Extracted from:

iCloud for Developers

Automatically Sync Your iOS Data, Everywhere, All the Time

This PDF file contains pages extracted from *iCloud for Developers*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2013 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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

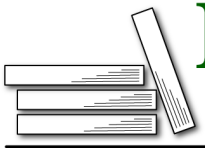
iCloud for Developers

Automatically Sync Your iOS Data, Everywhere, All the Time

Cesare Rocchi

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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)

Copyright © 2013 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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

As useful as synchronized key-value pairs can be, most applications must work with text, photos, video, music, or other data that is stored in files (or in Apple's terminology, *documents*).

To support document data, iCloud provides a second type of storage known as *document storage*. Unlike key-value data, the amount of document storage available to a user is limited only by the quota associated with the user's iCloud account, and documents can be used to store any type of data an application might require, even Core Data, as we'll see in *Working with Core Data and iCloud*.

In this chapter, we are going to take a close look at the document-based approach to data storage. First you will learn how the interaction between your application files and iCloud is handled by means of a daemon, the background process we explained in Chapter 1, *Preparing Your Application for iCloud*, on page ?. Then you will learn how to work with `UIDocument`, the class that provides an easy way to store and retrieve files so changes are propagated seamlessly to other devices via iCloud. And finally, you'll modify the Grocery application to store its shopping list items in single files. You'll start with a single item (in one file) in this chapter. In the following chapter, you'll see how to use a collection of files to implement a list with more than one item.

3.1 Interacting with iCloud

Building a document-based application means manipulating files in a way that the background process—the *daemon*—will know how to send and retrieve changes to their content to and from iCloud. As a programmer, you'll never interact directly with the daemon, and you'll never have to write code to tell it, say, to "synchronize now." Instead, whenever your application must read or write to a document, you simply open or close the file using appropriate methods. The daemon handles the locking of the file and determines when it is safe to read or write to it. As the developer, your only tasks are to open and close the file as needed and to declare how to encode or decode its data. These operations are facilitated by document storage's double queue architecture, shown in Figure 9, *The double queue architecture for the open operation*, on page 8.

The queues are threads that run on each device. The first queue is the main thread of the application, the one you can pilot via code and that draws the user interface. The second is the daemon, the background queue operated by iOS, which does all the read, write, and sync operations. This architecture

• Click HERE to purchase this book now. discuss

is shared by all three types of iCloud data storage described in Section 2.1, *iCloud Storage Types*, on page ?.

To store the grocery items generated by the user as plain files in the ubiquity container, you must learn how to extend the UIDocument class. UIDocument is handy because it already implements most of the functionality you need to interact with iCloud, leaving you with the tasks of mapping document contents into in-memory data structures when the file is opened and "dumping" them when the document is saved. Let's see what's required to extend UIDocument.

3.2 Extending the UIDocument Class

The easiest way to get started with document-based storage is to use UIDocument, a class meant to be extended to handle documents. A document is simply a collection of data that can be written to local storage as a single file or package of files (explained in Chapter 5, *Wrapping Items in a Single File*, on page ?). UIDocument provides two methods for reading data from a file via the daemon: openWithCompletionHandler and loadFromContents.

To read a file named doc that is an instance of UIDocument, here is the code you write:

```
[doc openWithCompletionHandler:^(BOOL success) {
  // code executed when the open has completed
}]
```

This simple call triggers a read operation on the background queue, the app's first point of contact with the daemon. You don't need to know whether the file is local (already pulled from iCloud) or still on the servers. All this is managed by the daemon, which notifies the main thread when it's done by calling the code in the block that you specify in openWithCompletionHandler:. As a result, the main thread is never blocked, and the user can continue working with the application while data in the file is retrieved. Of course, if that file is a resource that the application needs to continue, you can block further interaction with the user and display a spinner while the application waits for the file to be loaded.

Once the read operation is complete, the data contained in the file is loaded into the application. This is where you have an opportunity to code your own custom behavior to specify how data contained in a file is to be decoded. The method to override is loadFromContents:ofType:error:, which belongs to UIDocument.

```
- (BOOL) loadFromContents:(id)contents
                   ofType:(NSString *)typeName
                    error:(NSError **)outError {
            // decode data here
```

```
        return YES;
    }
```

loadFromContents:ofType:error: is called when the daemon has completed the read operation in the background.[1] One of its key parameters is contents, which is usually of type NSData; it contains the actual information you need to create your data within the application. This is the place where you decode data and save it in a local variable for future use. This method is called before the completion block specified in openWithCompletionHandler:. Figure 9, *The double queue architecture for the open operation*, on page 8 shows a diagram of this flow over time.

The write procedure is pretty similar, and it is based on the same double queue architecture. The key difference when writing is that you have to convert your document's contents to NSData. In essence, you have to provide a "snapshot" of the current situation of your document. To explicitly save a document, you can call saveToURL:forSaveOperation:completionHandler:, like so:

```
  [doc  saveToURL:[NSURL ...]
 forSaveOperation:UIDocumentChangeDone
completionHandler:^(BOOL success) {
              // code run when saving is done
  }];
```

Like the read operation, there is a completion block, triggered to notify that the operation has been completed. When the write is triggered on the background queue, the daemon will ask for a snapshot of the document by calling contentsForType:error:. This is the place where you need to encode the information stored in your local variables and return them, usually as an instance of NSData.

```
- (id) contentsForType:(NSString *)typeName
                 error:(NSError **)outError {

    // encode data here and return them, usually as NSData
}
```

Figure 10, *The double queue architecture for the save operation*, on page 8 shows the flow when saving an instance of UIDocument.

You can also work with a collection of files by storing them in a *file package*. Like an .app file, used to wrap iOS and Mac OS applications, a package is a directory that contains one or more files but is treated as a single file. I will

1. The ofType: parameter allows you to specify the *uniform type identifier (UTI)*. As you will see in Chapter 5, *Wrapping Items in a Single File*, on page ?, you can create a custom document file type.

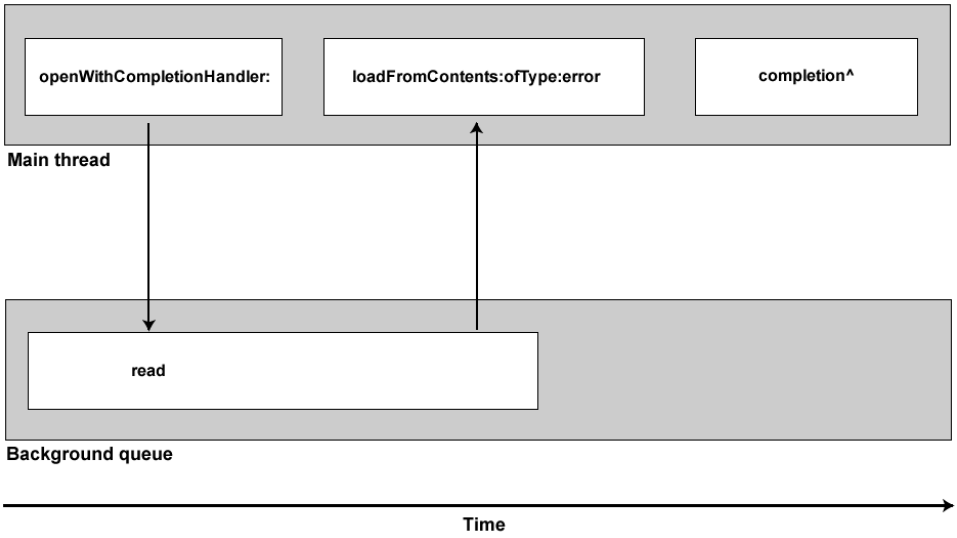


Figure 9—The double queue architecture for the open operation. This diagram shows the sequence of actions that occur under the hood when you open a file stored in the ubiquity container. The job of the daemon is illustrated in the background queue.

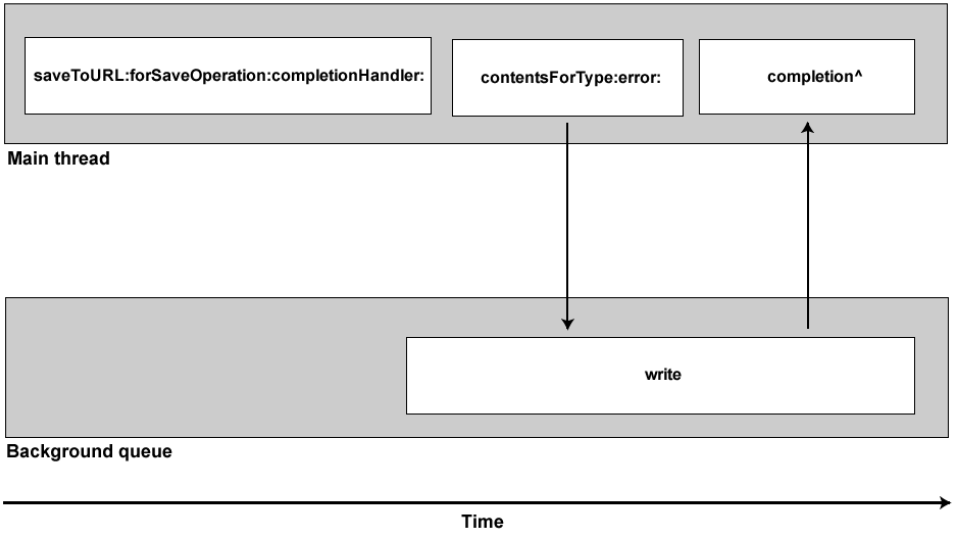


Figure 10—The double queue architecture for the save operation. This diagram shows the sequence of actions that occur under the hood when you save a file stored in the ubiquity container. The job of the daemon is illustrated in the background queue.

provide more details in Section 5.1, *Working with File Packages*, on page ?, where I'll use a package to store the grocery items.

In iCloud-enabled applications there is no need to explicitly call a save method, because UIDocument implements a save-less model. This means that the operating system saves data automatically at intervals. There is a method of UIDocument called hasUnsavedChanges, which returns whether an instance has been modified. When the return value is YES, the save procedure is triggered. There are two ways to influence the return value of this method.

- Explicitly call updateChangeCount:
- Use the *undo manager*, which enables quickly implementing undo and redo changes on a document

Undo Manager

UIDocument has a built-in undo manager. This enables to you implement undo and redo functionality when, for example, a user edits a document. You can access the undo manager of a UIDocument via the property undoManager. This returns an instance of NSUndoManager, which has helper methods to allow the implementation of undo and redo functionalities. If you use an undo manager, you do not need to call updateChangeCount:. For more details about the undo manager, visit this link: http://developer.apple.com/library/ios/#documentation/DataManagement/Conceptual/DocumentBasedAppPGiOS/ChangeTrackingUndo/ChangeTrackingUndo.html#//apple_ref/doc/uid/TP40011149-CH5-SW1.

Either method will tell the daemon that something has changed and that it should start the save procedure.

Notice that in either case, data may not be pushed immediately to iCloud and in turn to other devices. The calls to these methods are just "hints" to the background queue. The daemon tries to push metadata as soon as possible, whereas actual data is pulled by the cloud when appropriate, depending, for example, on the type of device and the quality of the connection.

Summing up, when we subclass UIDocument, we need to override the following two methods:

- loadFromContents:ofType:error:
- contentsForType:error:

The first method is called when the file is opened and allows the developer to "decode" the information and store it in an object or a property. The second is called when the file is saved and requires the developer to create a sort of

• Click HERE to purchase this book now. discuss

"screenshot" of the current information held in the object to be written in the iCloud container.

Now that you've mastered the basics of extending a `UIDocument`, let's move on to learn how to model a single grocery item, which will become the building block of our Grocery application.