Sample Code

Linking Data Between Two Core Data Stores

Organize data in two different stores and implement a link between them.

Download

iOS 13.2+ | iPadOS 13.2+ | Mac Catalyst 13.2+ | Xcode 11.4+

Overview

Sometimes your app needs to work with multiple Core Data stores. For example, if you need to label data in a read-only store, you must write the labeling result to a different store. If you need to synchronize part of a large data set to iCloud, your app can organize the data in two stores to mirror one to CloudKit and keep the other on the local device.

This sample app shows how to manage and link data from two different stores with one Core Data stack. The app organizes the records of books and user feedback on the books into two separate stores. It connects two entities from the stores with a fetched property, which enables the app to access a book's feedback directly. When users add or remove a feedback record and save the change, the app updates the fetched property by refreshing the book object.

Configure the Data Model

The sample app creates a `Book` entity and a `Feedback` entity in the Core Data model, and then creates one `Configuration` for each store, named `Book` and `Feedback` as well, to contain the store's entity.

To link the entities, the sample app adds a fetched property, `feedbackList`, in the `Book` entity, which is displayed in the entity's Fetched Properties section in Xcode. The fetched property's target is set to the `Feedback` entity and its predicate is `bookUUID == $FETCH_SOURCE.uuid`, where `bookUUID` is a key path of the target and `$FETCH_SOURCE` will be replaced with the source of the fetched property, which is the Book object in this sample. With this setup, when the app refreshes a Book object and accesses its feedbackList property, Core Data executes an NSFetchRequest with the predicate on the Feedback entity, and returns the result to the property.

Set Up the Core Data Stack

The sample app creates one NSPersistentStoreDescription object for each store with the store's URL and model configuration, then uses NSPersistentContainer to load the stores.

```swift
let container = NSPersistentContainer(name: "CoreDataFetchedProperty")
let defaultDirectoryURL = NSPersistentContainer.defaultDirectoryURL()

let bookStoreURL = defaultDirectoryURL.appendingPathComponent("Books.sqlite")
let bookStoreDescription = NSPersistentStoreDescription(url: bookStoreURL)
bookStoreDescription.configuration = "Book"

let feedbackStoreURL = defaultDirectoryURL.appendingPathComponent("Feedback.sqlite"
let feedbackStoreDescription = NSPersistentStoreDescription(url: feedbackStoreURL)
feedbackStoreDescription.configuration = "Feedback"

container.persistentStoreDescriptions = [bookStoreDescription, feedbackStoreDescript
container.loadPersistentStores(completionHandler: { (_, error) in
    guard let error = error as NSError? else { return }
    fatalError("###\(#function): Failed to load persistent stores:\(error)")
})
```

With this setup, when the app fetches or changes data, Core Data automatically routes the request to the right store based on where the entities are.

Implement the Fetched Property

Xcode currently doesn't generate code for fetched properties, so the sample app adds the following extension to provide the accessor for feedbackList.

```swift
extension Book {
    var feedbackList: [Feedback]? { // The accessor of the feedbackList property.
        return value(forKey: "feedbackList") as? [Feedback]
    }
}
```

With the `feedbackList` accessor, the app can access the fetched property directly.

```
guard let feedback = book.feedbackList?[indexPath.row] else { return cell }
let rating = Int(feedback.rating)
let comment = feedback.comment ?? ""
```

Unlike a relationship, a fetched property can’t be used in a predicate for `NSFetchRequest`. It isn’t automatically updated when the managed context is saved either. When the sample app saves changes on the `Feedback` entity, the app must refresh the `book` object to update the `book.feedbackList` property.

```
context.refresh(book, mergeChanges: true)
```

See Also

Essentials

Creating a Core Data model

Define your app’s object structure with a data model file.

Setting up a Core Data stack

Set up the classes that manage and persist your app’s objects.

Core Data stack

Manage and persist your app’s model layer.

Handling Different Data Types in Core Data

Create, store, and present records for a variety of data types.