11.Core Data

PRO

Second Edition

Swift 5.1, iOS 13, Xcode 11

Before You Begin

Concurrency

Dispatch

SECTION 0: 3 CHAPTERS

SECTION 1: 2 CHAPTERS

SECTION 2: 3 CHAPTERS

SECTION 3: 5 CHAPTERS

Section IV: Real-Life

SECTION 4: 3 CHAPTERS

11.1 NSManagedObjectContext is not

11.3 NSAsynchronousFetchRequest

11.4 Sharing an NSManagedObject

11.5 Where to go from here?

12. Thread Sanitizer

12.1 Why the sanitizer?

12.3 Enabling sanitization

12.4 It's not code analysis

12.6 Where to go from here?

12.5 Xcode keeps getting smarter

12.2 Getting started

13. Conclusion

Concurrency

> 11. Core Data

thread safe

11.2 Importing data

Section III: Operations

Section II: Grand Central

Concurrency by Tutorials

Section I: Getting Started with

11 Core Data
Written by Scott Grosch

Home > iOS & Swift Books > Concurrency by Tutorials

Many of the iOS apps you'll develop are likely to use Core Data for data storage. While mixing concurrency and Core Data is no longer complicated in modern versions of iOS, there are still a couple of key concepts that you'll want to be aware of. Just like most of UIKit, Core Data is not thread safe.

Note: This chapter assumes that you already know how to perform basic Core Data concepts such as searching, creating entities, etc... Please check out our book, Core Data by Tutorials at https://bit.ly/2VkUKNb if you are new to Core Data.

The NSManagedObjectContext, which gets created as part of an

NSManagedObjectContext is not thread safe

NSPersistentContainer from your AppDelegate, is tied to the main thread. As

you've learned throughout the rest of this book, this means you can only use that context on the main UI thread. However, if you do so, you're going to negatively impact your user's experience. There are two methods available on the NSManagedObjectContext class to help with concurrency:

perform(_:) performAndWait(_:)

main thread." You might create your own context on another queue. These methods

thus ensure that you are always executing against the proper thread and don't crash

at runtime.

What both methods do is ensure that whatever action you pass to the closure is executed on the same queue that created the context. Notice how it' not "on the

The only difference between the two is that the first is an asynchronous method, whereas the second is synchronous. It should be abnormal for you to perform Core Data tasks without utilizing either of these methods. Even though you might know you're on the proper thread already, you might refactor your code in the future and

forget to wrap the Core Data calls at that point in time. Save yourself the headache and use them right from the start! Importing data When your app starts, one of the first goals it frequently has is to contact the server

and download any new data. Once the network operation completes, an expensive

compare and import cycle will have to take place. However, you don't need to create an entire Operation for this common task. Core Data's NSPersistentContainer provides performBackgroundTask(_:) which will help you out:

persistentContainer.performBackgroundTask { context in COPY for json in jsonDataFromServer { let obj = MyEntity(context: context) obj.populate(from: json)

```
do {
    try context.save()
  } catch {
    fatalError("Failed to save context")
Notice how the closure argument is an NSManagedObjectContext. Core Data will
generate a new context on a private queue for you to work with so that you don't
have to worry about any concurrency issues. Be sure not to use any other context in
```

If you've worked with Core Data and concurrency in the past, you'll see that performBackgroundTask is a convenience method for this previously required song and dance:

childContext.perform { No lying, now: How many times have you written the preceding code and forgotten

COPY -

NSAsynchronousFetchRequest When you use an NSFetchRequest to query Core Data, the operation is

synchronous. If you're just grabbing a single object, that's perfectly acceptable.

When you're performing a time-consuming query, such as retrieving data to populate a UITableView, then you'll prefer to perform the query asynchronously.

the closure or you'll run into tough to debug runtime issues.

let childContext = NSManagedObjectContext(concurrencyType:

childContext.parent = persistentContainer.viewContext

.privateQueueConcurrencyType)

to set the parent context properly?

like so:

don't do it!

argument.

again.

To this:

Using NSAsynchronousFetchRequest is the obvious solution. Construct the fetch request as you normally would but then pass that fetch request as the first parameter to the constructor of NSAsynchronousFetchRequest. The second parameter is a closure to be executed when the fetch completes. The closure takes a single argument of type NSAsynchronousFetchResult.

Note: An asynchronous fetch request must be run in a private background queue. Running your asynchronous fetch request in a private background queue is easily

let ageKeyPath = #keyPath(Person.age) COPY let fetchRequest = Person.fetchRequest() as NSFetchRequest<Person> fetchRequest.predicate = NSPredicate(format: "%K > 13", ageKeyPath)

let asyncFetch = NSAsynchronousFetchRequest(fetchRequest: fetchRequest) {

accomplished via the newBackgroundContext method of your persistent container,

```
[weak self] result in
 guard let self = self,
       let people = result.finalResult else {
   return
 self.tableData = people
 DispatchQueue.main.async {
         self.tableView.reloadData()
do {
 let backgroundContext = persistentContainer.newBackgroundContext()
 try backgroundContext.execute(asyncFetch)
} catch let error {
  // handle error
Sharing an NSManagedObject
You can't share an NSManagedObject — or a subclass — between threads. Sure, you
can, and more often than not it will seem to work, but your app is going to break, so
```

That doesn't mean you can't effectively use Core Data if your app is written to fully utilize multiple threads. If two separate threads both need access to the same object, you must pass the NSManagedObjectId instead of the actual NSManagedObject. You

can get the NSManagedObjectId via the objectID property. A common situation wherein you'd need to use the same entity across multiple threads is when you post a notification after creating or updating the entity.

It's incredibly easy to get the entity you want based on an ID: let objectId = someEntity.objectID COPY DispatchQueue.main.async { [weak self] in guard let self = self else { return } let myEntity = self.managedObjectContext.object(with: objectId)

calling Core Data methods on the wrong thread in the debugger. While adding the debugging flag provides an extra level of safety, it also comes at a large performance cost. You'll find that leaving the flag enabled is usually a bad idea as it makes your

self.addressLabel.text = myEntity.address

Using Concurrency Debug

app look less performant, which is the opposite of what you're trying to do! However, you should periodically perform your full suite of tests with the flag enabled to catch any mistakes.

To help protect yourself from sharing an NSManagedObject across threads, you can

enable a runtime debug flag by editing the project' scheme and passing a runtime

Add -com.apple.CoreData.ConcurrencyDebug 1 to your app's scheme to catch

In this chapter project's download materials, open the **Concurrency.xcodeproj** project. In MainViewController.swift you'll see there are two aptly named methods defined, both match the callback signature for an NSNotification: doItTheRightWay(note:) doItTheWrongWay(note:) The project is currently set to pass the actual NSManagedObject to the notification,

Data entities were generated. What' going on here, though? I just told you the app is written incorrectly, yet everything works just fine! Edit the scheme of your app now and put a checkmark next to the concurrency debug argument so that it' enabled and then run the app

This time the progress bar will run to completion as normal but, instead of getting a

caused Xcode to notice that you crossed threads with an NSManagedObject and so it

text message telling you how many entities were created, you'll notice that you

crashed in Xcode. Using the extra debugging provided by the runtime argument

which, of course, you just learned is really bad! Build and run the app. You can tap

on the **Generate** button a bunch of times, and you'll likely be told how many Core

aborted with an EXC_BAD_INSTRUCTION exception. Running in debug mode catches your bad memory usage. Head back to MainViewController and change this line at the top of the class: private let passActualObject = true COPY

private let passActualObject = false COPY

Rebuild and rerun your app. This time you won't crash. You'll see the difference

the entity properly based on the objectID and not taking an actual object.

corruption if you try to share an NSManagedObject across threads.

between the two methods is that the doItTheRightWay(note:) method is looking up

Hopefully this small example shows you how important it is to run with the extra debugging enabled long before you ship your app to production. Just because the app normally works doesn't mean that it always will if you cross thread boundaries. On a real device, performing normal daily tasks, you'll eventually hit memory

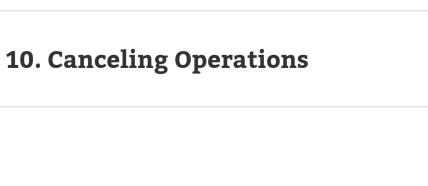
Where to go from here?

Core Data in a thread safe way. If you previously shied away from this great framework because it was hard to use concurrently, now is the time to take another look! If you'd like to know more about the intricacies of Core Data, please check out our book, Core Data by Tutorials at https://bit.ly/2VkUKNb.

Hopefully, the preceding few pages have shown you that it's pretty easy to work with

Mark Complete

12. Thread Sanitizer



Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.

Feedback about the UI, UX, or other features of the online reader? Leave them

Have feedback to share about the online reading experience? If you have feedback about

the UI, UX, highlighting, or other features of our online readers, you can send them to the

Send Feedback

© 2021 Razeware LLC

here!

design team with the form below: