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

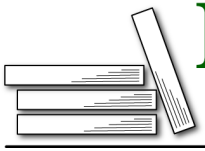
Core Data in Objective-C, Third Edition

Data Storage and Management for iOS and OS X

Marcus S. Zarra

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

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

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

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

CHAPTER 3

iOS: NSFetchedResultsController

The NSFetchedResultsController was introduced alongside Core Data when the framework was added to iOS with version 3.0. Since its introduction, developers have settled into a love–hate relationship with this class. When it's used in the way it was intended, it works extremely well. The hate part comes in when developers attempt to use the NSFetchedResultsController outside of its intended niche—that's when things start to fall apart quickly.

The NSFetchedResultsController is designed to be the glue between Core Data and the UITableView. When Core Data was first added to iOS, the Core Data team realized that there was a significant amount of work to get Core Data and table views talking well. They removed nearly all of that work with the introduction of the NSFetchedResultsController.

In this chapter, we discuss what the NSFetchedResultsController is designed to do and how it works. Once you have a handle on how it works, we'll explore alternatives so you know what to use when NSFetchedResultsController is not the correct fit.

How to Use the NSFetchedResultsController

When Core Data was added to iOS, it was clear to the Core Data team that Core Data and the UITableView would be used together extensively. They also realized that getting these two pieces to work together smoothly would require a fair amount of code that could be abstracted away; that abstraction is the NSFetchedResultsController. The NSFetchedResultsController is the glue that binds a UITableView to Core Data so that we need to write very little code.

The purpose of the NSFetchedResultsController is twofold. The NSFetchedResultsController is designed to retrieve data from Core Data and store that data for access. It does this with an internal NSFetchRequest that it uses for the retrieval. It then

stores the data and makes it available for use. As part of the storage and retrieval, the NSFetchedResultsController organizes the returned data into sections, in the process making the data more useful to a UITableView.

The NSFetchedResultsController's second purpose is to monitor changes in the data. Without the capability to be notified when the data has changed, the NSFetchedResultsController wouldn't be much more use than an NSArray. When the NSManagedObjectContext associated with the NSFetchedResultsController changes, the NSFetchedResultsController checks to see whether any of the objects it's referencing are impacted. Further, it watches inserts to determine whether any newly inserted objects should be included in what's being referenced. If any changes occur, the NSFetchedResultsController notifies its delegate of the changes. The delegate is normally its associated UITableView.

Standing Up an NSFetchedResultsController

The creation of a NSFetchedResultsController takes a number of steps and uses several of the classes that we discussed in Chapter 2, *Under the Hood of Core Data*, on page ?.

RecipesV1/PPRecipes/PPRMasterViewController.m

```
- (NSFetchedResultsController *)fetchedResultsController
{
  if (fetchedResultsController) return fetchedResultsController;
  NSManagedObjectContext *moc = [self managedObjectContext];

  NSFetchRequest *fetchRequest = nil;
  fetchRequest = [NSFetchRequest fetchRequestWithEntityName:@"Recipe"];

  NSSortDescriptor *sort = [[NSSortDescriptor alloc] initWithKey:@"name"
                                                       ascending:YES];
  [fetchRequest setSortDescriptors:[NSArray arrayWithObject:sort]];

  NSFetchedResultsController *frc = nil;
  frc = [[NSFetchedResultsController alloc] initWithFetchRequest:fetchRequest
                                            managedObjectContext:moc
                                              sectionNameKeyPath:nil
                                                       cacheName:@"Master"];

  [self setFetchedResultsController:frc];
  [[self fetchedResultsController] setDelegate:self];

  NSError *error = nil;
  ZAssert([[self fetchedResultsController] performFetch:&error],
          @"Unresolved error %@\n%@", [error localizedDescription],
          [error userInfo]);

  return fetchedResultsController;
}
```

The NSFetchedResultsController is effectively a wrapper around an NSFetchRequest. Therefore, we first need to construct the NSFetchRequest that will be used. In this example, we're building an NSFetchRequest that retrieves all of the available recipes. Further, we're going to sort the Recipe entities based on their name attribute.

Once we've built the NSFetchRequest, we construct the NSFetchedResultsController. In its initialization, it accepts an NSFetchRequest, an NSManagedObjectContext, an NSString for its sectionNameKeyPath, and another NSString for its cacheName. Let's explore each of these in turn.

NSFetchRequest

The NSFetchRequest retrieves the data from Core Data and makes it available for use. This is the NSFetchRequest we just defined in code.

NSManagedObjectContext

The NSFetchedResultsController requires an NSManagedObjectContext to perform the fetch against. Additionally, this is the NSManagedObjectContext that the NSFetchedResultsController will be monitoring for changes. Note that the NSFetchedResultsController is designed to work against user interface elements, and therefore, it works best when it's pointed at an NSManagedObjectContext that's running on the main/UI thread. Threading is discussed in more depth in Chapter 6, *Threading*, on page ?.

sectionNameKeyPath

The NSFetchedResultsController uses the sectionNameKeyPath to break the retrieved data into sections. Once the data is retrieved, the NSFetchedResultsController calls for a property on each entity using KVC (more on this in Chapter 10, *OS X: Bindings, KVC, and KVO*, on page ?). The value of that property will be used to break the data into sections. In our current example, we have this set to nil, which means our data will not be broken into sections. However, we could easily add it, as follows:

RecipesV1/PPRecipes/PPRMasterViewController.m

```
if (fetchedResultsController) return fetchedResultsController;
NSManagedObjectContext *moc = [self managedObjectContext];

NSFetchRequest *fetchRequest = nil;
fetchRequest = [NSFetchRequest fetchRequestWithEntityName:@"Recipe"];
NSMutableArray *sortArray = [NSMutableArray array];
[sortArray addObject:[[NSSortDescriptor alloc] initWithKey:@"type"
                                                 ascending:YES]];
[sortArray addObject:[[NSSortDescriptor alloc] initWithKey:@"name"
                                                 ascending:YES]];
```

```
[fetchRequest setSortDescriptors:sortArray];

NSFetchedResultsController *frc = nil;
frc = [[NSFetchedResultsController alloc] initWithFetchRequest:fetchRequest
                                          managedObjectContext:moc
                                            sectionNameKeyPath:@"type"
                                                     cacheName:@"Master"];
```

Something to note here is that along with passing in @"type" to the initialization of the NSFetchedResultsController, we also added a second NSSortDescriptor to the NSFetchRequest. The NSFetchedResultsController requires the data to be returned in the same order as it will appear in the sections. As a result, we must sort the data first by type and then by name.

cacheName

The last property of the initialization of the NSFetchedResultsController is the cacheName. This value is used by the NSFetchedResultsController to build up a small data cache on disk. That cache will allow the NSFetchedResultsController to skip the NSPersistentStore entirely when its associated UITableView is reconstructed. This cache can dramatically improve the launch performance of any associated UITableView.

However, this cache is extremely sensitive to changes in the data and the NSFetchRequest. Therefore, this cache name cannot be reused from one UITableView to another, nor can it be reused if the NSPredicate changes.

Once the NSFetchedResultsController has been initialized, we need to populate it with data. This can be done immediately upon initialization, as in our current example, or it can be done later. When to populate the NSFetchedResultsController is more of a performance question. If the associated UITableView is constructed very early, we may want to wait to populate the NSFetchedResultsController until the UITableView is about to be used. For now and until we can determine there's a performance issue, we'll populate it upon initialization. This is done with a call to -performFetch:, which takes a pointer to an NSError variable. If there's an error in the fetch, the NSError will be populated, and the call will return NO.

Wiring the NSFetchedResultsController to a UITableView

Now that we have our NSFetchedResultsController initialized, we need to wire it into its associated UITableView. We do this within the various UITableViewDatasource methods. At a minimum, we need to implement two of them but implementing the core three is better.

RecipesV1/PPRecipes/PPRMasterViewController.m

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
  return [[[self fetchedResultsController] sections] count];
}
```

The first one is -numberOfSectionsInTableView:. Here we ask the NSFetchedResultsController to return its array of sections, and we return the count of them. If we don't have a sectionNameKeyPath set on the NSFetchedResultsController, there will be either zero or one section in that array. In previous versions of iOS (prior to iOS 4.*x*), the UITableView did not like being told there were zero sections. You may run across older code that checks the section count and always returns a minimum of one section (with zero rows). That issue has been addressed, and the associated check is no longer needed.

RecipesV1/PPRecipes/PPRMasterViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
  NSArray *sections = [[self fetchedResultsController] sections];
  id <NSFetchedResultsSectionInfo> sectionInfo = nil;
  sectionInfo = [sections objectAtIndex:section];
  return [sectionInfo numberOfObjects];
}
```

The tableView: numberOfRowsInSection: method is slightly more complex. Here, we grab the array of sections, but we also grab the object within the array that's at the index being passed into the method. There's no need to check to see whether the index is valid since the -numberOfSectionsInTableView: method is the basis for this index. The object that's in the NSArray is undetermined but guaranteed to respond to the NSFetchedResultsSectionInfo protocol. One of the methods on that protocol is numberOfObjects, which we use to return the number of rows in the section.

RecipesV1/PPRecipes/PPRMasterViewController.m

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = nil;
  NSManagedObject *object = nil;

  object = [[self fetchedResultsController] objectAtIndexPath:indexPath];

  cell = [tableView dequeueReusableCellWithIdentifier:@"Cell"];
  [[cell textLabel] setText:[object valueForKey:@"name"]];

  return cell;
}
```

In the -tableView: cellForRowAtIndexPath: method, we use another very useful ability of the NSFetchedResultsController: the -objectAtIndexPath: method. With this method, we can retrieve the exact object we need to work with in a single call. This reduces the complexity of our -tableView: cellForRowAtIndexPath: method significantly.

There are many additional examples of how to wire in the NSFetchedResultsController to the UITableView, but these three highlight the most common usage. Even with just these three methods, you can see how the NSFetchedResultsController drastically reduces the amount of code you need to write (and thereby maintain) to access the data to be displayed.

Listening to the NSFetchedResultsController

In addition to making it very easy for us to retrieve and display the data for a UITableView, the NSFetchedResultsController makes it relatively painless to handle changes in that data. If the values within one of our recipes changes (perhaps through iCloud, as discussed in Chapter 8, *Using Core Data with iCloud*, on page ?, or through an import), we want our UITableView to immediately reflect those changes. In addition, if a recipe is removed or added, we want our UITableView to be accurate. To make sure these updates happen, we must add the delegate methods for the NSFetchedResultsControllerDelegate protocol. As mentioned, it's common for the UIViewController to also be the delegate for the NSFetchedResultsController. There are five methods in this protocol; let's take a look at each of them.

-controllerWillChangeContent:

The first method, -controllerWillChangeContent:, tells us that changes are about to start. This method is our opportunity to instruct the UITableView that changes are coming. Typically this is where we tell the UITableView to stop updating the user interface so that all of the changes can be displayed at once.

RecipesV1/PPRecipes/PPRMasterViewController.m

```
- (void)controllerWillChangeContent:(NSFetchedResultsController *)controller
{
  [[self tableView] beginUpdates];
}
```

-controller: didChangeSection: atIndex: forChangeType:

This method is called when a section changes. The only valid change types are NSFetchedResultsChangeInsert and NSFetchedResultsChangeDelete. This is our opportunity to tell the UITableView that a section is being added or removed.

RecipesV1/PPRecipes/PPRMasterViewController.m

```
- (void)controller:(NSFetchedResultsController *)controller
  didChangeSection:(id <NSFetchedResultsSectionInfo>)sectionInfo
           atIndex:(NSUInteger)sectionIndex
     forChangeType:(NSFetchedResultsChangeType)type
{
  NSIndexSet *indexSet = [NSIndexSet indexSetWithIndex:sectionIndex];
  switch(type) {
    case NSFetchedResultsChangeInsert:
      [[self tableView] insertSections:indexSet
                      withRowAnimation:UITableViewRowAnimationFade];
      break;

    case NSFetchedResultsChangeDelete:
      [[self tableView] deleteSections:indexSet
                      withRowAnimation:UITableViewRowAnimationFade];
      break;
    case NSFetchedResultsChangeUpdate:
    case NSFetchedResultsChangeMove:
      break;
  }
}
```

Here we use a switch to determine what the change type is and pass it along to the UITableView. You may note that there are two case statements at the end of this that don't react. They're added to satisfy the compiler and at this time aren't being used in this method.

-controller: didChangeObject: atIndexPath: forChangeType: newIndexPath:

This is the most complex method in the NSFetchedResultsControllerDelegate protocol. In this method, we're notified of any changes to any data object. The four types of changes that we must react to are listed next.

RecipesV1/PPRecipes/PPRMasterViewController.m

```
- (void)controller:(NSFetchedResultsController *)controller
   didChangeObject:(id)anObject
       atIndexPath:(NSIndexPath *)indexPath
     forChangeType:(NSFetchedResultsChangeType)type
      newIndexPath:(NSIndexPath *)newIndexPath
{
  NSArray *newArray = nil;
  NSArray *oldArray = nil;

  if (newIndexPath) newArray = [NSArray arrayWithObject:newIndexPath];
  if (indexPath) oldArray = [NSArray arrayWithObject:indexPath];
  switch(type) {
    case NSFetchedResultsChangeInsert:
      [[self tableView] insertRowsAtIndexPaths:newArray
                              withRowAnimation:UITableViewRowAnimationFade];
```

```
      break;
    case NSFetchedResultsChangeDelete:
      [[self tableView] deleteRowsAtIndexPaths:oldArray
                              withRowAnimation:UITableViewRowAnimationFade];
      break;
    case NSFetchedResultsChangeUpdate:
    {
      UITableViewCell *cell = nil;
      NSManagedObject *object = nil;
      cell = [[self tableView] cellForRowAtIndexPath:indexPath];
      object = [[self fetchedResultsController] objectAtIndexPath:indexPath];
      [[cell textLabel] setText:[object valueForKey:@"name"]];
      break;
    }
    case NSFetchedResultsChangeMove:
      [[self tableView] deleteRowsAtIndexPaths:oldArray
                              withRowAnimation:UITableViewRowAnimationFade];
      [[self tableView] insertRowsAtIndexPaths:newArray
                              withRowAnimation:UITableViewRowAnimationFade];
      break;
  }
}
```

An NSFetchedResultsChangeInsert is fired when a new object is inserted that we need to display in our UITableView. When we receive this call, we pass it along to the UITableView and tell the table view what type of animation to use.

An NSFetchedResultsChangeDelete is fired when an existing object is removed. Just as we do with an insert, we pass this information along to the UITableView and tell it what type of animation to use when removing the row.

An NSFetchedResultsChangeUpdate is fired when an existing object has changed internally—in other words, when one of its attributes has been updated. We don't know from this call whether it's an attribute that we care about. Instead of spending time determining whether we *should* update the row, it's generally cheaper to just update the row.

An NSFetchedResultsChangeMove is fired when a row is moved. The move could result from a number of factors but is generally caused by a data change resulting in the row being displayed in a different location. In our example, if the name or type of a recipe were altered, it would most likely cause this change type. It is completely possible—and quite common—to receive an NSFetchedResultsChangeMove and an NSFetchedResultsChangeUpdate in the same batch of changes. When this change type is received, we make two calls to the UITableView: one to remove the row from its previous location and another to insert it into its new location.

-controller: sectionIndexTitleForSectionName:

We use this method when we want to massage the data coming back from our NSFetchedResultsController before it's passed to the UITableView for display. One situation where this might be necessary is if we want to remove any extended characters from the title before it is displayed; another example is if we want to add something to the displayed title that's not in the data.

RecipesV1/PPRecipes/PPRMasterViewController.m

```
- (NSString*)controller:(NSFetchedResultsController*)controller
sectionIndexTitleForSectionName:(NSString*)sectionName
{
  return [NSString stringWithFormat:@"[%@]", sectionName];
}
```

-controllerDidChangeContent:

The final method tells us that this round of changes is finished and we can tell the UITableView to update the user interface. We can also use this method to update any other parts of the user interface outside of the UITableView. For example, if we had a count of the number of recipes displayed, we would update that count here.

RecipesV1/PPRecipes/PPRMasterViewController.m

```
- (void)controllerDidChangeContent:(NSFetchedResultsController *)controller
{
  [[self tableView] endUpdates];
}
```

With the implementation of the five methods described earlier, our UITableView can now retrieve, display, and update its display without any further work from us. In fact, a large portion of the code in the NSFetchedResultsControllerDelegate methods is fairly boilerplate and can be moved from project to project, further reducing the amount of "new" code we need to maintain.