentStore.identifier == storeUUID {
    request.affectedStores = [sharedPersistentStore]
}
```

The persistent history stays on the device as a part of the Core Data store, and accumulates over time. Apps that have a large data set can purge it. To do so, observe `eventChanged Notification` to determine the start date of the last successful `.export` event, and then purge the history that occurs sometime before that date. The *sometime* needs to be long enough for the history to become irrelevant, which can be several months for apps that people use on a regular basis. Apps generally only need to purge the history several times a year.

> **Note**
>
> `NSPersistentCloudKitContainer` relies on the persistent history to determine the data it needs to export. The history remains relevant before `NSPersistentCloudKitContainer` finishes processing it. Purging the history that is still relevant invalidates some internal state, and triggers a `reset` operation that synchronizes the store with the CloudKit server truth.

For more information about persistent history processing, see Consuming Relevant Store Changes.

Remove duplicate data

In the CloudKit environment, duplicate data is sometimes inevitable because:

- Different peers can create the same data. In the sample app, owners can share a photo with a permission that allows participants to tag it. When owners and participants simultaneously create the same tag, a duplicate occurs.
- Apps rely on some initial data and there's no way to allow only one peer to preload it. Duplicates occur when multiple peers preload the data at the same time.

To remove duplicate data (or *deduplicate*), apps need to implement a way that allows all peers to eventually reserve the same object and remove others. The sample app does so in the following way:

1. It gives each tag a universally unique identifier (UUID). Tags that have the same name (but different UUIDs) and are associated with the same share (and are, therefore, in the same CloudKit record zone) are duplicates, so only one of them can exist.
2. It detects new tags from CloudKit by looking into the persistent history each time a remote change notification occurs. Deduplication only applies to the private persistent store because the user may not have permission to change the shared persistent store.
3. For each new tag, it fetches the duplicates from the same persistent store, and sorts them by their UUID so the tag with the lowest UUID goes first.
4. It picks the first tag as the one to reserve and marks the others as `deduplicated`. Because each UUID is globally unique and each peer picks the first tag, all peers eventually reserve the same tag, which is the one that has the globally lowest UUID.
5. It removes the deduplicated tags sometime later.

When detecting duplicate tags, the sample app doesn't delete them immediately. It waits until the next `eventChangedNotification` occurs, and only removes the tags with a `deduplicated Date` that's sometime before the last successful export and import event. This allows enough time

for `NSPersistentCloudKitContainer` to synchronize the relationships of the deduplicated tags, and the app to establish the relationships for the tag it reserves.

The following code implements the deduplication process:

```
func deduplicateAndWait(tagObjectIDs: [NSManagedObjectID])
```

The following code shows how the app determines the deduplicated tags it can safely remove:

```
@objc
func containerEventChanged(_ notification: Notification)
```

Implement a custom sharing flow

Apps can implement a custom sharing flow when the system sharing UI is unavailable or doesn't fit. The sample app performs the following tasks so users can share photos from watchOS:

1. It creates a share using `share(_:to:completion:)` when an owner shares a photo.
2. It configures the share with appropriate permissions, and adds participants for a share. A share is private if its `publicPermission` is `.none`. For shares that have a public permission more permissive than `.none` (called *public shares*), users can participate by tapping the share link, so there's no need to add participants beforehand. The sample app looks up participants using `fetchParticipants(matching:into:completion:)`, configures the participant permission using `CKShare.Participant.permission`, and adds it to the share using `addParticipant(_:)`.
3. It allows users to deliver the share URL (`CKShare.url`) to a participant using `ShareLink`.
4. It implements `userDidAcceptCloudKitShare(with:)` to accept the share using `acceptShareInvitations(from:into:completion:)`. After the acceptance synchronizes, the photo and its relationships are available in the participant's shared persistent store.
5. It manages the participants using `addParticipant(_:)` and `removeParticipant(_:)` from the owner side, or stops the sharing or participation by calling `purgeObjectsAndRecordsInZone(with:in:completion:)`. (The purge API deletes the zone from CloudKit, and also the object graph from the Core Data store. Apps that need to keep the object graph can make a deep copy, ensure the new graph doesn't connect to any share, and save it to the store.)

> **Note**
>
> To be able to accept a share when people tap a share link, an app's `Info.plist` file needs to contain the `CKSharingSupported` key with a value of `true`.

In this process, the sample app calls `persistUpdatedShare(_:in:completion:)` when it changes the share using CloudKit APIs for `NSPersistentCloudKitContainer` to update the store. The following code shows how the app adds a participant:

```
participant.permission = permission
participant.role = .privateUser
share.addParticipant(participant)

self.persistentContainer.persistUpdatedShare(share, in: persistentStore) { (share, e
    if let error = error {
        print("\(#function): Failed to persist updated share: \(error)")
    }
    completionHandler?(share, error)
}
```

See Also

CloudKit mirroring

Mirroring a Core Data store with CloudKit

Back user interfaces with a local replica of a CloudKit private database.

Synchronizing a local store to the cloud

Share data between a user's devices and other iCloud users.

`class NSPersistentCloudKitContainer`

A container that encapsulates the Core Data stack in your app, and mirrors select persistent stores to a CloudKit private database.

`class NSPersistentCloudKitContainerOptions`

An object that customizes how a store description aligns with a CloudKit database.