Erica Sadun

Includes Nine Additional Chapters on Advanced Topics

The iOS 5 Developer's Cookbook

The Additional Recipes

Developer's Library

The iOS 5 Developer's Cookbook: Additional Recipes

Additional Recipes Found Only in the Expanded Electronic Edition

Erica Sadun

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

ISBN-13: 978-0-133-02839-3
ISBN-10: 0-133-02839-9

Senior Acquisitions Editors
Chuck Toporek
Trina MacDonald

Managing Editor
Kristy Hart

Project Editor
Anne Goebel

Proofreader
Sheri Cain

Editorial Assistant
Olivia Basegio

❖

I dedicate this book with love to my husband, Alberto,
who has put up with too many gadgets and too
many SDKs over the years while remaining both
kind and patient at the end of the day.

❖

Contents at a Glance

Acknowledgments

This book would not exist without the efforts of Chuck Toporek (my editor and whip-cracker) and Olivia Basegio (the faithful and rocking editorial assistant who kept things rolling behind the scenes). Also, a big thank you to the entire Addison-Wesley/Pearson production team, specifically Kristy Hart, Anne Goebel, and Sheri Cain. Thanks also to the crew at Safari for getting my book up in Rough Cuts and for quickly fixing things when technical glitches occurred.

Thanks go as well to Neil Salkind, my agent of many years and to all my colleagues, both present and former, at TUAW, Ars Technica, and the Digital Media/Inside iPhone blog.

I am deeply indebted to the wide community of iOS developers, including Tim Isted, Joachim Bean, Aaron Basil, Roberto Gamboni, John Muchow, Scott Mikolaitis, Alex Schaefer, Nick Penree, James Cuff, Jay Freeman, Mark Montecalvo, August Joki, Max Weisel, Optimo, Kevin Brosius, Planetbeing, Pytey, Michael Brennan, Daniel Gard, Michael Jones, Roxfan, MuscleNerd, np101137, UnterPerro, Jonathan Watmough, Youssef Francis, Bryan Henry, William DeMuro, Jeremy Sinclair, Arshad Tayyeb, Daniel Peebles, ChronicProductions, Greg Hartstein, Emanuele Vulcano, Sean Heber, Josh Bleecher Snyder, Eric Chamberlain, Steven Troughton-Smith, Dustin Howett, Dick Applebaum, Kevin Ballard, Hamish Allan, Kevin McAllister, Jay Abbott, Tim Grant Davies, Chris Greening, Landon Fuller, Wil Macaulay, Stefan Hafeneger, Scott Yelich, chrallelinder, John Varghese, Andrea Fanfani, J. Roman, jtbandes, Artissimo, Aaron Alexander, Christopher Campbell Jensen, rincewind42, Nico Ameghino, Jon Moody, Julián Romero, Scott Lawrence, Evan K. Stone, Kenny Chan Ching-King, Matthias Ringwald, Jeff Tentschert, Marco Fanciulli, Neil Taylor, Sjoerd van Geffen, Absentia, Nownot, Emerson Malca, Matt Brown, Chris Foresman, Aron Trimble, Paul Griffin, Paul Robichaux, Nicolas Haunold, Anatol Ulrich (hypnocode GmbH), Kristian Glass, Remy Demarest, Yanik Magnan, ashikase, Shane Zatezalo, Tito Ciuro, Jonah Williams of Carbon Five, Joshua Weinberg, biappi, Eric Mock, Jay Spencer, and everyone at the iPhone developer channels at irc.saurik.com and irc.freenode.net, among many others too numerous to name individually. Their techniques, suggestions, and feedback helped make this book possible. If I have overlooked anyone who helped contribute, please accept my apologies for the oversight.

Special thanks go out to my family and friends, who supported me through month after month of new beta releases and who patiently put up with my unexplained absences and frequent howls of despair. I appreciate you all hanging in there with me. And thanks to my children for their steadfastness, even as they learned that a hunched back and the sound of clicking keys is a pale substitute for a proper mother. My kids provided invaluable assistance over the last few months by testing applications, offering suggestions, and just being awesome people. I try to remind myself on a daily basis how lucky I am that these kids are part of my life.

About the Author

Erica Sadun is the bestselling author, coauthor, and contributor to several dozen books on programming, digital video and photography, and web design, including the widely popular *The iPhone Developer's Cookbook: Building Applications with the iPhone 3.0 SDK, Second Edition*. She currently blogs at TUAW.com, and has blogged in the past at O'Reilly's Mac DevCenter, Lifehacker, and Ars Technica. In addition to being the author of dozens of iOS-native applications, Erica holds a Ph.D. in Computer Science from Georgia Tech's Graphics, Visualization and Usability Center. A geek, a programmer, and an author, she's never met a gadget she didn't love. When not writing, she and her geek husband parent three geeks-in-training, who regard their parents with restrained bemusement, when they're not busy rewiring the house or plotting global dominance.

18

iCloud Basics

Ready to emerge from the confines of a single device or desktop system? Welcome to iCloud. iCloud lets you share data beyond the limitations of individual hardware and into a central data-storage hub. With iCloud, your applications can access and update shared documents without explicit synchronization. Changes on one unit propagate out to all other registered systems, updating when that hardware next accesses the Internet. This completely passive coordination is yours for free via iCloud. All you need to do is set up some entitlements and incorporate its features in your applications. It's surprisingly easy to use. This chapter introduces iCloud, shows you how it works, introduces the key classes you'll use, and walks you through several recipes and code snippets that show you how to use it in your own projects.

Setting Up an iCloud-Compatible Project

To get started on your first iCloud project, you add a simple entitlement. Open the Project > TARGETS > Entitlements pane and check the Enable Entitlements box shown in Figure 18-1. This option builds a new entitlement file, adding a trio of entitlement types. These types include iCloud containers (the shared folders where you can add and read data), iCloud Key-Value Store (enabling a shared dictionary look-up system), and Keychain Access Groups (allowing secure-item sharing between applications).

Although Apple briefly flirted with enabling iCloud access on an application-by-application basis during the iOS 5 beta period, it eventually decided to enable iCloud for all applications without special provisioning. Any application identifier added to the provisioning portal is automatically enabled with iCloud access. This decision means that when you check the Entitlements box, you've done everything needed to get started with iCloud.

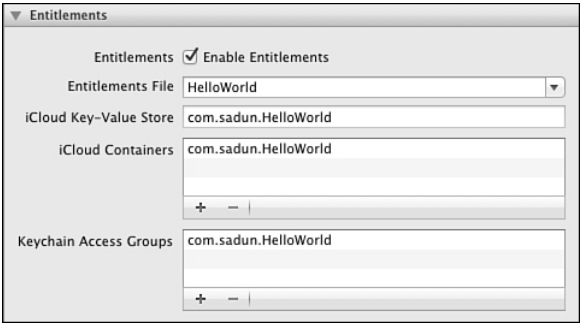

Figure 18-1 Enable iCloud functionality by establishing entitlements for your application. You can use the + button to add additional containers to your application. The first-listed container, which you can see defaults to the application identifier, acts as the default ubiquitous store.

Note

At the time this book was written, you cannot use iCloud on the iOS Simulator. You will want to have at least two iOS devices on-hand to work with, enabling you to test both data propagation and conflict resolution.

Determining Your Team Identifier Prefix

Every cloud application uses a special internal prefix—and it's not the one you're probably thinking of, the one associated with application identifiers at the provisioning portal. iCloud service uses your developer identifier (the "Team Identifier Prefix") to enable you to write to and access cloud-based data. This identifier is associated with your account and listed in the iOS Member Center in the Your Account page as your organization or individual ID.

You can look this identifier up by visiting developer.apple.com/ios, and clicking Member Center. Sign in with your developer credentials and view the Your Account tab. The Program Membership section lists the identifier between the account name and account type entries.

Outside of visiting the developer site, the easiest way to locate the prefix you're looking for is to take a peek into your Team Provisioning Profile. Use the Organizer > Devices > Provisioning Profiles pane to locate the provision file. Right-click iOS Team Provisioning Profile and choose Reveal Profile in Finder. Open this file in TextEdit and search for the `TeamIdentifier` key. It should appear exactly once in the file. Directly below the key, you will find a single-value, which is your team identifier prefix.

```
<key>TeamIdentifier</key>
<array>
    <string>YOURIDENTIFIER</string>
</array>
```

Containers

This team identifier prefix is used for all shared iCloud data containers. You can see this prefix in action by examining the default ubiquity folder for your application. Consider the following call. When passed `nil` as an argument, it looks up the default ubiquitous (i.e., iCloud) container URL. The default container is always the first item listed in the `com.apple.developer.ubiquity-container-identifiers` entitlement.

```
NSURL *ubiquity = [[NSFileManager defaultManager]
    URLForUbiquityContainerIdentifier:nil];
NSLog(@"Ubiquity: %@", ubiquity);
```

When executed, this code prints out the path to that container. The path includes a tilda-escaped version of the application identifier, complete with a team identifier prefix. The identifier ensures that containers are unique to each developer.

```
2011-08-29 12:17:41.094 HelloWorld[8069:707] Ubiquity:
file://localhost/private/var/mobile/Library/Mobile%20Documents/\
YOURTEAMIDENTIFIER~com~sadun~HelloWorld/
```

The default value added to the initial entitlement includes just one container, associated with the application identifier of the current project.

```
<key>com.apple.developer.ubiquity-container-identifiers</key>
<array>
    <string>$(TeamIdentifierPrefix)com.sadun.HelloWorld</string>
</array>
```

You can extend this entitlement to add more identifiers, so long as those items are associated with your team prefix, e.g.

```
<key>com.apple.developer.ubiquity-container-identifiers</key>
<array>
    <string>$(TeamIdentifierPrefix)com.sadun.HelloWorld</string>
    <string>$(TeamIdentifierPrefix)com.sadun.AnotherApp</string>
    <string>$(TeamIdentifierPrefix)com.sadun.SharedStorage</string>
</array>
```

Adding more identifiers enables you to add cross-application document sharing for any application built by the same developer account. Adding, for example, `com.sadun.AnotherApp` to these entitlements allows the `HelloWorld` application to access the `AnotherApp` application's default container.

The reverse domain-name identifiers do *not* have to be tied to a specific application. Although the entitlement domains default to the application name (`com.sadun.HelloWorld`), you can add shared items or replace the default container with a shared name (e.g., `com.sadun.SharedStorage`).

Retrieving Identifiers in Code

Your team prefix plays an important role in iCloud development. Outside of the default container (the first entitlement item), which you can access by passing `nil` to the

ubiquity URL query, you cannot access other containers without your team identifier prefix. You can hard code the prefix as a string constant, or you can retrieve the prefix with a simple method call. This method extracts the prefix for you:

```
+ (NSString *) teamPrefix
{
    NSURL *ubiquity = [[NSFileManager defaultManager]
        URLForUbiquityContainerIdentifier:nil];
    if (!ubiquity) return nil;
    NSArray *elements = [[ubiquity.path lastPathComponent]
        componentsSeparatedByString:@"~"];
    if (!elements.count) return nil;
    return [elements objectAtIndex:0];
}
```

Once extracted, you can create a valid container identifier by combining the prefix to a container name string (e.g., `MYPREFIX.com.sadun.SharedStorage`).

```
+ (NSString *) containerize: (NSString *) anIdentifier
{
    NSString *prefix = [self teamPrefix];
    if (!prefix) return nil;
    return [NSString stringWithFormat:@"%@.%@", prefix, anIdentifier];
}
```

Calls to `URLForUbiquityContainerIdentifier:` should not return `nil`. If they do, go through the following checklist:

- Did you add an entitlement for the container identifier?
- Did you use the proper team id prefix in the identifier?
- Did you enable iCloud on the device?

Making sure that you can retrieve a ubiquity container URL forms the first step for iCloud development.

Note

The two methods you saw in this section, `teamPrefix` and `containerize:`, are both part of a `CloudHelper` class, which was developed as part of this chapter. This class contains a variety of helper routines, mostly implemented at the class level, that allow you to more easily work with iCloud data in your applications.

How iCloud Works

In its simplest form, iCloud is merely a special folder. It lives on the iOS device in `/private/var/mobile/Library/Mobile Documents/`. You have partial permission to read from and write to this folder. That permission is limited to the container folder or folders declared in your application entitlements. iOS monitors and regularly updates this folder, synchronizing its contents with offsite storage. When you write to this folder, even just by dropping in an image or text file, it automatically joins the iCloud.

Your applications read and write local content from the folder. They do not need to directly access the Internet or check for connectivity. That is all handled for you by iCloud. All you have to deal with is local file URLs.

Updates arrive asynchronously at the device, where they can be handled immediately by open applications or addressed later by suspended ones. You're responsible for designing your application to handle situations where cloud access is restricted. Apps must implement conflict resolution strategies to deal with staggered updates from multiple parties to the same cloud account.

Imagine this scenario: Mom is flying to Detroit for a meeting, so she's offline for a while. Dad is at the gym, and Junior is checking in from the local library. Each of them adds new items to a shared To Do list application and may mark some off as “done.” That application must be able to handle these chronologically displaced updates in some smart manner, preferably involving users in its trickier decisions.

The `UIDocument` class, and its CoreData-enabled child `UIManagedDocument`, transform this otherwise complicated scenario into a simple notification. Applications are notified when a file’s state changes, allowing you to detect and resolve conflict situations as Dad and Junior's live updates overlap with each other, and when Mom disembarks from the plane and her changes have to somehow meld with the ones created while she was offline. You might automatically merge all added items and globally apply deletions, or you might lend weight to newer versions, or you might ask the current user what to do. (If you use Core Data with the `UIManagedDocument` subclass, much of this is automated for you.)

Think about that conflict resolution scenario. Most of your coding responsibilities involve wrestling with how to resolve these conflicts in each of your applications. The rest of iCloud is actually both simple and mechanical, as you'll see in the following sections.

iCloud Container Folders

Developing and using iCloud is primarily a matter of writing to, monitoring, and reading from the right folder at the right time. Under iOS 5 and later, `NSFileManager` creates authenticated on-demand URL access to that shared storage, which you can query at will. Your application *must* be properly entitled for iCloud access to happen. Consider the following call; you would substitute an actual container identifier you added to your entitlement:

```
NSString *container =
    [CloudHelper containerize:com.sadun.HelloWorld"];
NSURL *destinationURL = [[NSFileManager defaultManager]
    URLForUbiquityContainerIdentifier: container];
```

This call either returns a URL, in the case of proper entitlements, developer prefix, and container identifier or `nil`. The container the URL points to represents the base of where your application is allowed to store data. This is part of your application’s *ubiquitous data scope*.

Your data should actually live one step down from this container, in a `Documents` folder, much as it does in your Sandbox. This is part of your application's *ubiquitous documents scope*. Adding a documents folder allows your user to list and delete files individually from their Settings application. iCloud groups any items outside `Documents` into a single unit; these files must be deleted all at once. By creating a shared document home, you offer your user more nuanced access to his or her data.

You will encounter the terms *ubiquitous data scope* and *ubiquitous documents scope* throughout this chapter. Your application's ubiquitous data scope refers to all iCloud container content that lies outside `Documents` folders. The documents scope refers to iCloud content within `Documents` folders.

Building a Shared Document Home

After entitling your application, you'll be able to establish a home in which your app's iCloud documents will live. This involves creating a `Documents` subfolder in your container. The following method requests a URL that points to the container storage. If it does not find a subfolder there, it builds a new `Documents` folder. By adding it to the container, it mirrors the way your documents are stored in the application sandbox.

As you can see, this involves nothing more than using standard NSFileManager calls. The method checks to see if the folder exists. If it does not, it creates a new subfolder in the container.

```
// Return the iCloud Data URL
+ (NSURL *) ubiquityDataURLForContainer:
    (NSString *) container
{
    return [[NSFileManager defaultManager]
        URLForUbiquityContainerIdentifier:container];
}

// Return the iCloud Documents URL
+ (NSURL *) ubiquityDocumentsURLForContainer:
    (NSString *) container
{
    return [[self ubiquityDataURLForContainer:container]
        URLByAppendingPathComponent:@"Documents"];
}

// Prepare the Documents folder
+ (BOOL) setupUbiquityDocumentsFolderForContainer:
    (NSString *) container
{
    NSError *error;
    NSURL *targetURL =
        [self ubiquityDocumentsURLForContainer:container];

    // Create the ubiquity documents folder if needed
    if (![[NSFileManager defaultManager]
        fileExistsAtPath:targetURL.path])
    {
```

```
        if (![[NSFileManager defaultManager]
            createDirectoryAtPath:targetURL.path
            withIntermediateDirectories:YES
            attributes:nil error:&error])
        {
            NSLog(@"Error: %@", error.localizedFailureReason);
            return NO;
        }
    }
    return YES;
}
```

Writing To and Reading From Ubiquitous Storage

"Ubiquitous" items refer to those files that live in iCloud. Once you have a URL that points to your ubiquitous documents folder and not just the primary container, you may write to it and read from it just as you would in your sandbox. Be cautioned. This is generally a bad idea unless your files are entirely atomic, so you will not encounter any conflicts. Any file you create in that folder automatically updates its contents to iCloud.

Latency depends on network connectivity and device settings. Usually, small files update within a few seconds. You can monitor the connectivity by examining a device's console logs. Status change notifications indicate what percentage of the document has uploaded, ending with the item becoming current.

```
Jul  9 16:16:05 unknown librariand[50] <Notice>: STATUS CHANGE: url =
/private/var/mobile/Library/Mobile Docu-
ments/XYZZYPLUGH~com~sadun~CloudLearning/Documents/Output2.jpg status = 0xd02 (Has-
Revisions New Current InCloud) transferred = 0
Jul  9 16:16:05 unknown librariand[50] <Notice>: STATUS CHANGE: url =
/private/var/mobile/Library/Mobile Docu-
ments/XYZZYPLUGH~com~sadun~CloudLearning/Documents/Output2.jpg status = 0x40c02
(HasRevisions Current InCloud Uploading) transferred = 50
Jul  9 16:16:05 unknown librariand[50] <Notice>: <LBFileProvider: 0xe82c900>
itemIsNowCurrent: - url=/private/var/mobile/Library/Mobile Docu-
ments/XYZZYPLUGH~com~sadun~CloudLearning/Documents/Output2.jpg
```

Using the Organizer's device console logs allows you to monitor iCloud updates throughout your application runs. As there are no official notifications that you can subscribe to, I recommend leaving the console open as you develop and debug your iCloud apps. This allows you to better see what is going on behind the scenes with the iCloud daemon, providing live feedback of those changes.

Editing iCloud Storage

Determine which application files exist in your iCloud storage by visiting Settings > iCloud > Storage & Backup > Manage Storage > Documents and Data. There, listed (currently) under Unknown, you'll find any application-generated items. Figure 18-2 shows a single third-party text file created by my test application. Hopefully, Apple will update this in future releases to reflect the document source rather than "Unknown."

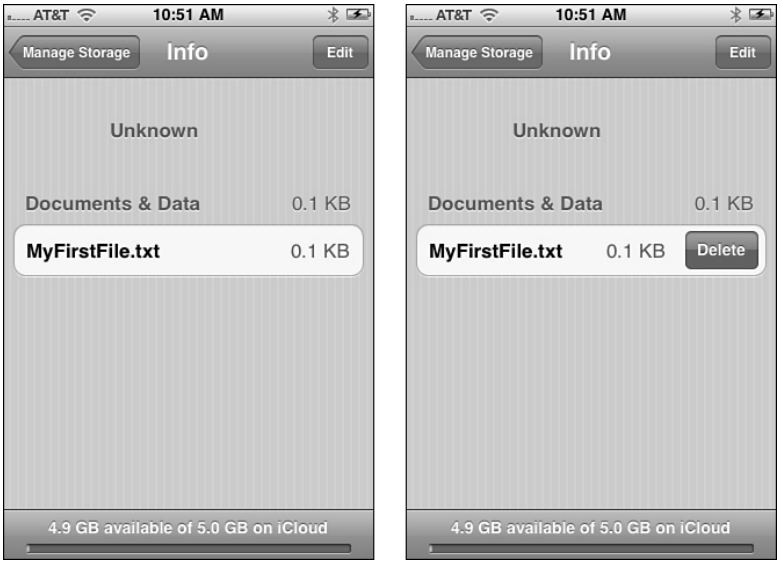

Figure 18-2 Application-generated iCloud documents appear in the Documents & Data section in Settings.

You can delete cloud files here by swiping-to-delete or tapping Edit. Removing a file here removes it from the cloud folder on your device and from the iCloud central storage. This is especially important for testing, because it allows you to reset across all devices.

Designing for iCloud Ubiquity

There are many issues to take into account when building applications for iCloud. As a rule, design your non-Core Data applications around `UIDocument` and your Core Data applications around `UIManagedDocument`. These classes were built for iCloud from the ground up and help simplify many of the synchronization issues that arise from using ubiquitous data.

When using iCloud, let your users choose which items to save to iCloud. Documents can live just as easily in your sandbox as they can in the cloud. Never force your users to pay extra money to Apple to gain ubiquitous access for items they don't want or need.

Be clever when letting users freely move their data into and out of the cloud. Removing data from the cloud can mean two things. The user may want to delete the file entirely, or they may want to move the data from the cloud to the device. Whether deleting or transferring, moving a file from the cloud removes that data from all other devices. Design for both scenarios.

When you need a file to survive its deletion from the cloud on a device that did not initiate its removal, you can take advantage of a user's `Library/Caches` folder.

Proactively caching copies of the data whenever the device is used allows you to store local versions so users don't entirely lose access to formerly ubiquitous data. The caches folder isn't backed up either by iTunes or iCloud, so you don't place as big a burden on a user's paid iCloud quota. At the same time, local caching does affect available space for the device, and you should allow the cached copy to be removed when the user no longer wants access to the data.

In an ideal world, Apple would introduce an optional iCloud *pending-deletion* state in addition to deleting cloud items immediately or transferring them from the cloud to the sandbox. Such an option would allow applications to check items out of the cloud on each registered device; the file would be removed once all registered devices were offered the opportunity to copy the cloud-stored version or when the user forced a full delete request.

You can create a rough approximation of this behavior by adding an "automatically check out copies of this document" feature to your applications. Copy the document from cloud to cache, and then, once the application detects that the document is no longer available from iCloud, move it from cache to sandbox for local use.

As a final note, remember that file names are case-sensitive on iOS, but are not on OS X. Take care when creating iCloud file names on iOS so you do not establish pairs of names that differ only by case. Add simple in-app checks to prevent users from doing so.

Note

Do not use the cloud to store sensitive information, like passwords. Use the device's keychain, and do not attempt to make that data ubiquitous. Familiarize yourself with Apple's iCloud security policies that explain this stance in further depth.

Recipe: Trying Out iCloud

The best way to convince yourself that you have properly entitled and established iCloud is to create a file from code and assure yourself that it propagates across to all devices. That's exactly what Recipe 18-1 does. It establishes the documents folder discussed earlier, writes a small text file ("Hello World") to the cloud, and builds a custom URL that you can use to access that new file from outside the cloud.

Requesting this URL requires net access and can take several seconds to retrieve. Please be patient during this time. The URL remains valid for a limited time, so the code also prints out the expiration date for that access period to the console.

After running this code, you'll want to ensure that cloud files reached their destination on each test device you have signed into the same account. Try downloading the file from the URL and, only after that, delete it using iCloud settings.

Recipe 18-1 **Giving iCloud a Spin**

```
+ (NSArray *) contentsOfUbiquityDocumentsFolderForContainer:
    (NSString *) container
```

```
{
    NSURL *targetURL =
        [self ubiquityDocumentsURLForContainer:container];
    if (!targetURL) return nil;

    NSArray *array = [[NSFileManager defaultManager]
        contentsOfDirectoryAtPath:targetURL.path error:nil];
    return array;
}

+ (NSArray *) contentsOfUbiquityDataFolder
{
    return [self contentsOfUbiquityDataFolderForContainer:nil];
}

+ (NSURL *) ubiquityDocumentsFileURL: (NSString *) filename
    forContainer: (NSString *) container
{
    if (!filename) return nil;
    NSURL *fileURL =
        [[self ubiquityDocumentsURLForContainer:container]
            URLByAppendingPathComponent:filename];
    return fileURL;
}

+ (NSURL *) ubiquityDocumentsFileURL: (NSString *) filename
{
    return [self ubiquityDocumentsFileURL:filename
        forContainer:nil];
}

// Log the contents of the ubiquity documents folder
- (void) list: (id) sender
{
    NSLog(@"Contents of Documents: %@",
        [CloudHelper contentsOfUbiquityDocumentsFolder]);
}

// Create a new text file
- (void) create: (id) sender
{
    // Write to default container
    NSError *error;
    NSURL *targetURL = [CloudHelper
        ubiquityDocumentsFileURL:@"MyFirstFile.txt"];
```

```
    // Write a "Hello World" Text file to the cloud
    NSLog(@"About to write to file.");
    if (![@"Hello from the cloud!" writeToURL:targetURL
        atomically:YES encoding:NSUTF8StringEncoding error:nil])
    {
        NSLog(@"Error writing to %@: %@",
            targetURL, error.localizedFailureReason);
        return;
    }

    // Retrieve a URL to share
    NSDate __autoreleasing *date;
    NSURL *url = [[NSFileManager defaultManager]
        URLForPublishingUbiquitousItemAtURL:targetURL
        expirationDate:&date error:&error];
    if (!url)
        NSLog(@"Error creating publishing URL: %@",
            error.localizedFailureReason);
    else
    {
        NSLog(@"iCloud URL: %@", url);
        NSLog(@"Expires: %@", date);
    }
}

- (void) loadView
{
    [super loadView];
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"List", @selector(list:));
    self.navigationItem.leftBarButtonItem =
        BARBUTTON(@"Create", @selector(create:));

    // Default Ubiquity Container
    NSLog(@"Data: %@", [CloudHelper ubiquityDataURL]);

    // Shared Ubiquity Container
    NSString *sharedIdentifier =
        [CloudHelper containerize:@"com.sadun.SharedStorage"];
    NSLog(@"Shared: %@", [CloudHelper
        ubiquityDataURLForContainer:sharedIdentifier]);

    // Nonexistent Ubiquity Container. Will error.
    NSString *nonexistentIdentifier =
        [CloudHelper containerize:@"com.sadun.nonexistent"];
    NSLog(@"Nonexistent: %@", [CloudHelper
```

```
        ubiquityDataURLForContainer:nonexistentIdentifier]);

    BOOL success = [CloudHelper setupUbiquityDocumentsFolder];
    if (success)
        NSLog(@"Default ubiquity Documents folder is ready");
    else
    {
        NSLog(@"Error setting up ubiquitous documents folder");
        return;
    }
}
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from this book, go to the folder for Chapter 18 and open the project for this recipe.

Working with UIDocument

The `UIDocument` class is an abstract class that supports cloud-ready asynchronous reading and writing of data. `UIDocument` simplifies the way you work with iCloud data, because it handles nearly all the issues you encounter about coordinating local data with cloud data. Recipe 18-1's "just write to the right folder" approach cannot be recommended for serious development. What `UIDocument` does is add a shell for managing your document, letting you send read, save, and write requests through it in a safer and more structured way.

You never work directly with the `UIDocument` class. Instead, you subclass it and implement several simple key features that link the class to the ubiquitous data.

Establishing the Document as a File Presenter

`UIDocument` implements the `NSFilePresenter` class. This means that you can register the class to receive updates whenever its cloud data gets updated by telling `NSFileCoordinator` to add the document as "presenter." This is terribly unhelpful naming, I know, but what it means is that a presenter class is one that takes a strong interest in knowing when outside changes happen to a given file. Receiving alerts about these changes allows it to update its GUI presentation in response.

The registration process works like this. You create a document and a coordinator, initializing the coordinator with the document as its performer as the following code snippet shows. Registering the new document/presenter with `NSFileCoordinator` class allows it to receive updates. You must perform both these steps, both at the instance and class level.

```
imageDocument = [[ImageDocument alloc] initWithFileURL:ubiquityURL];
imageDocument.delegate = self;
coordinator = [[NSFileCoordinator alloc]
    initWithFilePresenter:imageDocument];
[NSFileCoordinator addFilePresenter:imageDocument];
```

Although all the actual coordination implementation details are handled for you via the `UIDocument` class (for example, `UIDocument` handles all file coordination issues for you, so you do not need to manually coordinate reading and writing), this set up portion is not. Unless you establish the presenter/coordinator pair manually, your document *will not* receive updates from the cloud.

Moving to the Cloud

`NSFileManager` allows you to move local files to and from the cloud using the `setUbiquitous:itemAtURL:destinationURL:error:` method. What this method does is nothing more than move (not copy) the file in question from your sandbox into the central iCloud folder and back. However, it does so safely in an approved manner.

The method takes three arguments. The first establishes the direction of movement. `YES` brings items from the sandbox to the cloud and `NO` the reverse. The second argument must always be the source URL and the third its destination. The first argument is a little pointless in that the source URL will always be either sandbox or cloud and the destination must be the reverse. If all three arguments do not line up, the method will fail.

To move to the cloud, use `YES`, the local sandbox URL, and the destination cloud URL, as shown here. To remove an item from the cloud, use `NO`, then the cloud URL, and then the local sandbox URL.

```
if (![[NSFileManager defaultManager] setUbiquitous:YES
    itemAtURL:localURL destinationURL:ubiquityURL error:&error])
{
    NSLog(@"Error making local file ubiquitous. %@",
        [error localizedFailureReason]);
    return;
}
```

Retrieve the cloud URL by appending the file name to the results of the `ubiquity-DocumentsURL` method, which is shown in Recipe 18-1.

Evicting Items from the Cloud

Removing an item from the cloud by setting its ubiquity to `NO` deletes it from both the cloud and from all of your user's devices and computers. At times, you may want to remove a local copy in order to force it to refresh from the cloud without deleting it. To do this, do not alter the file's ubiquity. Instead, call an `NSFileManager` eviction method, as follows. Evicting a file allows iCloud to retrieve a new copy, but does not remove it from central cloud storage.

```
+ (BOOL) evictFile: (NSString *) filename
    forContainer: (NSString *) container
{
    if (!filename) return NO;

    NSURL *targetURL =
        [self ubiquityDocumentsFileURL:filename
            forContainer:container];
    BOOL targetExists =
        [[NSFileManager defaultManager]
            fileExistsAtPath:targetURL.path];
    if (!targetExists) return NO;

    NSError *error;
    if (![[NSFileManager defaultManager]
        evictUbiquitousItemAtURL:targetURL error:&error])
    {
        NSLog(@"Error evicting current copy of %@.: %@",
            filename, error.localizedFailureReason);
        return NO;
    }

    return YES;
}

+ (BOOL) evictFile: (NSString *) filename
{
    return [self evictFile:filename forContainer:nil];

}
```

In addition to file eviction, you can actually force iCloud to download a new copy. This is of use only when you write your own data handlers. You pretty much never have to do this if you use Apple's recommended `UIDocument` and `UIManagedDocument` classes.

```
+ (BOOL) forceDownload: (NSString *) filename
    forContainer: (NSString*) container
{
    if (!filename) return NO;
    NSURL *targetURL =
        [self ubiquityDocumentsFileURL:filename
            forContainer:container];
    if (!targetURL) return NO;

    NSError *error;
    if (![self evictFile:filename forContainer:container])
        return NO;
    if (![[NSFileManager defaultManager]
          startDownloadingUbiquitousItemAtURL:targetURL
          error:&error])
    {
        NSLog(@"Error starting download of %@: %@", filename,
              error.localizedFailureReason);
        return NO;
    }
```

```
    return YES;
}

+ (BOOL) forceDownload: (NSString *) filename
{
    return [self forceDownload:filename forContainer:nil];
}
```

Opening and Saving Files

You open and save `UIDocument` instances to read them in from a file and to store your changes out. Opening the file is generally performed as part of creating the document. You provide it with a file URL pointing to the stored file, and ask it to open the file. For iCloud materials, this path must point to the correct place in the ubiquitous documents folder to work with iCloud and not to a file saved locally. You can also use the `UIDocument` class to access and read non-iCloud files that have been moved back to the sandbox. This flexibility unifies how you access files, letting users choose which items to send to the cloud.

```
imageDocument = [[ImageDocument alloc] initWithFileURL:theURL];
[imageDocument openWithCompletionHandler:^(BOOL success) {
    NSLog(@"Open file was: %@", success ? @"successful" : @"failure");
    if (success) imageView.image = imageDocument.image;}];
```

Each time you make changes to your document, you'll want to save those updates as transparently as possible. Saving data, especially to small files, is cheap. Making updates as soon as your user creates changes allows you to build a persistent system that synchronizes with your user's adjustments.

```
imageDocument.image = image;
imageDocument saveToURL:imageDocument.fileURL
    forSaveOperation:UIDocumentSaveForOverwriting
    completionHandler:^(BOOL success){
      NSLog(@"Attempt to save to URL %@",
          success ? @"succeeded" : @"failed");
    }];
```

`UIDocument` is fully undo-aware and provides its own `undoManager` property hook. This allows you to save without having to worry about creating permanent changes from which you cannot recover.

Subscribing to UIDocument Notifications

When working with documents, your primary class must subscribe to a single notification: `UIDocumentStateChangedNotification`. Your handler uses that notification to handle conflict issues, allowing your document to update to the latest version. At its simplest, the notification subscription looks like this. This snippet resolves all conflict issues by accepting the most recent version and updating its presentation. This example, which works with the code in Recipe 18-2, is built around a simple image presentation app. When new images arrive, the application view updates and shows them.

```
 [[NSNotificationCenter defaultCenter]
    addObserverForName: UIDocumentStateChangedNotification
    object:nil queue:[NSOperationQueue mainQueue]
    usingBlock:^(NSNotification __strong *notification)
    {
        if (imageDocument.documentState == UIDocumentStateInConflict)
        {
            NSError *error;
            NSLog(@"Resolving conflict. (Newest wins.)");
            if (![NSFileVersion
                    removeOtherVersionsOfItemAtURL:
                        imageDocument.fileURL
                    error:&error])
        {
            NSLog(@"Error removing other document versions: %@",
                error.localizedFailureReason);
            return;
        }
        imageView.image = imageDocument.image;
    }
}];
```

This code, which uses "newest version wins," represents the simplest way you can handle document conflicts using `NSFileVersion`, a class built around time-based file revisions. Although it's the simplest approach, it's not always the best. You'll probably want to use a little more finesse in your applications. iOS can store more than one version of a file at a time. You can retrieve all conflicted versions of the file version class through something like the following. The snippet

```
[NSFileVersion unresolvedConflictVersionsOfItemAtURL:
    imageDocument.fileURL]
```

returns an array of file version instances. You can iterate through those instances to help resolve your conflicts, choosing which of the conflicted saves best represents the state that you want to present to the user.

When you need to examine the actual version data to better judge or merge that material, you can pull each file's versioned contents from the instance's `URL` property. From there, you can select which file version to use by removing the other versions (as in the previous example) or replace the file data entirely.

The document's `documentState` property lets you know whether your saves are currently conflicted or if you're clear to make changes. Additional states include `UIDocumentStateNormal`, which allows users to make unhindered changes; `UIDocumentStateClosed`, indicating that the document wasn't opened open properly (or has been deliberately closed) and is probably not valid for edits; `UIDocumentStateSavingError`, telling you that the file could not be saved out; and `UIDocumentStateEditingDisabled`, which means that the file is busy. A busy file is never safe for user edits, so your application should disable changes until the state changes back to something safer.

You may allow edits during both normal and conflict states but you should inform your user when conflict issues may prevent updates from propagating out to the cloud,

such as the Mom-on-a-Plane scenario discussed earlier. Apple recommends offering visual feedback, such as a green light for unhindered updates and a yellow one for modifications that may need later intervention to integrate.

Recipe: Subclassing UIDocument

At a minimum, each document instance inherits two core responsibilities: to produce its actual document data on demand and to update itself to new data in response to iCloud refreshes. When you subclass `UIDocument`, you must implement two methods. Recipe 18-2 presents the barest-bones document subclass that you might create:

- The `loadFromContents:error:` method brings new content to the class, allowing it to update its internal data. When it receives its new data, it pings an informal delegate to let it know that it's a good time to reload the image.
- The `contentsForType :error:` method gets called whenever the user saves data locally. Here, you produce a data representation of your document that can be stored to the cloud, allowing you to publish that data ubiquitously.

This recipe demonstrates the simplicity of creating a cloud-based document. With just these two methods, you can store nearly any kind of data ubiquitously.

Recipe 18-2 **Simple Cloud Document Class**

```
@interface ImageDocument : UIDocument
@property (strong) UIImage *image;
@property (weak) id delegate;
@end

#define SAFE_PERFORM_WITH_ARG(THE_OBJECT, THE_SELECTOR, THE_ARG) \
    (([THE_OBJECT respondsToSelector:THE_SELECTOR]) ? \
    [THE_OBJECT performSelector:THE_SELECTOR withObject:THE_ARG] : nil)

@implementation ImageDocument
@synthesize image, delegate;

- (BOOL) loadFromContents:(id)contents ofType:(NSString *)typeName
    error:(NSError *__autoreleasing *)outError
{
    NSLog(@"Loading external content");
    self.image = nil;
    if ([contents length] > 0)
        self.image = [[UIImage alloc] initWithData:contents];
    if (delegate)
        SAFE_PERFORM_WITH_ARG(delegate, @selector(imageUpdated:), self);
```

```
    return YES;
}

- (id) contentsForType:(NSString *)typeName
    error:(NSError *__autoreleasing *)outError
{
    NSLog(@"Publishing content");
    return UIImageJPEGRepresentation(self.image, 0.75f);
}
@end
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from this book, go to the folder for Chapter 18 and open the project for this recipe.

Metadata Queries and the Cloud

Two new metadata scopes, `NSMetadataQueryUbiquitousDataScope` and `NSMetadataQueryUbiquitousDocumentsScope`, point to the iCloud's Data and Documents folders. You can add an ongoing query to monitor these folders, creating a callback whenever the folder contents change. This allows you to know when new items appear or disappear and update your file listings accordingly. Listing 18-1 offers a pair of methods that demonstrate how to use this technology to query and monitor the folder:

- The first method, `startMonitoringUbiquitousDocumentsFolder`, sets up a monitor for the Documents folder scope, listening for update notifications. When these occur, the method uses an informal delegate callback to update a client.
- The second method, `stopMonitoringUbiquitousDocumentsFolder`, cancels the notification observation, allowing your code to stop any ongoing monitoring.

To search the ubiquitous container outside the Documents folder, use the `NSMetadataQueryUbiquitousDataScope` search scope instead of `NSMetadataQueryUbiquitousDocumentsScope`. You can add both scopes via `setSearchScopes:` or pass an empty array to remove search scope limitations. In iOS 5, you cannot yet use the `NSMetadataQuery` class to search sandboxes or other parts of iOS outside of authorized iCloud container storage.

The `startMonitoringUbiquitousDocumentsFolder` method, shown here, automatically extracts filenames (via the `NSMetadataItemFSNameKey` key) to an array and returns those to an informal delegate. You might want to adapt the method to return the raw `NSMetadataItem` objects or URLs (`NSMetadataItemURLKey`) instead.

Use file updates to refresh data sources but never depend on them to precisely indicate an update event. You may receive some callbacks that seem untied to anything other

than iCloud checking in. These callbacks are never excessive in terms of overwhelming your application, but "contents changed" doesn't *always* precisely mean that contents have actually changed. You will receive a callback when new items appear or are removed, but you may also receive callbacks when iCloud is just "being iCloud."

If you use more than one container in your application, make sure you adapt your callbacks to determine which container was updated. Listing 18-1 assumes you'll be working with a single ubiquitous Documents folder.

Listing 18-1 **Monitoring iCloud**

```
- (void) startMonitoringUbiquitousDocumentsFolder
{
    // Remove any existing query - stored in local instance variable
    if (query) [query stopQuery];

    // Search for all file names
    query = [[NSMetadataQuery alloc] init];
    query.predicate = [NSPredicate predicateWithFormat:
        @"NSMetadataItemFSNameKey == '*'"];
    query.searchScopes = [NSArray arrayWithObject:
        NSMetadataQueryUbiquitousDocumentsScope];

    // Subscribe to query updates
    [[NSNotificationCenter defaultCenter] addObserverForName:
        NSMetadataQueryDidUpdateNotification
     object:nil queue:[NSOperationQueue mainQueue]
      usingBlock:^(NSNotification __strong *notification)
      {
         NSLog(@"Contents changed in ubiquitous documents folder");

         // disable the query while iterating through its results
         [query disableUpdates];
         NSMutableArray *array = [NSMutableArray array];
         for (NSMetadataItem *item in query.results)
             [array addObject: [item
                 valueForAttribute:NSMetadataItemFSNameKey]];
         [query enableUpdates];

         // use an informal delegate callback with file names
         if (delegate) SAFE_PERFORM_WITH_ARG(delegate,
             @selector(folderContentsHaveChanged:),
             array);
     }];

    [query startQuery];
}
```

```
- (void) stopMonitoringUbiquitousDocumentsFolder
{
    [[NSNotificationCenter defaultCenter]
        removeObserver:self
        name:NSMetadataQueryDidFinishGatheringNotification
        object:nil];
    [query stopQuery];
    query = nil;
}
```

Handy Routines

This section presents a number of extremely simple routines that work together to allow you to manage files across your sandbox and cloud Documents folders. These methods assume your files live at the top level of these folders for several reasons:

- Both the container and sandbox Documents folders are made accessible to your users via Settings and iTunes. Therefore, they are directly visible.
- Neither system supports subfolders at this time.
- Your users own these folders directly, according to Apple's guidelines, and should have full access and visibility to all materials contained within.

Local URLs

Working with the sandbox Documents folder involves little more than querying for the documents directory. These methods return the documents folder and URLs that point inside that folder. This `localFileURL:` method does not test the URL it returns to see if it points to a valid object.

```
+ (NSString *) localDocumentsPath
{
    return [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
        NSUserDomainMask, YES) objectAtIndex:0];
}

+ (NSURL *) localDocumentsURL
{
    return [NSURL fileURLWithPath:[self localDocumentsPath]];
}

+ (NSURL *) localFileURL: (NSString *) filename
{
    if (!filename) return nil;
    NSURL *fileURL = [[self localDocumentsURL]
        URLByAppendingPathComponent:filename];
    return fileURL;
}
```

Locating the Ubiquitous Documents Folder

Supply a container identifier and `NSFileManager` returns URLs pointing to the root of that cloud folder. Appending Documents produces the ubiquitous documents URL.

```
+ (NSURL *) ubiquityDataURLForContainer: (NSString *) container
{
    return [[NSFileManager defaultManager]
        URLForUbiquityContainerIdentifier:container];
}

+ (NSURL *) ubiquityDocumentsURLForContainer: (NSString *) container
{
    return [[self ubiquityDataURLForContainer:container]
        URLByAppendingPathComponent:@"Documents"];
}
```

When you have that folder's URL, you can append file names to it to point to items within the Documents subfolder. Like the `localFileURL:` method, the following methods do not test the URL it returns to see if it points to a valid object:

```
+ (NSURL *) ubiquityDataFileURL: (NSString *) filename
    forContainer: (NSString *) container
{
    if (!filename) return nil;
    NSURL *fileURL =
        [[self ubiquityDataURLForContainer:container]
            URLByAppendingPathComponent:filename];
    return fileURL;
}

+ (NSURL *) ubiquityDocumentsFileURL: (NSString *) filename
    forContainer: (NSString *) container
{
    if (!filename) return nil;
    NSURL *fileURL =
        [[self ubiquityDocumentsURLForContainer:container]
            URLByAppendingPathComponent:filename];
    return fileURL;
}
```

Retrieving Files

The first three of these methods test whether a file exists, locally or in the data or documents ubiquity. The final method uses these tests to retrieve a file URL, searching first the local folder and then the cloud. If a matching file is found, it returns that URL.

```
+ (BOOL) isLocal: (NSString *) filename
{
    if (!filename) return NO;
    NSURL *targetURL = [self localFileURL:filename];
    if (!targetURL) return NO;
    return [[NSFileManager defaultManager]
        fileExistsAtPath:targetURL.path];
}
```

```
+ (BOOL) isUbiquitousData: (NSString *) filename
    forContainer: (NSString *) container
{
    if (!filename) return NO;
    NSURL *targetURL = [self ubiquityDataFileURL:filename
        forContainer:container];
    if (!targetURL) return NO;
    return [[NSFileManager defaultManager]
        fileExistsAtPath:targetURL.path];
}

+ (BOOL) isUbiquitousDocument: (NSString *) filename
    forContainer: (NSString *) container
{
    if (!filename) return NO;
    NSURL *targetURL = [self ubiquityDocumentsFileURL:filename
        forContainer:container];
    if (!targetURL) return NO;
    return [[NSFileManager defaultManager]
       fileExistsAtPath:targetURL.path];
}

+ (NSURL *) fileURL: (NSString *) filename
    forContainer:(NSString *)container
{
    if ([self isLocal:filename])
        return [self localFileURL:filename];
    if ([self isUbiquitousDocument:filename
        forContainer:container])
        return [self ubiquityDocumentsFileURL:filename
            forContainer:container];
    if ([self isUbiquitousData:filename
        forContainer:container])
        return [self ubiquityDataFileURL:filename
            forContainer:container];
    return nil;
}
```

Setting Ubiquity

The following method applies ubiquity to a file based on its name, not its current location. This routine checks for each possible file condition. If the file is already found where it's supposed to end up, the method returns with success. If the file cannot be found, it fails. Otherwise, the code moves the file to or from the cloud as requested, wrapping away the otherwise fussy implementation details of the native `NSFileManager` method.

```
+ (BOOL) setUbiquitous:(BOOL)yorn for:(NSString *)filename
    forContainer:(NSString *)container
{
    if (!filename) return NO;

    NSError *error;
```

```
    NSURL *localURL = [self localFileURL:filename];
    NSURL *ubiquityURL =
        [self ubiquityDocumentsFileURL:filename
            forContainer:container];

    BOOL localFound = [self isLocal:filename];
    BOOL ubiquityFound =
        [self isUbiquitousDocument:filename
            forContainer:container];

    // Check file not found
    if (!localFound && !ubiquityFound) return NO;

    // Check the two "nothing to be done" cases
    if (!yorn && localFound) return YES;
    if (yorn && ubiquityFound) return YES;

    // ubiquitous to local
    if (!yorn)
    {
        // Move file away from cloud
        if (![[NSFileManager defaultManager]
              setUbiquitous:NO
              itemAtURL:ubiquityURL
              destinationURL:localURL
              error:&error])
        {
            NSLog(@"Error removing %@ from %@ storage: %@",
                filename, container,
                error.localizedFailureReason);
            return NO;
        }

        return YES;
    }

    // local to ubiquitous
    if (![[NSFileManager defaultManager]
          setUbiquitous:YES
          itemAtURL:localURL
          destinationURL:ubiquityURL
          error:&error])
    {
        NSLog(@"Error moving %@ to %@ storage: %@",
            filename, container,
            error.localizedFailureReason);
        return NO;
    }

    return YES;
}
```

Deleting Files

These methods take three approaches to deleting files. The first deletes a local copy at the top of the Documents sandbox folder. The second moves items from the cloud (data and documents) and then deletes it locally. The third finds the file first and then invokes one of the previous solutions.

```
+ (BOOL) deleteLocal: (NSString *) filename
{
    NSURL *targetURL = [self localFileURL:filename];
    if (![[NSFileManager defaultManager]
        fileExistsAtPath:targetURL.path])
    {
        NSLog(@"Local file not found: %@", filename);
        return NO;
    }

    NSError *error;
    BOOL success = [[NSFileManager defaultManager]
        removeItemAtURL:targetURL error:&error];
    if (!success)
        NSLog(@"Error removing file %@: %@",
            filename, error.localizedFailureReason);

    return success;
}

+ (BOOL) deleteUbiquitousDocument:(NSString *)filename
    forContainer:(NSString *)container
{
    NSURL *targetURL =
        [self ubiquityDocumentsFileURL:filename
            forContainer:container];
    if (![[NSFileManager defaultManager]
        fileExistsAtPath:targetURL.path])
    {
        NSLog(@"Ubiquitous file not found: %@", filename);
        return NO;
    }

    // Remove from ubiquity and then delete
    BOOL success = [self setUbiquitous:NO
        for:filename forContainer:container];
    if (success)
        return [self deleteLocal:filename];
    return NO;
}

+ (BOOL) deleteUbiquitousData:(NSString *)filename
    forContainer:(NSString *)container
{
    NSError *error;
    BOOL success;

    NSURL *targetURL =
```

```
        [self ubiquityDataFileURL:filename
            forContainer:container];
    success = [[NSFileManager defaultManager]
        fileExistsAtPath:targetURL.path];
    if (!success)
    {
        NSLog(@"Ubiquitous file not found: %@", filename);
        return NO;
    }

    success = [[NSFileManager defaultManager]
        removeItemAtURL:targetURL error:&error];
    if (!success)
    {
        NSLog(@"Could not remove item at path: %@",
            error.localizedFailureReason);
        return NO;
    }

    return YES;
}

+ (BOOL) deleteDocument: (NSString *) filename
    forContainer:(NSString *)container
{
    // If local, delete it.
    if ([self isLocal:filename])
        return [self deleteLocal:filename];
    return [self deleteUbiquitousDocument:filename
        forContainer:container];
}
```

Retrieving Modification Dates

These methods retrieve the modification date for a file, in the top level of either Documents folder (sandbox or ubiquitous), depending on where the file is found.

```
+ (NSDate *) modificationDateForURL:(NSURL *) targetURL
{
    if (!targetURL) return nil;

    NSDictionary *attributes =
        [[NSFileManager defaultManager]
            attributesOfItemAtPath:targetURL.path error:nil];
    if (!attributes) return nil;

    return [attributes fileModificationDate];
}

+ (NSDate *) modificationDateForFile: (NSString *) filename
{
    if (!filename) return nil;
    return [self modificationDateForURL:
        [self fileURL:filename]];
}
```

```
+ (NSTimeInterval) timeIntervalSinceModification:
    (NSString *) filename
{
    return [[NSDate date] timeIntervalSinceDate:
        [self modificationDateForFile:filename]];
}
```

Recipe: Accessing the Ubiquitous Key-Value Store

iCloud's key-value store lets you share small amounts of state information between devices using a ubiquitous property list. "Small" means you are limited to a total of 256Kb per key-value store and 64Kb per individual value.

Those limits mean you won't be using this utility for sharing images and other large data items. They're meant to save state such as the last page read, the default account name, or other such tiny bits of information that can propagate between devices to enhance the seamless movement of the user from one device to the next.

Note

Apple's policy is that key-value stores are not charged against the user's 5GB iCloud quota. At the time this book was written, Apple was not actively enforcing the 64K quota mentioned for limited key-value stores. That quota will likely come online by the time this book is published.

How Key-Value Stores Work

Your application's ubiquitous key-value store is nothing more than a distributed defaults dictionary that can be reached from devices registered to the same iCloud account. This store isn't meant to replace `NSUserDefaults`; it augments it by allowing you to develop applications that enable users to resume work regardless of the device used.

Imagine your user begins editing a ubiquitous document on a certain device and later launches your app on a second one. The key value store would let you load the right document (by storing the file name) and scroll it to the current position (by storing the current offset), so the user resumes interaction at the point that he or she left off.

If you want to use more extensive shared defaults, you can create those defaults in a property list file *outside* the ubiquitous Documents folder. This has two advantages. First, it lets you store whatever data you need without regard to the key-value store's quota limitations. Second, it hides those defaults from the user. Data stored outside Documents cannot be directly seen or edited by the user but they remain visible to your application.

Accessing the Key-Value Store

You can retrieve the shared store from its class method, namely `[NSUbiquitousKeyValueStore defaultStore]`. Once retrieved, query it like you would user defaults, e.g.:

```
kv = [NSUbiquitousKeyValueStore defaultStore];
switchView.on = [kv boolForKey:@"switchIsOn"];
```

Similarly, set its value using the same dictionary-style method calls. As with user defaults, you must synchronize your changes to publish them from memory to the persistent iCloud store. Although the system *may* call synchronize automatically, make a habit of manually performing the synchronization. This ensures that your changes are pushed immediately rather than with a possible several-second delay.

```
- (void) toggleSwitch: (UISwitch *) aSwitch
{
    // Send switch update out to cloud
    [kv setBool:aSwitch.isOn forKey:@"switchIsOn"];
    [kv synchronize];
}
```

The previous sample code snippets implement a ubiquitous switch. When the user interacts with the switch on one device, the change propagates to all devices. In order for that change to register, your application must subscribe to a notification that informs it about key-value store updates.

Subscribing to Key-Value Notifications

Your application will want to know when the ubiquitous store has updated item values. Subscribe it to `NSUbiquitousKeyValueStoreDidChangeExternallyNotification`. This notification lets you determine when the store has updated and adjust your in-app settings to match.

The notification's user info dictionary provides a list of affected keys as well as a reason why the update occurred. `NSUbiquitousKeyValueStoreServerChange` means the values simply updated. In addition, your application may be notified of a quota violation (`NSUbiquitousKeyValueStoreQuotaViolationChange`) or that local changes were discarded because they could not be sent to the server for an initial sync (`NSUbiquitousKeyValueStoreInitialSyncChange`).

Recipe 18-3 builds callbacks based on these various scenarios, allowing a client class to add informal callbacks. If you don't care which key was updated here, just implement the `kvStoreUpdated:` callback, which is sent on any value change. The `kvStoreUpdatedForKeys:` callback sends an array of affected keys for more nuanced responses.

Recipe 18-3 Subscribing to Key-Value Update Notifications

```
[[NSNotificationCenter defaultCenter]
    addObserverForName:
```

```
        NSUbiquitousKeyValueStoreDidChangeExternallyNotification
    object:nil
    queue:[NSOperationQueue mainQueue]
    usingBlock:^(NSNotification __strong *notification) {
        NSDictionary *userInfo = [notification userInfo];

        NSUInteger reason = [[userInfo objectForKey:
            NSUbiquitousKeyValueStoreChangeReasonKey] intValue];
        NSArray *keys = [userInfo objectForKey:
        NSUbiquitousKeyValueStoreChangedKeysKey];

        // Perform updates only if there is a delegate to listen
        if (!delegate) return;

        if (reason == NSUbiquitousKeyValueStoreServerChange)
        {
             if (keys.count == 1)
                 SAFE_PERFORM_WITH_ARG(delegate,
                     @selector(kvStoreUpdatedForKey:),
                     [keys lastObject]);
             else if (keys.count)
                 SAFE_PERFORM_WITH_ARG(delegate,
                     @selector(kvStoreUpdatedForKeys:),
                     keys);

             SAFE_PERFORM_WITH_ARG(delegate,
                 @selector(kvStoreUpdated), nil);
         }
         else if (reason ==
             NSUbiquitousKeyValueStoreInitialSyncChange)
             SAFE_PERFORM_WITH_ARG(delegate,
                @selector(kvStorePerformedInitialSync), nil);
         else if (reason ==
             NSUbiquitousKeyValueStoreQuotaViolationChange)
             SAFE_PERFORM_WITH_ARG(delegate,
                @selector(kvStoreViolatedQuota), nil);
}];
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from this book, go to the folder for Chapter 18 and open the project for this recipe.

Recipe: UIManagedDocument and Core Data

The `UIManagedDocument` class allows you to work with ubiquitous Core Data stores with iCloud. This class integrates data with the cloud. It offers all the built-in services you need for a Core Data–compliant managed model object and extends those services to ubiquitous files. At the time this chapter was written, support for this new class was just coming online. Keep that preliminary nature in mind as you read this section as details have likely changed.

Although the class is derived from `UIDocument`, there are basic differences between the way you use `UIManagedDocument` and `UIDocument`:

- **You may want to use the data container, and not Documents**—That's because the default Core Data store created by `UIManagedDocument` (i.e., not a standard SQLite store) contains a rather complex file structure. Moving out of the Documents folder allows you to handle those items as a single unit rather than expose the individual files to the end user in his or her iCloud preferences.
- **Listen for persistent store content changes**—In addition to document state changes, your application needs to listen for persistent store updates. `NSPersistentStoreDidImportUbiquitousContentChangesNotification` indicates your application should merge iCloud changes into your managed context.
- **Merge external changes**—Use a method similar to the one discussed in this recipe to integrate inserted, modified, or deleted objects into your data store.
- **Do not save your changes**—Unlike `UIDocument`, where you can save both early and often, you only save a `UIManagedDocument` on creation—and you do so there in a rather tricky fashion, as you'll see in the steps that follow.

Keep these differences in mind as you move forward to the following how-to write-up.

Remove All Context Saves

The first iCloud refactoring step involves removing all context saves from your application. Any time you see code like this,

```
NSError *error = nil;
if (![context save:&error])
    NSLog(@"Error: %@", [error localizedDescription]);
```

go ahead and comment it out. `UIManagedDocument` handles all context saves for you.

Note

Context saves are a different matter from saving the managed document. With documents, you create or overwrite files using `saveToURL:forSaveOperation:completionHandler:`. The edits in this step only affect context saves.

Establish Your Identifiers and URLs

Your application should establish two key identifiers that you will use throughout your managed document implementation. They include

- A real-world ready name that your new file will use when it is saved to the sandbox (e.g., "ToDo")
- A unique name to use when saving the managed store to the cloud. Use a standard reverse-domain style for this private identifier, according to Apple standards.

Here's how those identifiers might look in your source file:

```
#define SharedFileName   @"ToDo"
#define PrivateName      @"com.sadun.CloudLearning.storage"
```

Next, create two key URLs that build on these identifiers. The first is a local URL. It points to the sandbox and uses the real-world file name. The second is a cloud URL. It points to the data storage area of your ubiquitous container and uses the private reverse domain identifier.

```
NSURL *localURL = [CloudHelper localFileURL:SharedFileName];
NSURL *cloudURL = [CloudHelper ubiquityDataFileURL:PrivateName];
```

The local and cloud URLs work as a pair. The local item points to a sandbox shell. The managed document links that shell to the persistent store, which is located in the ubiquitous container at the cloud URL.

Establish the Document

Create a new managed document object by allocating it and pointing it to the sandbox URL. Your `UIManagedDocument` instance should always point to the sandbox item and not to the ubiquitous persistent store. Here's how you can set up the object with the local URL:

```
// Establish the document by pointing to the local sandbox
document = [[UIManagedDocument alloc] initWithFileURL:localURL];
```

The document's persistent store options declare the content name and include a pointer to the cloud URL. They also should establish that the store migrates its data automatically and use an inferred mapping model. Set the options like this:

```
// Set the persistent store options to point to the cloud
NSDictionary *options = [NSDictionary dictionaryWithObjectsAndKeys:
    PrivateName, NSPersistentStoreUbiquitousContentNameKey,
    cloudURL, NSPersistentStoreUbiquitousContentURLKey,
    [NSNumber numberWithBool:YES],
        NSMigratePersistentStoresAutomaticallyOption,
    [NSNumber numberWithBool:YES],
        NSInferMappingModelAutomaticallyOption,
    nil];
document.persistentStoreOptions = options;
```

Once you're done setting the store options, you're ready to read in the file if it exists or create it. Allocating a document and setting its URL does not open the file; it merely creates an instance of the class. You must use that instance to save or open the file.

Opening the Document

Here's a basic rule of thumb: When the sandbox item exists, you can assume that the ubiquitous store it links to exists as well. The converse is not true. A ubiquitous store created on another device might not have a local shell component. In that case, you can create one.

Test for the sandbox file using `NSFileManager` and the local URL. If the file exists, open it and perform a first fetch on its data. You're then ready to start operating as you normally do.

```
if ([helper isLocal:SharedFileName])
{
    NSLog(@"Attempting to open existing file");
    [document openWithCompletionHandler:^(BOOL success){
        if (!success) {NSLog(@"Error opening file"); return;}
        NSLog(@"File opened");
        [self performFetch];;
    }];
}
```

When the file is not available, create it in the sandbox. Do this regardless of whether the ubiquitous file exists or not. If it exists, the sandbox shell links to it. If it does not, both "files" are created at the same time.

This snippet creates the file, closes it, and then opens it up again. This is probably overkill, but I found that it works consistently in my testing and, when skipping these steps, it does not. Your mileage is sure to vary, especially after the beta period for this new release is over and this book goes live. When the ubiquitous portion does not yet exist, `UIManagedDocument` automatically creates the persistent store component in the location you specified in its options. In the end, the document exists in two places on each device: the store in the cloud and the wrapper in the sandbox.

```
{
    NSLog(@"Creating file.");
    // 1. save it out, 2. close it, 3. read it back in.
    // You probably can get away with doing less
    [document saveToURL:localURL
        forSaveOperation:UIDocumentSaveForCreating
        completionHandler:^(BOOL success){
            if (!success) { NSLog(@"Error creating file"); return; }
            NSLog(@"File created");
            [document closeWithCompletionHandler:^(BOOL success){
                NSLog(@"Closed new file: %@", success ?
                    @"Success" : @"Failure");
                [document openWithCompletionHandler:^(BOOL success){
                      if (!success) {
                          NSLog(@"Error opening file for reading.");
```

```
                        return;}
                    NSLog(@"File opened for reading.");
                    [self performFetch];;
            }];
        }];
    }];
}
```

Start Observing

Edits on each device are propagated out via iCloud updates. You need to listen for persistent store updates to respond to these changes, so subscribe in your setup to the following notification:

```
[[NSNotificationCenter defaultCenter]addObserver:self
    selector:@selector(documentContentsDidUpdate:)
    name:NSPersistentStoreDidImportUbiquitousContentChangesNotification
    object:nil];
```

Implement a cloud-updated method (the name is arbitrary, although `documentContentsDidUpdate:` is typical) to match the notification selector. The following method relays the notification data to an asynchronous method that merges the update with the current store:

```
 - (void) documentContentsDidUpdate: (NSNotification *) notification
{
    NSDictionary* userInfo = [notification userInfo];
    [context performBlock:^{
        [self mergeiCloudChanges:userInfo forContext:context];}];
}
```

From here, it's up to the `mergeiCloudChanges:forContext:` method, which you can read through in Recipe 18-4, to manage those merges. This code is again adapted from Apple sample code. It iterates through the updated, refreshed, and invalidated objects and applies those changes to the local managed context.

The merging rounds out the updates made to the standard Core Data application. All other elements of your app can remain as they were prior to iCloud integration.

Recipe 18-4 Ubiquitous Core Data

```
#pragma mark Initialize the Core Data Stores
- (void) initCoreData
{
    NSURL *localURL = [CloudHelper localFileURL:SharedFileName];
    NSURL *cloudURL = [CloudHelper ubiquityDataFileURL:PrivateName];

    // Create the document pointing to the local sandbox
    document = [[UIManagedDocument alloc] initWithFileURL:localURL];

    // Set the persistent store options to point to the cloud
    NSDictionary *options = [NSDictionary dictionaryWithObjectsAndKeys:
```

```
    PrivateName,
        NSPersistentStoreUbiquitousContentNameKey,
    cloudURL,
        NSPersistentStoreUbiquitousContentURLKey,
    [NSNumber numberWithBool:YES],
        NSMigratePersistentStoresAutomaticallyOption,
    [NSNumber numberWithBool:YES],
        NSInferMappingModelAutomaticallyOption,
nil];
document.persistentStoreOptions = options;
context = document.managedObjectContext;

// Register as presenter
coordinator = [[NSFileCoordinator alloc]
    initWithFilePresenter:document];
[NSFileCoordinator addFilePresenter:document];

// Check at the local sandbox
if ([helper isLocal:SharedFileName])
{
    NSLog(@"Attempting to open existing file");
    [document openWithCompletionHandler:^(BOOL success){
        if (!success) {NSLog(@"Error opening file"); return;}
        NSLog(@"File opened");

        [self performFetch];;
    }];
}
else
{
    NSLog(@"Creating file.");
    // 1. save it out, 2. close it, 3. read it back in.
    // You probably can get away with doing less
    [document saveToURL:localURL
       forSaveOperation:UIDocumentSaveForCreating
      completionHandler:^(BOOL success){
          if (!success) { NSLog(@"Error creating file"); return; }
          NSLog(@"File created");
          [document closeWithCompletionHandler:^(BOOL success){
              NSLog(@"Closed new file: %@", success ?
                  @"Success" : @"Failure");
          [document openWithCompletionHandler:^(BOOL success){
                  if (!success) {
                      NSLog(@"Error opening file for reading.");
                      return;}
                  NSLog(@"File opened for reading.");
```

```
                    [self performFetch];;
                }];
            }];
        }];
    }

    // Register to be notified of changes to the persistent store
    [[NSNotificationCenter defaultCenter]addObserver:self
        selector:@selector(documentStateChanged:)
        name: NSPersistentStoreDidImportUbiquitousContentChangesNotification
        object:nil];
}

#pragma mark Courtesy of Apple. Thank you Apple
// Merge the iCloud changes into the managed context
- (void)mergeiCloudChanges: (NSDictionary*)userInfo
    forContext: (NSManagedObjectContext*)managedObjectContext
{
    @autoreleasepool
    {
        NSMutableDictionary *localUserInfo =
           [NSMutableDictionary dictionary];

        // Handle the invalidations
        NSSet* allInvalidations =
            [userInfo objectForKey:NSInvalidatedAllObjectsKey];
        NSString* materializeKeys[] = { NSDeletedObjectsKey,
                NSInsertedObjectsKey };

        if (nil == allInvalidations)
        {
            int c = (sizeof(materializeKeys) / sizeof(NSString*));
            for (int i = 0; i < c; i++)
            {
                NSSet* set = [userInfo objectForKey:materializeKeys[i]];
                if ([set count] > 0)
                {
                    NSMutableSet* objectSet = [NSMutableSet set];
                    for (NSManagedObjectID* moid in set)
                        [objectSet addObject:[managedObjectContext
                            objectWithID:moid]];
                   [localUserInfo setObject:objectSet
                        forKey:materializeKeys[i]];
                }
            }
```

```
            // Handle the updated and refreshed Items
            NSString* noMaterializeKeys[] = { NSUpdatedObjectsKey,
                NSRefreshedObjectsKey, NSInvalidatedObjectsKey };
            c = (sizeof(noMaterializeKeys) / sizeof(NSString*));

            for (int i = 0; i < 2; i++)
            {
                NSSet* set = [userInfo objectForKey:noMaterializeKeys[i]];
                if ([set count] > 0)
                {
                    NSMutableSet* objectSet = [NSMutableSet set];
                    for (NSManagedObjectID* moid in set)
                    {
                        NSManagedObject* realObj =
                           [managedObjectContext
                               objectRegisteredForID:moid];
                        if (realObj)
                            [objectSet addObject:realObj];
                    }
                    [localUserInfo setObject:objectSet
                        forKey:noMaterializeKeys[i]];
                }
            }

            // Fake a save to merge the changes
            NSNotification *fakeSave = [NSNotification
               notificationWithName:
                   NSManagedObjectContextDidSaveNotification
               object:self  userInfo:localUserInfo];
            [managedObjectContext
                mergeChangesFromContextDidSaveNotification:fakeSave];
        }
        else
            [localUserInfo setObject:allInvalidations
                forKey:NSInvalidatedAllObjectsKey];

        [managedObjectContext processPendingChanges];
        [self performSelectorOnMainThread:@selector(performFetch)
            withObject:nil waitUntilDone:NO];
    }
}

// When notified about a cloud update, start merging changes
- (void) documentContentsDidUpdate: (NSNotification *) notification
{
    NSLog(@"Cloud has been updated.");
```

```
    NSDictionary* userInfo = [notification userInfo];
    [context performBlock:^{
        [self mergeiCloudChanges:userInfo forContext:context];}];
}
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from this book, go to the folder for Chapter 18 and open the project for this recipe.

Summary

This chapter introduced iCloud and its ability to manage ubiquitous data. In it, you discovered the basics of how to integrate its features in your own applications. You saw how to create new provisions and entitlements, learned what's going on under the hood, and discovered how to integrate iCloud access into your applications. Here are a few final thoughts to take with you:

- Once you hand over control to a `UIDocument` subclass, there's little you need to do other than to save your changes and update your undo manager on a regular basis.
- While accepting the most recently saved item is a really easy way to go with conflict resolution, it's not necessarily the best way. Role-play how your application might be used in the real world, with all kinds of possible conflicts, and design your resolution strategy around that.
- If you are working with Core Data, use `UIManagedDocument` and its built-in Core Data integration rather than subclassing `UIDocument`.
- `UIManagedDocument` takes a simple "last-changed wins" philosophy to life. Once you update your code with the elements shown in Recipe 18-4, you can keep reusing those elements across all your projects. Unlike `UIDocument`-based apps, which require more nuanced thinking, the managed document's conflict resolution strategy can be used over and over again. That makes Core Data and `UIManagedDocument` a powerfully simple solution for iCloud integration.
- Details matter. When it comes to iCloud, you must punch every point in the process. You can use a simple Recipe 18-1 "Hello World"–style test to ensure that you're correctly accessing the cloud with your entitlements before moving on to more complicated matters. Don't forget to register your document as a file presenter.
- The reason you use your developer ID with your entitlements is to allow access between all your applications. You can create a suite of applications that all work together, allowing file transfer using a shared iCloud storage container.

- Use key-value stores to provide a continuous user experience, so your user can pick up on a new device where he or she left off on another. KV stores are meant to enhance, not replace, user defaults.
- iCloud shares data between devices for a single account, and for a single developer's applications. Multiple user accounts cannot share data into a central source and separate developers cannot create shared folders. If you need this functionality, look to providers outside iCloud or build your own Web service.
- Unlike iOS, Mac OS X HFS+ file names are generally case insensitive. Creating container IDs and file names that differ only by case will produce unhappy development outcomes. Design your iOS apps to be Mac-safe from the start and vice versa.
- Users pay for cloud storage. So, use it wisely or not at all. A little state data can go a long way, and a lot of media storage will get you nowhere fast.