Extracted from:

Core Data in Objective-C, Third Edition

Data Storage and Management for iOS and OS X

The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

Third Edition

Core Data in Objective-C

Data Storage and Management for iOS and OS X

Marcus S. Zarra

Edited by Jacquelyn Carter

Core Data in Objective-C, Third Edition

Data Storage and Management for iOS and OS X

Marcus S. Zarra

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

Printed in the United States of America.
ISBN-13: 978-1-68050-123-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2016

Multithreading is one of the great double-edged swords of programming. If it's done correctly, it can be a real boon to your application; done incorrectly, it leads to strange, unreproducible errors in the application. Multithreading has a tendency to polarize developers: they either swear that it's necessary for any application to perform properly or declare it's to be avoided at all costs. The truth, of course, is somewhere in the middle. Multithreading is a vital piece of the overall performance puzzle. While adding more threads won't make your application automatically faster, it can make it "feel" faster to the user. That perception is what we're going to focus on in this chapter.

It's a common misconception among developers that the point of adding threads to an application is to improve performance. While there's no argument that proper threading support can improve performance in an application, treating threading like a silver bullet is a sure way to disaster.

Threading should be introduced to an application as part of the design process, whenever there's a situation where the application can or should be doing more than one thing. Any situation during the application design where an operation is needed but the user doesn't need to wait on that operation is a perfect situation for an additional thread.

Here are some common operations that fall into this category:

- Exporting data to a web service
- Importing data from a web service
- Recalculating data (totals and balances)
- Caching images
- Caching videos

And the list goes on. In addition to these concepts there's the concept of anticipating the user and what data the user is going to want next. If, for example, you're developing a news application, it makes sense to load the full news articles, images, videos, and so forth while the user is still scrolling through the list of articles. Ideally your application will have the data loaded and ready to display before the user selects it.

When an application can correctly predict what data a user is going to want to see before the user requests it, that application reaches a whole new level in user experience.

The purpose of adding threads to your application is to improve user experience by offloading work the user doesn't need to be blocked by, as well as predictively load data before the user needs it.

With that goal for threading in mind, let's look at how to use Core Data in a multithreaded environment.

Threading and Core Data

Throughout the life of the Core Data framework, the approach to using Core Data with threading has changed many times. Originally there was no support for threading other than "figure it out," which evolved into the basic rule of "A context and its data must stay on one thread."

With the introduction of iOS 5.0 and OS X 10.8, Core Data began utilizing GCD (Grand Central Dispatch) and blocks that were introduced to the overall system in the previous generation. To further define how threading should be approached with Core Data, the threading model was again refined in iOS 8.0 and OS X 10.10, and yet again in iOS 9.0 and OS X 10.11.

With these changes, the threading model for Core Data has become a binary decision. We can use Core Data on the main thread (also known as the UI thread), or we can use Core Data on a background thread for things that don't directly impact the UI thread, which is a binary question. This evolution has been an incredibly good thing for Core Data. The question of how and when to use threads has been boiled down to a single Boolean question.

Starting with iOS 9.0 and OS X 10.11, the generic -init method of the NSManagedObjectContext has been deprecated. Previously, when you called this initializer an instance of NSManagedObjectContext was returned that was *associated* with the thread that created it.

This type of creation was one of the parts of Core Data and threading that confused developers and therefore was finally removed. Now, when you want a new instance of NSManagedObjectContext you need to explicitly specify what thread that the NSManagedObjectContext will be associated with:

```
NSManagedObjectContext *moc = nil;
moc = [[NSManagedObjectContext alloc] initWithConcurrencyType:${XXX}];
```

With this initializer, we can pass in two options for ${XXX}:

- NSMainQueueConcurrencyType: This option will configure the NSManagedObjectContext so that it can only be run on the main queue/thread.
- NSPrivateQueueConcurrencyType: This option will configure the NSManagedObjectContext so that it can only be run on a private queue/thread.

As mentioned, this is now a binary decision when we're initializing an NSManagedObjectContext. If the NSManagedObjectContext is going to be used with the user interface, then the NSMainQueueConcurrencyType will be used. Otherwise you *must* use the NSPrivateQueueConcurrencyType.

Working on the Main Queue

In general, working on the main queue hasn't changed from the original design. Assuming you're working with an NSManagedObjectContext that's configured to run on the main queue, you'd access that NSManagedObjectContext the exact same way as before.

The big difference is when your code is on *another* queue and you need to do some data work on the main queue. Getting that work onto the main queue has changed, fortunately for the better. This improvement is in the form of two methods: -performBlock: and -performBlockAndWait:.

Introducing -performBlock:

The goal of -performBlock: is to guarantee that a block of code is being executed on the correct queue, which is the queue that the NSManagedObjectContext is associated with. Therefore, if you have a block of code that you need to execute on the main queue against the NSManagedObjectContext associated with the main queue, you can do the following:

```
NSManagedObjectContext *moc = ...; //Reference to main queue context
[moc performBlock:^{
  NSFetchRequest *request = ...;
  //... Define the request
  NSError *error = nil;
  NSArray *results = [moc executeRequest:request error:&error];
  if (!results) {
    NSLog(@"Failed to fetch: %@\n%@", [error localizedDescription],
        [error userInfo]);
  }
  // Do something with the results
}];
```

In this example, you retrieve a reference to the existing NSManagedObjectContext that's instantiated against the main queue. From there you call -performBlock: and inside that block is where you do *all* of the work that needs to be performed on the main queue against Core Data.

The call to -performBlock: takes the block of code and puts in the "todo" list for the queue associated with the NSManagedObjectContext that it's called against. As soon as that queue gets to the block of code, it will be executed. Generally the execution happens right away, but if that queue is busy with another task (for example, it has another block of code to execute), then the block will be performed later.

Calling -performBlock: isn't a "blocking" call, which means that the queue that calls -performBlock: won't be halted or paused and the line of code after the -performBlock: call will be executed immediately—most likely before the block is executed.

What this also means is that the -performBlock: call is re-entrant. While you're inside one call to -performBlock:, you can kick off another call. Your second call to -performBlock: is guaranteed to be executed *after* the first call. Therefore, you could do something clever like this:

```
NSManagedObjectContext *moc = ...; //Reference to main queue context
[moc performBlock:^{
  [moc performBlock:^{
    NSError *error = nil;
    if (![moc save:&error]) {
      NSLog(@"Failed to save: %@\n%@", [error localizedDescription],
        [error userInfo]);
      abort();
    }
  }];
  NSFetchRequest *request = ...;
  //... Define the request
  NSError *error = nil;
  NSArray *results = [moc executeRequest:request error:&error];
  if (!results) {
    NSLog(@"Failed to fetch: %@\n%@", [error localizedDescription],
        [error userInfo]);
  }
  // Do something with the results
}];
```

And the -save: call would be executed *after* the data manipulation code. This effectively gives you a try/finally pattern.

Introducing -performBlockAndWait:

There are plenty of situations where you want to execute code on the main queue but you want your background queue (aka the non-main queue) to wait for that execution to be completed. That's where the API -performBlockAndWait: is used. The parameters are exactly the same, but the behavior is a little bit different.

The main difference is that this API call will *block* the calling queue until the block is completed. This also means that -performBlockAndWait: is *not* re-entrant.

Working off the Main Queue

Now that threading with Core Data has been reduced to a binary question, the *other* type of NSManagedObjectContext we'll look at is NSPrivateQueueConcurrencyType.

The primary difference between the two context types is what queue the context is associated with. When you're working with an NSMainQueueConcurrencyType context, the context automatically associates itself with the main queue. When you initialize an NSPrivateQueueConcurrencyType context, the context will associate itself with a non-main queue that's *private* to the context.

Private means that you can't access that queue directly. Calling dispatch_sync or dispatch_async on that queue is against the API. The *only* way to interact with a private queue context is through -performBlock: and -performBlockAndWait:.

This difference also means that *any* interaction with the private queue context must be inside a -performBlock: or -performBlockAndWait: call. The three exceptions to that rule are -initWithConcurrencyType:, -setParentContext:, and -setPersistentStoreCoordinator:. Any other interaction with a private queue context must be wrapped in a block.

As with a main queue context, any objects created or retrieved from a private queue context can only be accessed on that private queue. If you attempt to access those objects outside the code block, then you're violating the thread constraint rules of Core Data and will run into data integrity issues.

Interqueue Communication

Since objects created with or retrieved from a context can only be accessed on the queue associated with that context, the challenge becomes passing references to those objects between queues. This is arguably the biggest area where multiple threads with Core Data cause people the most issues.

If a reference to an object must be passed between queues, the best way to handle that hand-off is via the object's -objectID property. This property is designed to be safe to access from multiple queues and is a unique identifier to the object.

A Note About the NSManagedObjectID

This unique identifier is generally only guaranteed to reference the object for the current life cycle of the application. While the -objectID can be persisted through various means, that approach isn't recommended. There are several external actions that can void the reference and cause it to no longer function.

Once you have a reference to the objectID associated with an NSManagedObject, you can retrieve another reference to that NSManagedObject from another context through a few methods:

- -objectWithID: will return an object for any objectID passed to it. The danger with this method is that it's *guaranteed* to return an object, even if it has to return an empty shell pointing to a nonexisting object. This can happen if an objectID is persisted and restored in a later application life cycle.
- -existingObjectWithID: error: is a preferred method to use because it will give back an object if it exists and will return nil if no object exists for the objectID. The slight negative with this method is that it can perform I/O if the object isn't cached.
- -objectRegisteredForID: is the third option for object retrieval with objectID. This method will return the object if it's already registered in the context that the method is being called against. Generally this method is only useful if you already know that the object has previously been fetched in the context.

In addition to object hand-off between queues (the passing of an object reference from one queue to another), there's the handling of changes performed on a queue. By default, one context won't notify another context if an object has been changed. It's the responsibility of the developer to notify the other context of any changes. This is handled through the notification system.

Every time -save: is called against an NSManagedObjectContext, that context will broadcast a few notifications. The one that's useful for cross-context notifications is NSManagedObjectContextDidSaveNotification. That notification is fired once the save has completed successfully, and the notification object that's passed along includes all of the objects that were a part of the save.

If you have two contexts that you wish to keep in sync with each other, you can subscribe to this notification and then instruct the other context to consume the notification. For example, imagine that in your data controller you have two contexts—contextA and contextB—and you wish to keep them in sync. Once those contexts have been initialized, you can then subscribe to their notifications:

```
NSNotificationCenter *center = [NSNotificationCenter defaultCenter];
[center addObserver:self
           selector:@selector(contextASave:)
             forKey:NSManagedObjectContextDidSaveNotification
             object:contextA];
[center addObserver:self
           selector:@selector(contextBSave:)
```

```
            forKey:NSManagedObjectContextDidSaveNotification
            object:contextB];
```

In general you want notification observations to be as narrowly focused as possible. Although you could pass nil to the object: parameter, there would be no guarantee of who was broadcasting the notification, and you'd then need to filter inside the receiving method. By defining what objects we're willing to accept notifications from, we don't need to write defensive code in the receiving method and as a result we keep the receiving methods cleaner.

Once you see this notification, you can then consume it:

```
- (void)contextASave:(NSNotification*)notification
{
  [self.contextB performBlock:^{
  [self.contextB mergeChangesFromContextDidSaveNotification:notification];
  }];
}

- (void)contextBSave:(NSNotification*)notification
{
  [self.contextA performBlock:^{
  [self.contextA mergeChangesFromContextDidSaveNotification:notification];
  }];
}
```

With this implementation, every time contextA is saved, contextB will be notified and every time contextB is saved, contextA will be notified. Note that these -mergeChangesFromContextDidSaveNotification: calls should be wrapped in a -performBlock: to guarantee that they're being processed on the proper queue.

It should be noted that while the -mergeChangesFromContextDidSaveNotification: is being consumed, the context is also notifying any of its observers that changes are taking place. This means that there can be side effects to this call.

For example, if contextA has an NSFetchedResultsController associated with it and that NSFetchedResultsController has some expensive cell drawing associated with it, we can expect to see a performance hotspot while consuming notifications. The reason for that is that the processing of these notifications isn't threaded and the call to -mergeChangesFromContextDidSaveNotification: won't return until all of the cells associated with that NSFetchedResultsController have completed *their* processing. Worse, since NSFetchedResultsController and the associated cells are on the main queue, the *entire* application's user interface is effectively halted while these changes are being processed. This can result in some surprising user interface delays. The best way to avoid these types of performance issues is to keep the cells from taking too long to draw or to break the save notification up into smaller units that can be processed faster.