Extracted from:

iCloud for Developers

Automatically Sync Your iOS Data, Everywhere, All the Time

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic Press

iCloud for Developers

Automatically Sync Your iOS Data, Everywhere, All the Time

Cesare Rocchi

edited by John Osborn

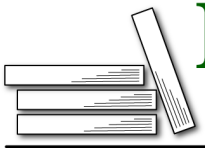
iCloud for Developers

Automatically Sync Your iOS Data, Everywhere, All the Time

Cesare Rocchi

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:

John Osborn (editor)
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: 978-1-937785-60-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2013

8.2 Initializing a Core Data Stack for iCloud

To initialize a classic Core Data application for iCloud, we must provide it with the following: a data model, a context, and a coordinator. The context mediates between the objects of an application (for example, the instances of grocery items) and the Core Data framework. When we change an object or create a new one, it's never stored directly in the underlying store; it's committed to the application's context. Only when we "flush" a context by calling the save: method are the changes made permanent in the local device, the data having first been checked for consistency. A call to the save: method also triggers the propagation of data to iCloud. As with UIDocument data, the changes may not be propagated immediately depending on the availability of an appropriate connection, status of the device battery, and other conditions determined by the operating system.

The coordinator, short for "persistent store coordinator," mediates between the context and the actual store (for example, a SQLite database). It is up to the coordinator to pick out the changes in the context and serialize them according to the type of Core Data storage (that is, SQLite, XML, or binary) being used.

Model, context, and coordinator are usually defined in the application delegate. In the Grocery-chp8-starter project for this chapter, these properties are defined in the SMAppDelegate class.

In the next two sections, let's see how the coordinator should be updated to work with iCloud and then do the same for the application context.

Modifying the Coordinator

To a create coordinator for any Core Data application, we must complete the following steps:

1. Create a reference to the database, which can be a SQLite, XML, or binary file.
2. Instantiate a coordinator, passing the instance of model.
3. Register the coordinator by means of the addPersistentStoreWithType:configuration:URL:options:error: method.

For example, here's the code to create a coordinator for the non-iCloud version of Grocery:

Grocery-chp8-starter/Grocery/SMAppDelegate.m

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator
{
```

```
    if (_persistentStoreCoordinator != nil) {
        return _persistentStoreCoordinator;
    }

    NSURL *storeURL = [[self applicationDocumentsDirectory]
                       URLByAppendingPathComponent:@"Grocery.sqlite"];

    NSError *error = nil;
    _persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
                                   initWithManagedObjectModel:
                                   [self managedObjectModel]];

    if (![_persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
                                                   configuration:nil
                                                             URL:storeURL
                                                         options:nil
                                                           error:&error]) {

        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }

    return _persistentStoreCoordinator;
}
```

To initialize a coordinator for iCloud, it's the third step that we need to change. There are two modifications to be made. The first is to create a dictionary of options required of an iCloud coordinator and pass it to the addPersistentStoreWithType:configuration:URL:options:error: method. The second is to make sure that the registration of the coordinator is nonblocking.

The dictionary has to include values for three properties.

NSPersistentStoreUbiquitousContentNameKey A name that uniquely identifies the store in the ubiquity container

NSPersistentStoreUbiquitousContentURLKey A path to a file that will store the transaction logs

NSMigratePersistentStoresAutomaticallyOption A boolean value to specify how to perform automatic migrations in the store

Before we build the dictionary, we first have to create the URL for the transaction log file whose name we'll assign to NSPersistentStoreUbiquitousContentURLKey. Here's the code to create a subfolder in the ubiquity container, named grocery_data, that we'll use for the Grocery application:

Grocery-chp8-end/Grocery/SMAppDelegate.m

```
NSFileManager *fileManager = [NSFileManager defaultManager];
```

```
NSURL *transactionLogsURL = [fileManager
                             URLForUbiquityContainerIdentifier:nil];

NSString* coreDataCloudContent = [[transactionLogsURL path]
                                  stringByAppendingPathComponent:
                                  @"grocery_data"];

transactionLogsURL = [NSURL fileURLWithPath:coreDataCloudContent];
```

With a URL for the transaction file in hand, we can now build a dictionary to initialize the coordinator for the Grocery database.

Grocery-chp8-end/Grocery/SMAppDelegate.m

```
NSDictionary* options = @{NSPersistentStoreUbiquitousContentNameKey :
                              @"com.studiomagnolia.coredata.grocery",
                          NSPersistentStoreUbiquitousContentURLKey:
                              transactionLogsURL,
                          NSMigratePersistentStoresAutomaticallyOption:
                              @YES
                          };
```

The final task is to register the coordinator in a way that doesn't block interaction with the user interface. Nothing is more frustrating to users. To avoid blocking, we should put registration into a background queue, a secondary thread that performs data synchronization with the servers and notifies the application when it's done. To do this, I've resorted to Grand Central Dispatch *(GCD)* and its dispatch_async() method.[5]

Once the registration of the coordinator has been completed, we will want to notify the application. Here's the code to do it for our Grocery project:

Grocery-chp8-end/Grocery/SMAppDelegate.m

```
dispatch_async(dispatch_get_main_queue(), ^{
    NSLog(@"persistent store added");
    [[NSNotificationCenter defaultCenter]
     postNotificationName:
     @"com.studiomagnolia.groceryItemsSynchronized"
     object:self
     userInfo:nil];
});
```

Now, let's put all of this together and define the method persistentStoreCoordinator. It belongs to the application delegate for Grocery, SMAppDelegate, as in the starter project.

5. If you are not familiar with GCD, you should read this document from Apple's documentation: https://developer.apple.com/library/mac/#documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html.

Grocery-chp8-end/Grocery/SMAppDelegate.m

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator
{
    if (_persistentStoreCoordinator != nil) {
        return _persistentStoreCoordinator;
    }

    _persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
                                   initWithManagedObjectModel:
                                   [self managedObjectModel]];

    NSString *storePath = [[self applicationDocumentsDirectory]
                           stringByAppendingPathComponent:@"Grocery.sqlite"];

    NSPersistentStoreCoordinator* psc = _persistentStoreCoordinator;

    dispatch_async(
       dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
       ^{

           NSURL *storeUrl = [NSURL fileURLWithPath:storePath];


           // building the path to store transaction logs
           NSFileManager *fileManager = [NSFileManager defaultManager];
           NSURL *transactionLogsURL = [fileManager
                                        URLForUbiquityContainerIdentifier:nil];

           NSString* coreDataCloudContent = [[transactionLogsURL path]
                                             stringByAppendingPathComponent:
                                             @"grocery_data"];

           transactionLogsURL = [NSURL fileURLWithPath:coreDataCloudContent];

           //  Building the options array for the coordinator
           NSDictionary* options = @{NSPersistentStoreUbiquitousContentNameKey :
                                         @"com.studiomagnolia.coredata.grocery",
                                     NSPersistentStoreUbiquitousContentURLKey:
                                         transactionLogsURL,
                                     NSMigratePersistentStoresAutomaticallyOption:
                                         @YES
                                     };

           NSError *error = nil;

           [psc lock];

           if (![psc addPersistentStoreWithType:NSSQLiteStoreType
                                  configuration:nil
                                            URL:storeUrl
```

```
                                          options:options
                                            error:&error]) {

            NSLog(@"Core data error %@, %@", error, [error userInfo]);

        }

        [psc unlock];

        // post a notification to tell the main thread
        // to refresh the user interface
        dispatch_async(dispatch_get_main_queue(), ^{
            NSLog(@"persistent store added");
            [[NSNotificationCenter defaultCenter]
             postNotificationName:
             @"com.studiomagnolia.groceryItemsSynchronized"
             object:self
             userInfo:nil];
        });
    });
    return _persistentStoreCoordinator;
}
```

Now let's move on to the last step in modifying Core Data for iCloud: context creation.

Modifying Context

To update the definition of context and make it iCloud aware, we need to make these three changes:

1. Choose a concurrency type to initialize the context with.
2. Set the persistent store coordinator.
3. Listen for NSPersistentStoreDidImportUbiquitousContentChangesNotification notifications.

Our choice of concurrency type determines how the context behaves when it's accessed concurrently by different threads. We can associate it with a private queue (which runs in the background) or with the main application thread (which is used to draw the user interface). In the case of the Grocery application, I've chosen to update the context with objects that live in the main thread, so we will initialize the context as follows:

Grocery-chp8-end/Grocery/SMAppDelegate.m

```
NSManagedObjectContext* moc = [[NSManagedObjectContext alloc]
                               initWithConcurrencyType:
                               NSMainQueueConcurrencyType];
```

This line of code declares the context to be queue-based, so to set its properties, we need to use either the performBlockAndWait: or performBlock: method. For example, to set the coordinator of the context, we write the following:

```
[moc performBlockAndWait:^{
            [moc setPersistentStoreCoordinator:coordinator];
}];
```

Having specified the concurrency type, the last step is to add an observer to the context to listen for iCloud notifications that the store has been modified. The name of the notification that we need to listen for is pretty long:

NSPersistentStoreDidImportUbiquitousContentChangesNotification.

This notification will be thrown when there are changes in the content of the persistent store (for example, when a new item is created or edited).

Pulling all of it together, here is the new definition of context:

Grocery-chp8-end/Grocery/SMAppDelegate.m

```
- (NSManagedObjectContext *)managedObjectContext
{
    if (_managedObjectContext != nil) {
        return _managedObjectContext;
    }

    NSPersistentStoreCoordinator *coordinator =
    [self persistentStoreCoordinator];
    if (coordinator != nil) {
        // choose a concurrency type for the context
        NSManagedObjectContext* moc = [[NSManagedObjectContext alloc]
                                       initWithConcurrencyType:
                                       NSMainQueueConcurrencyType];

        [moc performBlockAndWait:^{
            // configure context properties
            [moc setPersistentStoreCoordinator: coordinator];
            [[NSNotificationCenter defaultCenter]

             addObserver:self
             selector:@selector(mergeChangesFromiCloud:)
             name:NSPersistentStoreDidImportUbiquitousContentChangesNotification
             object:coordinator];
        }];
        _managedObjectContext = moc;
    }
    return _managedObjectContext;
}
```

The method mergeChangesFromiCloud: that appears in this code is defined as follows:

Grocery-chp8-end/Grocery/SMAppDelegate.m

```
- (void)mergeChangesFromiCloud:(NSNotification *)notification {
    NSManagedObjectContext* moc = [self managedObjectContext];
    NSDictionary *noteInfo = [notification userInfo];

    [moc performBlock:^{
        NSMutableDictionary *mergingPolicyResult = [NSMutableDictionary dictionary];
        [mergingPolicyResult setObject:noteInfo[NSInsertedObjectsKey]
                               forKey:NSInsertedObjectsKey];
        [mergingPolicyResult setObject:noteInfo[NSUpdatedObjectsKey]
                               forKey:NSUpdatedObjectsKey];
        [mergingPolicyResult setObject:[NSSet set] // Exclude deletions
                               forKey:NSDeletedObjectsKey];
        NSNotification *saveNotification =
        [NSNotification notificationWithName:notification.name
                                      object:self
                                    userInfo:mergingPolicyResult];

        [moc mergeChangesFromContextDidSaveNotification:saveNotification];
        [moc processPendingChanges];
    }];
}
```

This method is responsible of merging changes notified from iCloud into the local store, applying the default merge policy whenever conflicts occur. I will provide more details about custom policies for conflict resolution in Section 8.3, *Handling Conflicts*, on page ?.

In the code, on page 8, a notification is posted when the data synchronization done at startup is finished. You need to listen for that notification. I have set up the observer at the end of viewDidLoad of SMMasterViewController, as follows:

```
[[NSNotificationCenter defaultCenter]
     addObserver:self
     selector:@selector(reloadItems)
     name:@"com.studiomagnolia.groceryItemsSynchronized"
     object:nil];
```

The reloadItems method that appears in the preceding code triggers a reload of the table and is defined like this:

Grocery-chp8-end/Grocery/SMMasterViewController.m

```
- (void) reloadItems {
    NSLog(@"============ reloading items");
    NSError *error = nil;

    if (![[self fetchedResultsController] performFetch:&error]) {
```

```
        NSLog(@"Core data error %@, %@", error, [error userInfo]);

    } else {
        [self.tableView reloadData];
    }
}
```

Finally, we need to refactor applicationDocumentsDirectory in SMAppDelegate as follows and change its signature in the header accordingly:

Grocery-chp8-end/Grocery/SMAppDelegate.m

```
- (NSString *)applicationDocumentsDirectory {
        return [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                    NSUserDomainMask, YES) lastObject];
}
```

Now it's time test the new Grocery application on some real devices. Install it on an iPhone or iPad and add a few items with a bunch of tags. Then install it on a second iOS device and confirm that the items on the first are correctly propagated. Keep both applications open, change items on one device, and then look to confirm they've been propagated to the other.

Now that you've learned the basics of iCloud-enabling a Core Data application, you're ready for something more complex: conflict resolution and related policies.