Article

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.

Overview

As people interact with your app, the app's persisted model data may change over time. For example, a person (or process) may create, update, or delete model instances. When the app fetches data from the data store, those results represent only the current state of that model data; there's no straightforward way to determine if there's been any changes to that data since the last fetch.

SwiftData History enables your app to track changes in its data store over time. The data store organizes changes as a series of chronological transactions, where each transaction contains information about one or more persisted changes. Your app can fetch these transactions and react accordingly. For example, your app may need to determine changes made by another process such as a Widget or App Intent and reflect those changes in its user interface.

To use SwiftData History in your app, create a history descriptor and use a model context to fetch the corresponding chronological transactions. After the fetch completes, determine which of those fetched transactions relate to the current view or task and process them accordingly. After you finish processing them, delete the transactions from the store to reclaim the disk space.

> **Note**
>
> SwiftData History is available in data stores that adopt the `HistoryProviding` protocol such as `DefaultStore` . All examples in this article assume the use of `DefaultStore` and its related types.

Fetch a store's change transactions

Transactions group together one or more changes that occur on a specific boundary — such as when a model context writes pending changes to the store — and are identifiable by their associated history token. SwiftData stores transactions in the order they occur, and a model context fetches them in that same order. The group of changes that a transaction contains (inserts, updates, deletes) are also ordered chronologically.

Using a model context, you can fetch all transactions from the persistent store, or just a subset by specifying a history token, an author, or both. Tokens are opaque objects that conform to the `Comparable` and `Codable` protocols, enabling you to store the most recent token on-disk and use it in the next fetch to receive only newer changes. An author is a short string that your app uses to identify the origin of a transaction, which you specify on the model context that writes those changes to the store.

For example, you may want to fetch all new transactions that originate from your app's widget.

```
func fetchWidgetTransactions(after tokenData: Data) -> Result<[DefaultHistoryTransac
    do {
        // Decode the given token data.
        let token = try JSONDecoder().decode(History.DefaultToken.self, from: tokenD
        // Create a history descriptor and specify the predicate.
        var descriptor = History.HistoryDescriptor<History.DefaultTransaction>()
        descriptor.predicate = #Predicate {
            ($0.token > token) && ($0.author == "widget")
        }
        // Fetch the matching history transactions.
        let context = ModelContext(modelContainer)
        let txns = try context.fetchHistory(descriptor)
        return .success(txns)
    } catch {
        return .failure(error)
    }
}
```

After processing the fetched transactions, make sure that you write the most recent transaction's token to disk so you can use it in the next fetch.

Identify relevant model changes

As transactions represent points in time, they're heterogenous — a single transaction can (and often will) contain changes for several different model types. Because of this, transactions aren't

bound to a specific model type. When you fetch them from a data store, the results will likely contain transactions, and changes within those transactions, that are unrelated to the current view or task. Filter each transaction's changes and identify only those that are relevant.

The following example shows how you might identify trips with updated flight times:

```
let context = ModelContext(modelContainer)
var updatedTrips = Set<Trip>()

for txn in transactions {
    // Filter out any change that isn't an update.
    for change in txn.changes where change is History.DefaultUpdateChange<Trip> {
        // Proceed only when there's a single change, and that change
        // is to the `flightTime` attribute.
        guard change.updatedAttributes.count == 1,
              change.updatedAttributes.contains(\.flightTime)
        else { continue }
        // Use the model ID from the change to fetch the actual model.
        let changedModelID = change.changedModelID
        var fetchDescriptor = FetchDescriptor<Trip>(predicate: #Predicate {
            $0.persistentModelID == changedModelID
        })
        if let trip = try? taskContext.fetch(fetchDescriptor).first {
            updatedTrips.insert(trip)
        }
    }
}
```

Preserve important attributes of deleted models

After deleting a model from the data store, its values are gone with no way to recover them. In most situations, this is the preferred behavior. However, there may be occasions where your app needs to retain one or more attribute values from a deleted model. For example, using a model's `persistentModelID` as a means of identifying that model is only relevant within the scope of the local data store, and a different attribute may provide a stable identity across different devices and services. By retaining that attribute's value, you're able to reliably identify the deleted model after it's gone.

To retain a value, use the `Attribute(_:originalName:hashModifier:)` macro and specify the `preserveValueOnDeletion` option:

```
@Model
final class Trip {
    @Attribute(.preserveValueOnDeletion)
    var airlineBookingRef: String
    // ...
}
```

Then, when processing a transaction's changes, use the `tombstone` property to retrieve the preserved value. `History/Tombstone` is a generic sequence type that lets you iterate over the preserved values, or access a specific value directly using the corresponding model key path.

```
if let deletion = change as? History.DefaultDeleteChange<Trip> {
   bookingRef = deletion.tombstone[\.airlineBookingRef]
}
```

Delete stale change transactions

SwiftData writes transactions to the data store alongside the model data, and as such, transactions require additional disk space. To make sure the store doesn't consume more space than necessary, determine a suitable clean-up strategy to remove stale transactions when your app no longer needs them.

> **Important**
>
> If you attempt to fetch deleted transactions, SwiftData throws a `historyTokenExpired` error.

Similar to fetching, use a model context to delete transactions and provide a predicate to narrow the scope. For example, you may want to delete all transactions that occur before a given token:

```
func deleteTransactions(before token: History.DefaultToken) -> Result<Void, Error> {
    do {
        // Create a history descriptor and specify the predicate.
        var descriptor = History.HistoryDescriptor<History.DefaultTransaction>()
        descriptor.predicate = #Predicate {
            $0.token < token
        }
        // Delete the matching history transactions.
        let context = ModelContext(modelContainer)
```

```
            try context.deleteHistory(descriptor)
            return .success
        } catch {
            return .failure(error)
        }
    }
```

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.

`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.

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.

Concurrency support

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