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

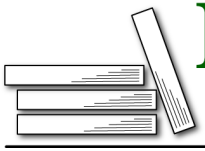
Core Data, 2nd Edition

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

6.1 Introducing the UIManagedDocument

In the release of iCloud, Apple introduced a new API called UIDocument. UIDocument is designed to be an abstract parent class that makes it easy to integrate applications with iCloud. One of the Core Data API changes for iOS 6.0 is UIManagedDocument, a subclass of UIDocument.

Fundamentally, the biggest advantage of using UIManagedDocument is the ability to abstract away the saving and state of your Core Data stack. With UIManaged-Document, saving is handled automatically and generally occurs asynchronously. You can request saves to occur more frequently than the autosave handles, but in general you shouldn't need to do that. In addition to managing the saving of the Core Data stack, UIManagedDocument has added features that allow you to store files outside of the persistent store that will also be pushed to iCloud.

UIManagedDocument is meant to be used in applications that have a document design. Pages, Numbers, Omnigraffle, and so on, are great examples of iOS applications that manage a form of document. Having said that, however, there is nothing stopping you from using a UIManagedDocument as your single Core Data stack enclosure. It is not specifically designed for a single stack design, but it will work. It is even appealing in some ways, since it abstracts out the creation of the stack.

iCloud/PPRecipes/PPRAppDelegate.m

```
dispatch_queue_t queue;
queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{
  NSFileManager *fileManager = [NSFileManager defaultManager];
  NSURL *storeURL = nil;
  storeURL = [[fileManager URLsForDirectory:NSDocumentDirectory
                                  inDomains:NSUserDomainMask] lastObject];
  storeURL = [storeURL URLByAppendingPathComponent:@"PPRecipes"];

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

The first step in constructing a UIManagedDocument is to resolve the file URL where we will store the document and determine what the file URL is going to be for iCloud. The first step, where we are saving the data to, is virtually the same as when we construct a Core Data stack. We determine where the documents directory is located via a call to the NSFileManager, and then we append a path component to the end to identify our document. In this case, the document is called PPRecipes.

The second URL—the iCloud URL—is a bit different. -URLForUbiquityContainerIdentifier: is a new addition to the NSFileManager that came with iCloud. This call requests a URL in which to store information that iCloud is going to use to sync the NSPersistentStore. If iCloud is disabled on the device (iOS or OS X), this call returns nil. Once we have a URL and it is not nil, we know iCloud is available, and we need to configure it.

It is the second call that has an interesting issue; specifically, this call can take an indeterminate amount of time. If iCloud is not available or if the directory structure had previously been constructed, this call could return nearly instantaneously. However, if iCloud is enabled and the directory structure needs to be constructed, the call might take a significant amount of time—long enough that we need to worry about the watchdog killing it for taking too long to launch. Therefore, because of this method call, we need to wrap all of this construction in a dispatch queue and let it run asynchronously with the main thread.

Once we know whether iCloud is available, we can continue with our configuration.

Configuring iCloud

To add iCloud to a Core Data stack, we need to add some additional options to the NSPersistentStore when we add the NSPersistentStore to the NSPersistentStoreCoordinator.

iCloud/PPRecipes/PPRAppDelegate.m

```
NSMutableDictionary *options = [[NSMutableDictionary alloc] init];
[options setValue:[NSNumber numberWithBool:YES]
           forKey:NSMigratePersistentStoresAutomaticallyOption];
[options setValue:[NSNumber numberWithBool:YES]
           forKey:NSInferMappingModelAutomaticallyOption];

if (cloudURL) {
  cloudURL = [cloudURL URLByAppendingPathComponent:@"PPRecipes"];

  [options setValue:[[NSBundle mainBundle] bundleIdentifier]
             forKey:NSPersistentStoreUbiquitousContentNameKey];
  [options setValue:cloudURL
             forKey:NSPersistentStoreUbiquitousContentURLKey];
}
```

The first part of this code should be familiar. We create an NSMutableDictionary and add the options both to infer a mapping model and to attempt a migration automatically. From there, if iCloud is enabled, we need to add the iCloud URL to our options dictionary. However, we do not want our document stored at the “root” of our iCloud sandbox. Rather, we want to create a directory

under the root with the same name as the document we are creating locally. Therefore, we are going to append "PPRecipes" to the end of the URL. Once the URL is defined, we need to add it to our options dictionary with the key NSPersistentStoreUbiquitousContentURLKey.

In addition to the URL for the storage location, we need to tell iCloud what data it is going to sync. If we have a single application shared between iPhone and iPad, as in our current example, we can use the bundle identifier as a unique key to define what data is to be shared across the devices. However, if we are also sharing data with a desktop application, the bundle identifier may not be appropriate. The data identifier is stored in the options dictionary with the key NSPersistentStoreUbiquitousContentNameKey.

The addition of these two keys is the bare minimum required to enable iCloud for an iOS application. With that information, the operating system creates a directory for the content, downloads any content that exists in the cloud, and begins syncing the data. However, as with the URL call, the initial download (or for that matter subsequent syncing) can take an indeterminate amount of time. If there is nothing currently in the store, the creation of the directory structure will be virtually instantaneous. But if there is data to download, it could take some time, depending on the speed of the network connection and the amount of data. Therefore, the application needs to be able to handle a delay in the creation of the persistent store. There are many ways to deal with this delay, and that is an exercise left to the user experience experts.

Building the UIManagedDocument

Once the options dictionary has been constructed, it's time to build the UIManagedDocument. The construction of the document itself is short.

iCloud/PPRecipes/PPRAppDelegate.m

```
UIManagedDocument *document = nil;
document = [[UIManagedDocument alloc] initWithFileURL:storeURL];
[document setPersistentStoreOptions:options];

NSMergePolicy *policy = [[NSMergePolicy alloc] initWithMergeType:
                         NSMergeByPropertyObjectTrumpMergePolicyType];
[[document managedObjectContext] setMergePolicy:policy];

void (^completion)(BOOL) = ^(BOOL success) {
  if (!success) {
    ALog(@"Error saving %@\n%@", storeURL, [document debugDescription]);
    return;
  }
```

```
    dispatch_queue_t mainQueue;
    mainQueue = dispatch_get_main_queue();

    dispatch_sync(mainQueue, ^{
      [self contextInitialized];
    });
  };

  if ([[NSFileManager defaultManager] fileExistsAtPath:[storeURL path]]) {
    [document openWithCompletionHandler:completion];
    return;
  }

  [document saveToURL:storeURL
     forSaveOperation:UIDocumentSaveForCreating
    completionHandler:completion];
  [self setManagedDocument:document];
});
```

Constructing the UIManagedDocument is a case of calling +alloc and then -initWithFileURL: and passing in the storeURL that we previously constructed. Once the UIManagedDocument is initialized, we can set the options for the NSPersistentStore via a call to -setPersistentStoreOptions:. Note that we do not have the ability to add more than one NSPersistentStore to a UIManagedDocument.

We also want to take this opportunity to set the merge policy for the UIManagedDocument. This setting is actually performed on the NSManagedObjectContext directly.

Unlike when we construct a straight Core Data stack, though, the initialization of the UIManagedDocument is not the end for us. We must now save it. We could save it later, but it is best to put all of this initialization code in the same place rather than have it spread out across our UIApplicationDelegate. To save the UIManagedDocument, we must first discover if it already exists; based on that information, we can call the appropriate method.

Whether the UIManagedDocument existed before, the process is the same: we call a method on the UIManagedDocument and pass in a completion handler. Since that completion handler is the same no matter which method we call, we construct the completion handler first and then determine which method to call.

In the completion handler, we check to see whether it completed successfully. If it was not successful, we present an error to the user and perhaps try to recover from the error. If the completion was successful, we want to notify the AppDelegate that the UIManagedDocument has been initialized and that normal program flow can resume.

With the completion block constructed, we can now ask the NSFileManager if the file already exists; if it does, we call -openWithCompletionHandler: on the UIManagedDocument. If it does not exist, we need to create it with a call to -saveToURL: forSaveOperation:UIDocumentSaveForCreating: completionHandler:. If this seems overly complicated, that's because it is. This really should be abstracted away into the framework.

Observing Changes to the UIManagedDocument

Once our UIManagedDocument has been constructed, it can be quite useful to know its current state. Since the UIManagedDocument saves on its own accord, we won't automatically know whether it is clean or dirty. We need some kind of callback system in place to notify us. Fortunately, the UIManagedDocument does broadcast notifications when the state changes. By adding our UIApplicationDelegate as an observer to the notification UIDocumentStateChangedNotification, we are notified of those changes and can act accordingly.

iCloud/PPRecipes/PPRAppDelegate.m

```
- (void)contextInitialized;
{
  DLog(@"fired");
  NSNotificationCenter *center = [NSNotificationCenter defaultCenter];
  [center addObserver:self
             selector:@selector(documentStateChanged:)
                 name:UIDocumentStateChangedNotification
               object:[self managedDocument]];
```

There are several places that we could start observing this notification; placing it in the -contextInitialized is a personal preference. It is possible to start listening to it as part of the initialization of the UIManagedDocument, for example. When this notification fires, we receive the UIManagedDocument as the object of the notification. From the UIManagedDocument, we then respond accordingly.

iCloud/PPRecipes/PPRAppDelegate.m

```
- (void)documentStateChanged:(NSNotification*)notification
{
  switch ([[notification object] documentState]) {
    case UIDocumentStateNormal:
      DLog(@"UIDocumentStateNormal");
      break;
    case UIDocumentStateClosed:
      DLog(@"UIDocumentStateClosed %@", notification);
      break;
    case UIDocumentStateInConflict:
      DLog(@"UIDocumentStateInConflict %@", notification);
      break;
    case UIDocumentStateSavingError:
      DLog(@"UIDocumentStateSavingError %@", notification);
```

```
      break;
    case UIDocumentStateEditingDisabled:
      DLog(@"UIDocumentStateEditingDisabled %@", notification);
      break;
  }
}
```

From the state of the UIManagedDocument, we can update our user interface to reflect that state and let the user know what is going on with the underlying data.

Manually Saving a UIManagedDocument

By default, the UIManagedDocument works to ensure that our data is saved as frequently as makes sense.

The UIDocument design knows to listen for UIApplicationWillResignActiveNotification, UIApplicationDidEnterBackgroundNotification, and UIApplicationWillTerminateNotification notifications. When it receives one of these notifications, it saves. It also saves periodically during the life of the application. On average, these periodical saves take place every five minutes.

However, we know our application better than the frameworks do. We know when something nonrecoverable or vital has just occurred, and we can decide that a save is mandatory at a specific point. Fortunately, it is possible to convey that need to UIManagedDocument.

iCloud/PPRecipes/PPRAppDelegate.m

```
NSURL *fileURL = [[self managedDocument] fileURL];
[[self managedDocument] saveToURL:fileURL
                 forSaveOperation:UIDocumentSaveForCreating
                completionHandler:^(BOOL success) {
                  //Handle failure
                }];
```

The call to request a save is the same we used when we were initially creating the UIManagedDocument. Further, we can request the URL for the save directly from the UIManagedDocument. The only detail left is planning how to properly respond to a failed save.