Article

Consuming relevant store changes

Filter store transactions for changes relevant to the current view.

Overview

Use persistent history tracking to determine what changes have occurred in the store, and to update your view context only as needed.

For example, consider an app that sometimes shows a list of colors, and sometimes shows a list of shapes. As the user views the `Color` objects from the view context, a background context may download additional `Color` data from a remote source. If the import happens through a batch operation, the save to the store doesn't generate an `NSManagedObjectContextDidSave` notification, and the view misses these relevant updates. Alternatively, the background context may save changes to the store that don't affect the current view—for example, inserting, modifying, or deleting `Shape` objects. These changes *do* generate context save events, so your view context processes them even though it doesn't need to.

Persistent history solves the problem by keeping track of every transaction on the store. You can filter this history for relevant changes and decide how or whether to update a view.

Enable History Tracking for Your Local Store

When creating a persistent container in your app's delegate, set the `NSPersistentHistoryTrackingKey` option on the store description to `true`.

```
class AppDelegate: UIResponder, UIApplicationDelegate {

    // ...

    lazy var persistentContainer: PersistentContainer = {
        let container = PersistentContainer(name: "PersistentHistoryTracking")

        // turn on persistent history tracking
        let description = container.persistentStoreDescriptions.first
        description?.setOption(true as NSNumber,
                               forKey: NSPersistentHistoryTrackingKey)

        // ...

        return container
    }()

    // ...
}
```

Core Data now tracks all changes to your local store.

Listen for Remote Changes

In the persistent container in your app's delegate, toggle the store description option for enabling remote change notifications to `true`.

```
class AppDelegate: UIResponder, UIApplicationDelegate {

    // ...

    lazy var persistentContainer: PersistentContainer = {
        let container = PersistentContainer(name: "PersistentHistoryTracking")

        // ...

        // turn on remote change notifications
        let remoteChangeKey = "NSPersistentStoreRemoteChangeNotificationOptionKey"
        description?.setOption(true as NSNumber,
                                   forKey: remoteChangeKey)

        // ...

        return container
    }()

    // ...
}
```

In your view controller, add an observer to listen for remote change notifications.

```
NotificationCenter.default.addObserver(
    self,
    selector: #selector(fetchChanges),
    name: NSNotification.Name(
        rawValue: "NSPersistentStoreRemoteChangeNotification"),
        object: persistentContainer.persistentStoreCoordinator
)
```

Provide Details About a Transaction’s Source

Each history transaction automatically includes the originating `storeID`, `bundleID` and `process ID`. You can supply additional information about the source of a change by setting each managed object context’s `name` and `transactionAuthor`.

Provide a unique `name` for each context to identify it in the persistent history. The context’s `name` becomes the persistent history transaction’s `contextName`. You only need to set this once per context.

```
class PersistentContainer: NSPersistentContainer {

    override init(name: String, managedObjectModel model: NSManagedObjectModel) {
        super.init(name: name, managedObjectModel: model)

        // set the context name
        viewContext.name = "viewContext"
    }
}
```

You can also set a `transactionAuthor` before each context save to differentiate among multiple call sites that modify the same context. The context’s `transactionAuthor` becomes the `author` of subsequent persistent history transactions.

```
func addColor(_ name: String, in context: NSManagedObjectContext) {
    let color = Color(context: context)
    color.name = name
    color.creationDate = Date()

    // set the transaction author
    context.transactionAuthor = "addColor"
    persistentContainer.saveContext(context)
    context.transactionAuthor = nil
}
```

Reset the context's `transactionAuthor` to `nil` after saving the context to prevent misattribution of future transactions.

Keep Track of Your Place in the History

Create an instance of `NSPersistentHistoryToken` to keep track of the most recent history that you have processed.

```
var lastToken: NSPersistentHistoryToken?
```

You can encode the token to disk so that when your app exits, you can keep track of where you were in the history. When you relaunch your app, fetch history based on your token.

```
var lastToken: NSPersistentHistoryToken? = nil {
    didSet {
        guard let token = lastToken,
            let data = try? NSKeyedArchiver.archivedData(
                withRootObject: token,
                requiringSecureCoding: true
            ) else { return }
        do {
            try data.write(to: tokenFile)
        } catch {
            let message = "Could not write token data"
            print("###\(#function): \(message): \(error)")
        }
    }
}

lazy var tokenFile: URL = {
    let url = NSPersistentContainer.defaultDirectoryURL().appendingPathComponent(
        "YourProjectName",
        isDirectory: true
    )
    if !FileManager.default.fileExists(atPath: url.path) {
        do {
            try FileManager.default.createDirectory(
                at: url,
                withIntermediateDirectories: true,
                attributes: nil
            )
        } catch {
            let message = "Could not create persistent container URL"
            print("###\(#function): \(message): \(error)")
        }
    }
    return url.appendingPathComponent("token.data", isDirectory: false)
}()
```

Request History

To request history, use the `fetchHistory(after:)` type method on `NSPersistentHistoryChangeRequest`. This example shows a request to fetch the history that is new since you last fetched history. Execute the fetch request on a background context to avoid blocking the main thread. Convert the `NSPersistentHistoryResult` to an array of `NSPersistentHistoryTransaction`.

```
let fetchHistoryRequest = NSPersistentHistoryChangeRequest.fetchHistory(
    after: lastToken
)

let context = persistentContainer.backgroundContext
guard
    let historyResult = try? context.execute(fetchHistoryRequest)
        as? NSPersistentHistoryResult,
    let history = historyResult!.result as? [NSPersistentHistoryTransaction]
    else {
        fatalError("Could not convert history result to transactions.")
}
```

Alternatively you can use `fetchHistory(after:)` to get history after a particular date, or after a particular a transaction.

Read History Transactions

Each transaction represents a set of changes. Iterate through the array of transactions to learn their details. The following code loops through the results of the `fetchHistoryRequest` to inspect the properties of each transaction.

```
for transaction in history.reversed() {

    // token, date and transaction number
    let token = transaction.token
    let timestamp = transaction.timestamp
    let transactionNumber = transaction.transactionNumber

    // transaction source details
    let store = transaction.storeID
    let bundle = transaction.bundleID
    let process = transaction.processID
    let context = transaction.contextName ?? "unknown context"
    let author = transaction.author ?? "unknown author"

    // the list of changes
    guard let changes = transaction.changes else { continue }
}
```

A transaction's `changes` array includes information about multiple changes. A single `NSPersistentHistoryChange` represents the insertion, update, or deletion of an object.

Iterate through a transaction's changes to identify each object that changed, the type of change that occurred, and any details about the change.

In the case of an update, the `updatedProperties` set includes any updated attributes and relationships. In the case of a deletion, the `tombstone` dictionary includes key-value pairs for any attributes marked for preservation after deletion.

```
    for change in changes {

        let objectID = change.changedObjectID
        let changeID = change.changeID
        let transaction = change.transaction
        let changeType = change.changeType

        switch(changeType) {
        case .update:
            guard let updatedProperties = change.updatedProperties else { break }
            for updatedProperty in updatedProperties {
                let name = updatedProperty.name
            }
        case .delete:
            if let tombstone = change.tombstone {
                let name = tombstone["name"]
            }
        default:
            break
        }
    }
```

Filter for Transactions Relevant to the View

Filter the history to narrow it to changes affecting the current view. The following code filters for changes to `Color` instances, updating the last transaction token as it goes.

```
var filteredTransactions = [NSPersistentHistoryTransaction]()
for transaction in transactions {
    let filteredChanges = transaction.changes!.filter { change -> Bool in
        return Color.entity().name == change.changedObjectID.entity.name
    }
    if !filteredChanges.isEmpty {
        filteredTransactions.append(transaction)
    }
    self.lastToken = transaction.token
}
if filteredTransactions.isEmpty { return }
```

Relevant changes may include all changes to a given entity, or more selectively, only changes to those properties that are visible on the screen.

Merge Relevant Transactions into a Context

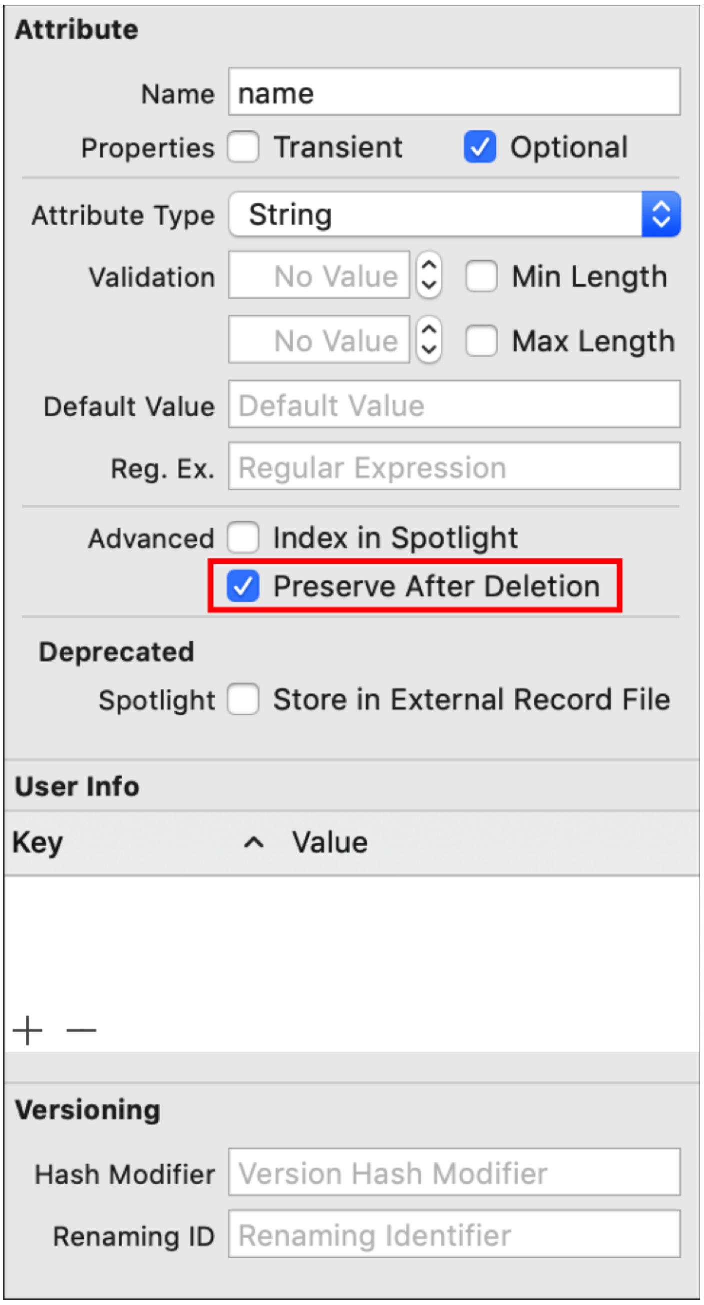
To merge the relevant changes into your view context, first obtain a notification by calling `objectIDNotification()` on the transaction. Then, pass the notification to `mergeChanges(fromContextDidSave:)`.

```
for transaction in filteredTransactions {
    self.fetchedResultsController.managedObjectContext.perform {
        self.fetchedResultsController.managedObjectContext.mergeChanges(
            fromContextDidSave: transaction.objectIDNotification()
        )
    }
}
```

Access Attributes of a Deleted Object

After you delete an object from the store, its `objectID` is no longer relevant. Identify a deleted object by recording select properties in its tombstone.

In the Core Data model editor, select an attribute. In the data model editor, select the Preserve After Deletion checkbox.



Attribute

Name	name
Properties	☐ Transient ☑ Optional
Attribute Type	String
Validation	No Value ☐ Min Length
	No Value ☐ Max Length
Default Value	Default Value
Reg. Ex.	Regular Expression
Advanced	☐ Index in Spotlight
	☑ Preserve After Deletion

Deprecated

Spotlight	☐ Store in External Record File

User Info

Key	Value

+ −

Versioning

Hash Modifier	Version Hash Modifier
Renaming ID	Renaming Identifier

In the persistent history, `NSPersistentHistoryChangeType.delete` changes include a `tombstone` dictionary with key-value pairs for any attributes marked for preservation after deletion.

```
for transaction in history.reversed() {
    guard let changes = transaction.changes else { continue }
    for change in changes where change.changeType == .delete {
        if let tombstone = change.tombstone {
            let name = tombstone["name"]
        }
    }
}
```

Purge History

Because persistent history tracking transactions take up space on disk, determine a clean-up strategy to remove them when they are no longer needed. Before pruning history, a single gatekeeper should ensure that your app and its clients have consumed the history they need.

Similar to fetching history, you can use `deleteHistory(before:)` to delete history older than a token, a transaction, or a date. For example, you can delete all transactions older than seven days.

```
let sevenDaysAgo = Date(timeIntervalSinceNow: TimeInterval(exactly: -604_800)!)
let purgeHistoryRequest =
    NSPersistentHistoryChangeRequest.deleteHistory(
        before: sevenDaysAgo)

do {
    try persistentContainer.backgroundContext.execute(purgeHistoryRequest)
} catch {
    fatalError("Could not purge history: \(error)")
}
```

If you attempt to fetch purged history, Core Data throws an expired token error.

See Also

Change processing

Accessing data when the store changes

Guarantee that a context won’t see store changes until you tell it to look.

Persistent history

Use persistent history tracking to determine what changes have occurred in the store since the enabling of persistent history tracking.