Extracted from:

Core Data, 2nd Edition

Data Storage and Management for iOS, OS X, and iCloud

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Core Data
Second Edition

Data Storage and Management for iOS, OS X, and iCloud

Marcus S. Zarra

Edited by Colleen Toporek

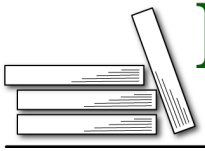
Core Data, 2nd Edition

Data Storage and Management for iOS, OS X, and iCloud

Marcus S. Zarra

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Colleen Toporek (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-08-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2013

5.5 Parent-Child NSManagedObjectContext Instances

With the release of iOS 6.0 and Mac OS X 10.8 Mountain Lion, a number of new features have been added to Core Data. One of the more interesting changes was the introduction of parent and child contexts. Parent-child contexts allow one context to be dependent upon another. These child contexts do not have direct access to the NSPersistentStoreCoordinator but instead are dependent on their parent context. This new feature has many subtle benefits, some of which we'll demonstrate here.

-performBlock: and -performBlockAndWait:

A lot of effort was put into the threading of Core Data with the release of iOS 6.0 and Mac OS X 10.8. To make threading a bit easier, two new methods have been introduced. Both methods accept a block and allow use of the NSManagedObjectContext, regardless of what thread we are currently running on.

-performBlock:

The purpose of -performBlock: is to allow code to execute on the correct thread for the associated NSManagedObjectContext. By utilizing this method, we can properly and safely access an NSManagedObjectContext without necessarily being on its thread. The -performBlock: executes the block on the thread associated with the NSManagedObjectContext. Note that this method does not block the calling thread. It should also be noted that this method is not "reentrant." For example, if you call -performBlock: within a -performBlock:, the new block is added to the end of the queue, as opposed to executing immediately.

-performBlockAndWait:

Like its counterpart -performBlock:, -performBlockAndWait: allows us to execute code against an NSManagedObjectContext regardless of the thread we are currently on. The primary difference between -performBlockAndWait: and -performBlock: is that -performBlockAndWait: is a blocking call. The calling thread waits for the block to finish execution before it continues. This also makes -performBlockAndWait: reentrant. You can nest -performBlockAndWait: calls infinitely, and they will execute in the order they are called.

Saving the NSManagedObjectContext

With the introduction of parent and child NSManagedObjectContext instances, we need to explore how saving works. While we continue to use -save: to commit changes in an NSManagedObjectContext, the result can vary depending on the context.

To start with, if we save an NSManagedObjectContext that is associated with an NSPersistentStoreCoordinator, the changes are written to the NSPersistentStoreCoordinator, which generally means the changes are written to disk.

However, when we call -save: on a child context *that is not associated* with an NSPersistentStoreCoordinator, the changes are *not* written to the NSPersistentStoreCoordinator. Instead, those changes are "pushed up" one level to the parent of the current NSManagedObjectContext. When the changes are pushed up, they effectively dirty the parent NSManagedObjectContext, and its -hasChanges method will then return YES. It should be noted that while the changes will get pushed up to the parent context, they will not get pushed down to any existing children. It is best to treat existing children as "snapshots" of the data taken at the time that the child was created.

Concurrency Types

There are a few requirements that must be satisfied in order to use parent and child contexts. Each NSManagedObjectContext that is associated in this way must be initialized with the new -initWithConcurrencyType: initializer. A concurrency type describes how an NSManagedObjectContext can be interacted with, because that relates to threading. Three concurrency types are available.

NSMainQueueConcurrencyType

The first type is called NSMainQueueConcurrencyType. This concurrency type is formally declared as accessible only from the main thread. An NSManagedObjectContext that is being used by the user interface should be defined with this concurrency type. It does not matter what thread we are currently on when we initialize an NSManagedObjectContext with this concurrency type because it must always be used on the main thread.

When accessed on the main thread, it can be treated normally. All access is available. However, if it is accessed from a background/nonmain thread, it can be accessed only via the -performBlock: and -performBlockAndWait: methods.

NSPrivateQueueConcurrencyType

The private queue concurrency type creates an NSManagedObjectContext that can be accessed *only* from its private queue. Because the queue is private, the NSManagedObjectContext can be used only via the -performBlock: and -performBlockAndWait: methods.

NSConfinementConcurrencyType

The confinement concurrency type is the "normal" concurrency type. When an NSManagedObjectContext is initialized using the -init method, this is the concur-

rency type that is configured. A confinement concurrency type means that the NSManagedObjectContext is confined to the thread that created it. If the NSManagedObjectContext is accessed from a thread other than the one that created it, an exception is thrown.

Let's walk through a couple of changes to our application in order to demonstrate the benefits of these new features.

Asynchronous Saving

One of the biggest issues with threading prior to iOS 6.0 and Mac OS X 10.8 Mountain Lion had to do with *thread blocking*. No matter how cleverly we wrote our import and export operations, sooner or later we *had to* block the main thread to let the main/UI NSManagedObjectContext "catch up." With the introduction of private queue contexts, this performance issue is finally solved.

If we start our Core Data stack with a private queue NSManagedObjectContext and associate it with the NSPersistentStoreCoordinator, we can have the main/UI NSManagedObjectContext as a child of the private queue NSManagedObjectContext. Furthermore, when the main/UI NSManagedObjectContext is saved, it will not produce a disk hit and will instead be nearly instantaneous. From there, whenever we want to actually write to disk, we can kick off a save on the private queue of the private context and get asynchronous saves.See Figure 17, *Private queue for asynchronous saves*, on page 8.

Adding this ability to our application requires a relatively small change. First, we need to add a property (nonatomic, strong) to hold onto our new private NSManagedObjectContext. Next, we tweak the -initializeCoreDataStack a little bit.

Baseline/PPRecipes/PPRAppDelegateAlt1.m

```
NSPersistentStoreCoordinator *psc = nil;
psc = [[NSPersistentStoreCoordinator alloc] initWithManagedObjectModel:mom];
ZAssert(psc, @"Failed to initialize persistent store coordinator");

NSManagedObjectContext *private = nil;
NSUInteger type = NSPrivateQueueConcurrencyType;
private = [[NSManagedObjectContext alloc] initWithConcurrencyType:type];
[private setPersistentStoreCoordinator:psc];

type = NSMainQueueConcurrencyType;
NSManagedObjectContext *moc = nil;
moc = [[NSManagedObjectContext alloc] initWithConcurrencyType:type];
[moc setParentContext:private];
[self setPrivateContext:private];

[self setManagedObjectContext:moc];
```

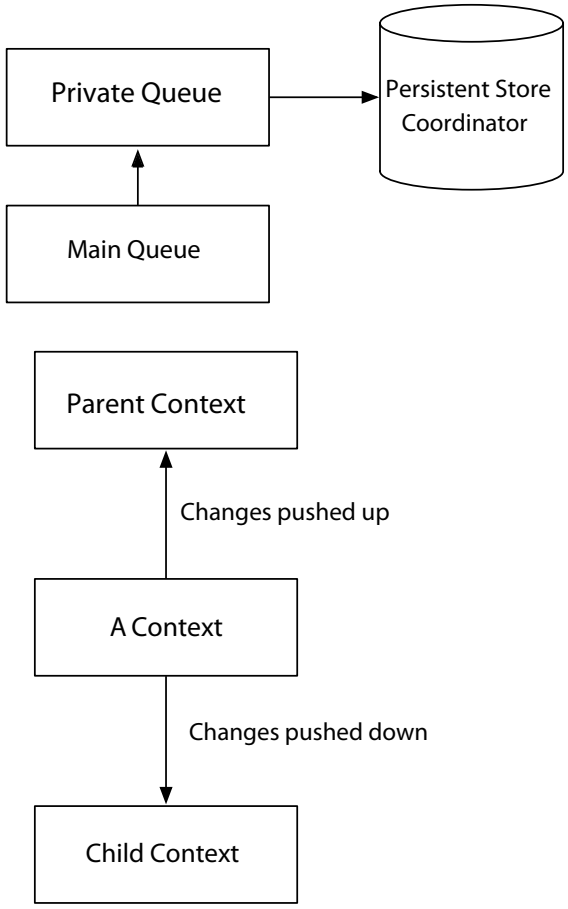


Figure 17—Private queue for asynchronous saves

Before, we had one NSManagedObjectContext configured to be on the main queue and writing to the NSPersistentStoreCoordinator. Now we have added a new NSManagedObjectContext that is of type NSPrivateQueueConcurrencyType. We set the NSPersistentStoreCoordinator to that private queue. Finally, we construct our main queue NSManagedObjectContext. Instead of handing off the NSPersistentStoreCoordinator to the main context, we give it a parent: the private queue context.

With that change, any saves on the main NSManagedObjectContext will push up the changes only to the private queue NSManagedObjectContext. No writing to the NSPersistentStoreCoordinator occurs. However, there are times when we really do want to write to disk and persist our data changes. In that case, a couple of other changes are in order.

Baseline/PPRecipes/PPRAppDelegateAlt1.m

```
- (void)saveContext:(BOOL)wait
{
  NSManagedObjectContext *moc = [self managedObjectContext];
  NSManagedObjectContext *private = [self privateContext];

  if (!moc) return;
  if ([moc hasChanges]) {
    [moc performBlockAndWait:^{
      NSError *error = nil;
      ZAssert([moc save:&error], @"Error saving MOC: %@\n%@",
              [error localizedDescription], [error userInfo]);
    }];
  }

  void (^savePrivate) (void) = ^{
    NSError *error = nil;
    ZAssert([private save:&error], @"Error saving private moc: %@\n%@",
            [error localizedDescription], [error userInfo]);
  };

  if ([private hasChanges]) {
    if (wait) {
      [private performBlockAndWait:savePrivate];
    } else {
      [private performBlock:savePrivate];
    }
  }
}
```

Previously in our -saveContext method, we checked to make sure we had an NSManagedObjectContext and that it had changes. We can still check to see whether we have an NSManagedObjectContext, since we create both of them at the same time. However, we now need to check both the main NSManagedObjectContext and the private NSManagedObjectContext for changes.

If the main NSManagedObjectContext has changes, we execute a -performBlockAndWait: to give the main NSManagedObjectContext all of the time that it needs to save. When the main has finished saving (or if it didn't need a save), we check the private context to see whether it also needs saving. When the private NSManagedObjectContext needs a save, we perform that save on the private context's queue. However, there are some situations in which we might want to block on the save and others where we might want it to be asynchronous. For example, when we are going into the background or terminating the application, we will want to block. If we are doing a save while the application is still running, we would want to be asynchronous. Fortunately, the -saveContext: method accepts a boolean that lets us know which method to call.

The only other alterations we need now are to change any calls from -saveContext to -saveContext: and pass in a boolean to determine whether the save blocks.

Updating Our PPRImportOperation

Another situation in which the parent-child context design provides a huge performance gain is when we need to merge changes between contexts. There's a good example of this in our current application, in the PPRImportOperation. Without the parent-child NSManagedObjectContext design, we must block the main thread during our save of the import. If there is only one recipe coming in, there probably won't be any issue. The save will be fast enough that the user won't notice. However, if we import hundreds of recipes, the save would cause a noticeable delay.

This can be fixed.

Baseline/PPRecipes/PPRImportOperationAlt1.m

```
NSManagedObjectContext *localMOC = nil;
NSUInteger type = NSConfinementConcurrencyType;
localMOC = [[NSManagedObjectContext alloc] initWithConcurrencyType:type];
[localMOC setParentContext:[self mainContext]];
```

The first change is to initialize our local NSManagedObjectContext as a type of NSConfinementConcurrencyType. This step locks the NSManagedObjectContext to our thread. Next, we configure the NSManagedObjectContext to be a child of the main NSManagedObjectContext.

The other changes are to remove code.

First, we no longer need to listen for change notifications. As soon as we save the local NSManagedObjectContext, the changes are moved up to our parent NSManagedObjectContext.

Since we are no longer listening for change notifications, we no longer need our -contextDidSave: method. Changes now propagate automatically.

These alterations are the only changes required. Once they're in place, we will no longer block the main thread on saves. We could even increase our save frequency so the user can see the recipes coming in one at a time, if we wanted. Without a disk I/O cost or a main thread block, we have a lot more options for how to handle the user experience.

These are just two examples of how the iOS 6.0/Mac OS X 10.8 changes to Core Data can improve the performance of our applications. As developers become more familiar with these new tools, we can expect to see many more innovative ways that Core Data will be used.

5.6 Wrapping Up

In this chapter, we explored the often-controversial subject of multithreading. There are a number of myths about Core Data and threading, but with this foundation, deciding whether to add multithreading to your Core Data application should be a great deal clearer.