Article

Syncing model data across a person's devices

Add the required capabilities and define a compatible schema to enable SwiftData to automatically sync your app's model data using iCloud.

Overview

People who use your app to create content expect that content to be available on all of their devices. SwiftData makes it possible to synchronize content by abstracting away the associated complexities. To adopt the framework's automatic sync functionality, add two Xcode capabilities to your app. The system operates with a set of predictable behaviors, such as using your app's `Entitlements.plist` file to infer the CloudKit configuration.

SwiftData uses the `NSPersistentCloudKitContainer` class from Core Data to handle CloudKit synchronization. For more information about how your models become instances of `CKRecord`, see Reading CloudKit Records for Core Data.

Add the iCloud and Background Modes capabilities

SwiftData requires two separate capabilities to perform automatic iCloud sync: the iCloud capability, which lets you configure CloudKit, and the Background Modes capability, which lets your app receive remote notifications from CloudKit that contain information about new changes on the server.

> **Important**
>
> The iCloud capability requires an active Apple Developer account with admin permissions.

To add the iCloud and Background Modes capabilities:

1. Follow the steps in Configuring iCloud services to add the iCloud capability to your Xcode project, enable CloudKit, and create or choose an existing *container* — an object that CloudKit uses to isolate your app's databases on the iCloud servers and manage their access and operations.

2. Follow the steps in Configuring background execution modes to add the Background Modes capability, enabling the Remote notifications option. The system delivers remote notifications silently to your app, allowing SwiftData to process the changes they describe and keep your local model data in sync with the iCloud servers.

Define a CloudKit compatible schema

A model layer described using macros in SwiftData will, in many cases, generate a schema already compatible with CloudKit. However, the SwiftData framework does include a small number of features that CloudKit doesn't support natively, such as unique constraints and nonoptional relationships. It's important you consider these limitations as you design your app's model layer (or adapt an existing one) to ensure it remains compatible with CloudKit.

SwiftData macro	CloudKit schema limitation
`@Attribute`	The framework synchronizes changes concurrently and at opportune times, which means CloudKit is unable to enforce the `unique` property option.
`@Relationship`	The iCloud servers don't guarantee atomic processing of relationship changes, so CloudKit requires all relationships to be optional. SwiftData automatically sets the inverse of a relationship if it can reliably infer that inverse from your schema. Otherwise, explicitly set the inverse before saving because CloudKit processes changes in an indeterminate order. The framework doesn't immediately synchronize changes, meaning CloudKit is unable to support the `Schema.Relationship.DeleteRule.deny` delete rule.

You manually initialize your app's CloudKit schema during development — as the following section describes — but you need to promote that schema to production before releasing your app. CloudKit schemas are additive only, which means you're unable to delete model types or change existing model attributes after you promote a schema to production.

Initialize the CloudKit development schema

After you define a model layer that's compatible with CloudKit, use the existing integration from Core Data with CloudKit to initialize a copy of that model layer on the iCloud servers. For example, you might do this during app launch by adding the necessary code to the `init()` function of the type in your app that adopts the App protocol from SwiftUI.

Follow these steps to ensure proper CloudKit schema initialization:

1. Create an instance of `ModelConfiguration`, which provides some basic information about the app's SwiftData stack.
2. Use the configuration's `url` property to create an instance of `NSPersistentStoreDescription`, enabling SwiftData and Core Data to reference the same store on disk.
3. Configure the store description with your app's CloudKit container identifier.
4. Request Core Data load the store synchronously, to guarantee that the load finishes before you attempt to initialize the CloudKit schema.
5. Create a managed object model that contains the same model types as the `ModelConfiguration` instance.
6. Use `NSPersistentCloudKitContainer` to load the store from the description and to initialize the CloudKit schema.
7. Unload the persistent store before creating an instance of `ModelContainer` to avoid both frameworks attempting to sync data to CloudKit.

```
let config = ModelConfiguration()

do {
#if DEBUG
    // Use an autorelease pool to make sure Swift deallocates the persistent
    // container before setting up the SwiftData stack.
    try autoreleasepool {
        let desc = NSPersistentStoreDescription(url: config.url)
        let opts = NSPersistentCloudKitContainerOptions(containerIdentifier: "iCloud
        desc.cloudKitContainerOptions = opts
        // Load the store synchronously so it completes before initializing the
        // CloudKit schema.
        desc.shouldAddStoreAsynchronously = false
        if let mom = NSManagedObjectModel.makeManagedObjectModel(for: [Trip.self, Ac
            let container = NSPersistentCloudKitContainer(name: "Trips", managedObje
            container.persistentStoreDescriptions = [desc]
            container.loadPersistentStores {_, err in
                if let err {
                    fatalError(err.localizedDescription)
```

```
                }
            }
            // Initialize the CloudKit schema after the store finishes loading.
            try container.initializeCloudKitSchema()
            // Remove and unload the store from the persistent container.
            if let store = container.persistentStoreCoordinator.persistentStores.fi
                try container.persistentStoreCoordinator.remove(store)
            }
        }
    }
#endif
    modelContainer = try ModelContainer(for: Trip.self, Accommodation.self,
                                        configurations: config)
} catch {
    fatalError(error.localizedDescription)
}
```

To ensure that schema initialization runs only in nonproduction builds, wrap your code with the `#if` compiler directive and specify the DEBUG compiler flag.

Go to the CloudKit Console to verify the initialized schema. If you're unable to see your schema's record types or data, you may need to enable querying support. For more information, see Inspecting and Editing an iCloud Container's Schema.

Configure SwiftData to use an existing CloudKit container

By default, SwiftData inspects your app's `Entitlements.plist` file to determine which CloudKit container to use, and selects the first identifier it finds in that file. If your app uses multiple CloudKit containers, you may need to configure SwiftData to use a specific identifier instead of relying on the default behavior.

> **Important**
>
> For apps already using a production CloudKit schema, specify only containers that SwiftData or Core Data have managed previously. All other CloudKit containers are incompatible.

To opt out of automatic container discovery in SwiftData, create an instance of `ModelConfiguration` and use the initializer's `cloudKitDatabase` parameter to specify your preferred identifier:

```
let config = ModelConfiguration(cloudKitDatabase: .private("iCloud.com.example.Trips
let modelContainer = try ModelContainer(for: Trip.self, Accommodation.self,
```

```
                                        configurations: config)
```

Disable automatic sync in apps already using CloudKit

SwiftData uses CloudKit to provide automatic iCloud sync and therefore requires the same Xcode-managed capabilities as those found in traditional CloudKit apps. This sharing of capabilities may lead to issues in apps already using CloudKit, because SwiftData assumes the presence of those capabilities as an indication that it handles sync. For example, automatic sync isn't possible if there are incompatibilities between a SwiftData schema and an existing CloudKit schema.

In such scenarios, opt out of automatic iCloud sync by creating an instance of `ModelConfiguration` and explicitly pass `none` for the `cloudKitDatabase` parameter:

```
let config = ModelConfiguration(cloudKitDatabase: .none)
let modelContainer = try ModelContainer(for: Trip.self, Accommodation.self,
                                        configurations: config)
```

Specifying `none` overrides any automatically discovered identifiers and disables SwiftData's automatic iCloud sync.

See Also

Model life cycle

`class ModelContainer`

An object that manages an app's schema and model storage configuration.

`class ModelContext`

An object that enables you to fetch, insert, and delete models, and save any changes to disk.

Fetching and filtering time-based model changes

Track all inserts, updates, and deletes that occur in a data store and process them as a series of chronological transactions.

`struct HistoryDescriptor`

A type that describes the criteria, and, optionally, sort order, to use when fetching history data

{} Deleting persistent data from your app

Explore different ways to use SwiftData to delete persistent data.

Reverting data changes using the undo manager

Automatically record data change operations that people perform in your SwiftUI app, and let them undo and redo those changes.

Concurrency support

Types you use to access model attributes and perform storage-related tasks in a safe and isolated way.