Sample Code

Sharing CloudKit Data with Other iCloud Users

Create and share private CloudKit data with other users by implementing the sharing UI.

Download

iOS 13.0+ | iPadOS 13.0+ | Mac Catalyst 13.0+ | Xcode 13.0+

Overview

As technology advances, people collaborate with others through cloud-based apps more than ever. They can share digital assets with friends, or invite their colleagues living around the world to work together. To support such use cases, apps need to move user data to the cloud and implement a data-sharing flow that includes features like sharing management, data synchronization, and access control.

This sample demonstrates how to use CloudKit to implement these features by allowing users to create topic and note records in their private databases and share them with other users. With the CloudKit sharing UI, users can send a share link, stop sharing topics, and manage permissions for a shared topic. Users who accept the share, called *participants*, can view or edit the shared record, or stop participating in the share.

The sample also demonstrates how to create an in-memory cache for a CloudKit record zone. Because of this local cache, the sample doesn't have to query the server while navigating the UI within the zone.

Configure the Sample Code Project

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

1. In the General pane of the `CloudKitShare` 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 CloudKit option 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, and specify the container identifier when creating the `container` variable in the `AppDelegate` class. An iCloud container identifier is case-sensitive and must begin with "`iCloud.`".

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

1. Log in with an Apple ID. For the CloudKit private database to synchronize across devices, the Apple ID must be the same on the devices.
2. Choose Settings > Apple ID > iCloud, and turn on iCloud Drive, if it is off.

Create a CloudKit Schema for the App

CloudKit apps need to create a schema to define the record types and fields, and CloudKit Dashboard is the tool for doing that. For more information, see Inspecting and Editing an iCloud Container's Schema.

The sample uses the following record types and fields:

```
Topic
    name (String)
Note
    title (String)
    topic (Reference, pointing to the parent topic)
```

In this instance, there is no need to manually create the schema before running the sample because:

- When an app saves a record in the development environment, CloudKit automatically creates the corresponding record type if it doesn't exist. For more information, see Designing and Creating a CloudKit Database.

cloudkit/designing-and-creating-a-cloudkit-database

- Before saving a record, the sample doesn't use any record-type information.

For real-world apps that use record types at an earlier phase, like creating a `CKQuerySubscription` at the beginning of a launch session, the schema must be ready first.

Create and Share a Topic

To create and share a topic with another iCloud user using the sample, follow these steps:

1. Prepare two devices, A and B, and log in to each device with a different iCloud account.
2. Use Xcode to build and run the sample app on the devices. If the system shows an alert that requests permission to use notifications, allow it.
3. On device A, tap the Zones button to show the zone view, then tap the Edit button and add a zone in the private database.
4. Tap the new zone to navigate to the topic view, then tap the Edit button and add a topic. Each topic has a Share button on the right.
5. Tap the Share button to show the CloudKit sharing UI, then follow the UI to send a link to the iCloud account for device B. Try to use Messages because it's easier to set up.
6. After receiving the link on device B, tap it to accept and open the share. The sample app launches, and then the shared topic and its zone appear in the shared database.

> **Note**
>
> To be able to accept a share when users tap a share link, the app's `info.plist` file must contain the `CKSharingSupported` key and its value must be `true`.

To discover more features in the CloudKit sharing UI:

- On device A, find the shared topic and tap the Share button. Because it's a shared topic, the sharing UI allows users to stop sharing or to change the permission for a participant.
- On device B, tap the Share button of the accepted topic. On the participant side, the sharing UI allows users to remove their participation from the topic.
- On device B, navigate to the shared database's topic view, then tap the Edit button and add a note under the shared topic. The new note synchronizes within seconds to the private database on device A. (This assumes the topic's "Anyone with this link can make changes" option is in an enabled state. If the topic's "Anyone with this link can view" option is in an enabled state, participants have read-only permissions, and can't add a note under the topic.)

- On device A, add a note under the shared topic. The note synchronizes within seconds to device B. When creating a note, the sample sets its `parent` property to the topic, so the system automatically shares the note with its parent topic.

```
newNoteRecord.parent = CKRecord.Reference(record: topicRecord, action: .none)
```

Share a Record

The sample uses `UICloudSharingController` to implement the sharing flow. Depending on whether the root record is in a shared state, there are different ways to create a sharing controller.

```
if rootRecord.share != nil {
    newSharingController(sharedRootRecord: rootRecord, database: database,
                         completionHandler: completionHandler)
} else {
    newSharingController(unsharedRootRecord: rootRecord, database: database, zone: z
                         completionHandler: completionHandler)
}
```

If the root record is in a shared state, the sample grabs the `recordID` from the `share` property of the root record, uses it to fetch the share, which is the associated `CKShare` object, from the server, and calls `init(share:container:)` to create a sharing controller.

```
let sharingController = UICloudSharingController(share: share, container: self)
```

If the root record isn't in a shared state, the sample uses `init(preparationHandler:)` to create the sharing controller.

```
let sharingController = UICloudSharingController { (_, prepareCompletionHandler) in
```

In the preparation handler, the sample sets up a new `CKShare` object using the root record.

```
let shareID = CKRecord.ID(recordName: UUID().uuidString, zoneID: zone.zoneID)
var share = CKShare(rootRecord: unsharedRootRecord, shareID: shareID)
share[CKShare.SystemFieldKey.title] = "A cool topic to share!" as CKRecordValue
share.publicPermission = .readWrite
```

The sample then saves the share and its root record together using `CKModifyRecordsOperation`.

```
let modifyRecordsOp = CKModifyRecordsOperation(recordsToSave: [share, unsharedRootRe
```

After creating the sharing controller, the sample uses the following code to present it:

```
sharingController.popoverPresentationController?.sourceView = sender as? UIView
self.rootRecord = topicRecord
sharingController.delegate = self
sharingController.availablePermissions = [.allowPublic, .allowReadOnly, .allowReadWr
self.present(sharingController, animated: true) { self.spinner.stopAnimating() }
```

Using the sharing UI, users can send a link, stop sharing the record, change the permission for a participant, or quit the flow by closing the UI. According to what users do, the sharing controller may change the root record and its share, and notify the app through the `UICloudSharingControllerDelegate` protocol. To ensure the cached data is consistent with the server truth, the sample implements the following delegate methods:

- `cloudSharingControllerDidSaveShare(_:)` — CloudKit calls this method when it successfully shares a topic. When this happens, it creates the share and updates the shared topic and notes on the server, so the sample fetches the changes and updates the local cache.
- `cloudSharingControllerDidStopSharing(_:)` — CloudKit calls this method when users stop sharing a record. When this happens, it removes the share and updates the shared topic and notes on the server, so the sample fetches the changes and updates the local cache.
- `cloudSharingController(_:failedToSaveShareWithError:)` — CloudKit calls this method when the sharing controller fails to save a share. When this happens, the sample alerts the error and updates the cached root record to avoid an inconsistent status.

Maintain a Local Cache of CloudKit Records

To avoid fetching data from the server each time the zone view and topic view are about to appear, the sample caches the zones in the container, and the topics and notes in the current zone. The caches are both in-memory because the sample doesn't tend to add more complexity by introducing a persistence layer. Real-world apps can persist their cache to avoid doing an initial fetch on each launch.

The sample establishes the local caches with two steps: initial fetch and incremental update. In `sceneWillEnterForeground(_:)`, the sample checks the account status, and then starts the initial fetch if there isn't a cache for the current account.

```
let building = appDelegate.buildZoneCacheIfNeed(for: newUserRecordID)
```

CloudKit notifications trigger the incremental updates. The sample uses `CKDatabaseSubscription` to subscribe to CloudKit database changes.

```
let subscription = CKDatabaseSubscription(subscriptionID: subscriptionID)
let notificationInfo = CKSubscription.NotificationInfo()
notificationInfo.shouldBadge = true
notificationInfo.alertBody = "Database (\(subscriptionID)) was changed!"
subscription.notificationInfo = notificationInfo

let operation = CKModifySubscriptionsOperation(subscriptionsToSave: [subscription],
operation.modifySubscriptionsCompletionBlock = { _, _, error in
    completionHandler(error as NSError?)
}

add(operation, to: operationQueue)
```

With the subscriptions, the sample gets push notifications when the database changes, and starts the incremental update from the following `UNUserNotificationCenterDelegate` method:

```
func userNotificationCenter(_ center: UNUserNotificationCenter, willPresent notifica
                            withCompletionHandler completionHandler: @escaping (UNNo
    print("\(#function)")
    updateWithNotificationUserInfo(notification.request.content.userInfo)
    completionHandler([])
}
```

The sample uses `CKFetchDatabaseChangesOperation` to fetch the deleted or changed zones. When doing the fetch, CloudKit provides a `serverChangeToken` (`CKServerChangeToken`) by calling the operation's `changeTokenUpdatedBlock`. Apps keep the token and use it as `previousServerChangeToken` for the next fetch.

```
operation.changeTokenUpdatedBlock = { serverChangeToken in
    self.setServerChangeToken(newToken: serverChangeToken, cloudKitDB: cloudKitDB)
}
```

When apps use the token to create and run a CloudKit operation, the token tells the server which portions of the zones to return. If the token is `nil`, the server returns all zones.

```
let serverChangeToken = getServerChangeToken(for: cloudKitDB)
let operation = CKFetchDatabaseChangesOperation(previousServerChangeToken: serverCha
```

After gathering the deleted and changed zones, the sample updates the zone cache and makes it consistent with the server truth.

Similarly, the sample uses `CKFetchRecordZoneChangesOperation` to gather the deleted and changed topic and notes, and uses them to maintain the topic cache.

```
let configuration = CKFetchRecordZoneChangesOperation.ZoneConfiguration()
configuration.previousServerChangeToken = getServerChangeToken()

let operation = CKFetchRecordZoneChangesOperation(
    recordZoneIDs: [zone.zoneID], configurationsByRecordZoneID: [zone.zoneID: config
)
```

To avoid blocking an app's main queue, CloudKit operations and their callbacks must run on a secondary queue, which can be an app-provided operation queue (`OperationQueue`), or a private operation queue that CloudKit manages. The sample provides an operation queue to run CloudKit operations and update the cached data when the operations complete. This means the system can access the cached data from different queues: the app's main queue that reads the data and updates the app UI, and a secondary queue that runs CloudKit operations and updates the data.

To be thread-safe, the sample serializes data access with a dispatch queue (`DispatchQueue`). One caveat of this solution is when the main queue needs to read the cached data while the secondary queue is updating it, the main queue must wait until the data update finishes. If the update takes a long time, it blocks the main queue for a long time, which leads to UI unresponsiveness. Real-world apps using the same method to serialize data access need to update the data quickly enough to avoid this issue.

See Also

Collaboration

{} 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.

`class CKShare`

A specialized record type that manages a collection of shared records.

`struct CKShareTransferRepresentation`

A transfer representation the system uses to share an item.

`class CKAllowedSharingOptions`

An object that controls participant access and permission options.

`class CKSystemSharingUIObserver`

An object the system uses to monitor changes in sharing.

`@MainActor class UICloudSharingController`

A view controller that presents standard screens for adding and removing people from a CloudKit share record.

`CKSharingSupported`

A Boolean value that indicates your app supports CloudKit Sharing.