Sample Code

Loading and displaying a large data feed

Consume data in the background, and lower memory use by batching imports and preventing duplicate records.

Download

iOS 15.0+ | iPadOS 15.0+ | macOS 12.0+ | Xcode 13.0+

Overview

This sample creates an app that shows a list of earthquakes recorded in the United States in the past 30 days by consuming a U. S. Geological Survey (USGS) real-time data feed.

To load the USGS JSON feed, perform either of the following:

- On iOS, pull to refresh the `List`.
- On both iOS and macOS, press the refresh button (⌘R).

The app will load the requested data on the default delegate queue of `URLSession`, which is an operation queue that runs in the background. After the feed is downloaded and the session data task completes, the app continues working on this queue to import the large number of feed elements to the store without blocking the main queue.

> Note
>
> This sample code project is associated with WWDC21 session 10017: Bring Core Data Concurrency to Swift and SwiftUI.

Import data in the background

To import data in the background, apps may use one or two managed object contexts. The sample uses two (`NSManagedObjectContext`) instances:

- A main queue context to provide data to the user interface.
- A private queue context to perform the import on a background queue.

Both contexts are connected to the same `persistentStoreCoordinator`. This configuration is more efficient than using a nested context.

The sample creates a main queue context by setting up a Core Data stack using `NSPersistentContainer`, which initializes a main queue context in its `viewContext` property.

```
let container = NSPersistentContainer(name: "Earthquakes")
```

Create a private queue context by calling the persistent container's `newBackgroundContext()` method.

```
let taskContext = container.newBackgroundContext()
```

When the feed download finishes, the sample uses the task context to consume the feed in the background. In Core Data, every queue-based context has its own serial queue, and apps must serialize the tasks that manipulate the context with the queue by wrapping the code with a `perform(_:)` — with or without the `await` keyword — or `performAndWait(_:)` closure.

```
try await taskContext.perform {
```

For more information about working with concurrency, see `NSManagedObjectContext`.

To efficiently handle large data sets, the sample uses `NSBatchInsertRequest` which accesses the store directly — without interacting with the context, triggering any key value observation, or allocating managed objects. The closure-style initializer of `NSBatchInsertRequest` allows apps to provide one record at a time when Core Data calls the `dictionaryHandler` closure, which helps apps keep their memory footprint low because they do not need to prepare a buffer for all records.

```
let batchInsertRequest = self.newBatchInsertRequest(with: propertiesList)
if let fetchResult = try? taskContext.execute(batchInsertRequest),
   let batchInsertResult = fetchResult as? NSBatchInsertResult,
   let success = batchInsertResult.result as? Bool, success {
    return
}
```

Merge changes and update the user interface

Because `NSBatchInsertRequest` bypasses the context and doesn't trigger a `NSManagedObjectContextDidSaveNotification` notification, apps that need to update the UI with the changes have two options:

- Extract the relevant changes by parsing the store's Persistent history, then merge them into the view context. For more information on persistent history tracking, see Consuming relevant store changes.
- Re-fetch the data from the store. However, if the view context is pinned to a query generation, the context will need to be reset before fetching data. For more information on query generations, see Accessing data when the store changes.

This sample uses persistent store remote change notifications and persistent history tracking to update the UI, because:

- The data model contains a single entity, so all changes are relevant to the `List` and do not require parsing specific changes within the history.
- `FetchRequest` fetches and retrieves results directly from the store, and the `List` refreshes its contents automatically.
- SwiftUI is only concerned about the view context, so `QuakesProvider` observes the `NSPersistentStoreRemoteChange` notification to merge changes from the background context, performing the batch operations, into the view context.

Enable remote change notifications for a persistent store by setting the `NSPersistentStoreRemoteChangeNotificationPostOptionKey` option on the store description to `true`.

```
description.setOption(true as NSNumber,
                      forKey: NSPersistentStoreRemoteChangeNotificationPostOptionKey
```

Enable persistent history tracking for a persistent store by setting the `NSPersistentHistoryTrackingKey` option to `true` as well.

```
description.setOption(true as NSNumber,
                      forKey: NSPersistentHistoryTrackingKey)
```

Whenever changes occur within a persistent store, including writes by other processes, the store posts a remote change notification. When the sample receives the notification, it fetches the persistent history transactions and changes occurring after a given token. After the persistent history change request retrieves the history, the sample merges each transaction's `objectIDNotification()` into the view context via `mergeChanges(fromContextDidSave:)`.

```
let changeRequest = NSPersistentHistoryChangeRequest.fetchHistory(after: lastToken)
let historyResult = try taskContext.execute(changeRequest) as? NSPersistentHistoryRe
if let history = historyResult?.result as? [NSPersistentHistoryTransaction] {
    return history
}
```

After executing each `NSBatchInsertRequest` or `NSBatchDeleteRequest`, the sample dispatches any UI updates back to the main queue, to render them in SwiftUI.

```
let viewContext = container.viewContext
let tokens = await viewContext.perform {
    history.map { (transaction: NSPersistentHistoryTransaction) -> NSPersistentHisto
        viewContext.mergeChanges(fromContextDidSave: transaction.objectIDNotificatio
        return transaction.token
    }
}
```

After merging changes from the last transaction, the sample needs to store the token in memory or on disk, to use it in subsequent persistent history change requests.

Work in batches to lower memory footprint

When apps fetch or create objects in a context, Core Data caches the object to avoid a round trip to the store file when the app uses those objects again. However, that approach grows the memory footprint of an app as it processes more and more objects, and can eventually lead to low-memory warnings or app termination on iOS. `NSBatchInsertRequest` doesn't obviously increase an app's memory footprint because it doesn't load data into memory.

> **Note**
>
> Apps targeted to run on a system earlier than iOS 13 or macOS 10.15 need to avoid memory footprint growing by processing the objects in batches and calling `reset()` to reset the context after each batch.

The sample sets the `viewContext`'s `automaticallyMergesChangesFromParent` property to `false` to prevent Core Data from automatically merging changes every time the background context is saved.

```
container.viewContext.automaticallyMergesChangesFromParent = false
```

Prevent duplicate data in the store

Every time the sample app reloads the JSON feed, the parsed data contains all earthquake records for the past month, so it can have many duplicates of already imported data. To avoid creating duplicate records, the app constrains an attribute, or combination of attributes, to be unique across all instances.

The `code` attribute uniquely identifies an earthquake record, so constraining the `Quake` entity on `code` ensures that no two stored records have the same `code` value.

Select the `Quake` entity in the data model editor. In the data model inspector, add a new constraint by clicking the + button under the Constraints list. A constraint placeholder appears.

```
comma, separated, properties
```

Double-click the placeholder to edit it. Enter the name of the attribute, or comma-separated list of attributes, to serve as unique constraints on the entity.

```
code
```

When saving a new record, the store now checks whether any record already exists with the same value for the constrained attribute. In the case of a conflict, an `NSMergeByPropertyObjectTrumpMergePolicy` policy comes into play, and the new record overwrites all fields in the existing record.

```
container.viewContext.automaticallyMergesChangesFromParent = false
```

See Also

Accessing Core Data

`var managedObjectContext: NSManagedObjectContext`

`struct FetchRequest`

A property wrapper type that retrieves entities from a Core Data persistent store.

`struct FetchedResults`

A collection of results retrieved from a Core Data store.

`struct SectionedFetchRequest`

A property wrapper type that retrieves entities, grouped into sections, from a Core Data persistent store.

`struct` `SectionedFetchResults`

A collection of results retrieved from a Core Data persistent store, grouped into sections.