iCloud Design Guide

Developer

Contents

Figures, Tables, and Listings

About Incorporating iCloud Into Your App

iCloud is a free service that lets users access their personal content on all their devices—wirelessly and automatically via Apple ID. iCloud does this by combining network-based storage with dedicated APIs, supported by full integration with the operating system. Apple provides server infrastructure, backup, and user accounts, so you can focus on building great iCloud-enabled apps.

The core idea behind iCloud is to eliminate explicit synchronization between devices. A user never needs to think about syncing and your app never interacts directly with iCloud servers. When you adopt iCloud storage APIs as described in this document, changes appear automatically on all the devices attached to an iCloud account. Your users get safe, consistent, and transparent access to their personal content everywhere.

At a Glance

iCloud is all about content, so your integration effort focuses on the model layer of your app. Because instances of your app running on a user's other devices can change the local app instance's data model, you design your app to handle such changes. You might also need to modify the user interface for presenting iCloud-based files and information.

There is one important case for which Cocoa adopts iCloud for you. A document-based app for OS X v10.8 or later requires very little iCloud adoption work, thanks to the capabilities of the `NSDocument` class.

There are many different ways you can use iCloud storage, and a variety of technologies available to access it. This document introduces all the iCloud storage APIs and offers guidance in how to design your app in the context of iCloud.

iCloud Supports User Workflows

Adopting iCloud in your app lets your users begin a workflow on one device and finish it on another.

Say you provide a podcast app. A commuter subscribes to a podcast on his iPhone and listens to the first twenty minutes on his way to work. At the office, he launches your app on his iPad. The episode automatically downloads and the play head advances to the point he was listening to.

Or say you provide a drawing app for iOS and OS X. In the morning, an architect creates some sketches on her iPad while visiting a client. On returning to her studio, she launches your app on her iMac. All the new sketches are already there, waiting to be opened and worked on.

To store state information for the podcast app in iCloud, you'd use iCloud key-value storage. To store the architectural drawings in iCloud, you'd use iCloud document storage.

Relevant Chapter: "iCloud Fundamentals" (page 9)

Three Kinds of iCloud Storage

iCloud supports three kinds of storage. To pick the right one (or combination) for your app, make sure you understand the intent and capabilities of each. The three kinds of iCloud storage are:

- **Key-value storage** for discrete values, such as preferences, settings, and simple app state.
- **Document storage** for user-visible file-based information such as word processing documents, drawings, and complex app state.
- **Core Data storage** for shoebox-style apps and server-based, multi-device database solutions for structured content. iCloud Core Data storage is built on document storage and employs the same iCloud APIs.

Relevant Chapters: "Designing for Key-Value Data in iCloud" (page 24), "Designing for Documents in iCloud" (page 30), "Designing for Core Data in iCloud" (page 43)

Prepare for iCloud with Provisioning and Entitlements

The first two steps in adopting iCloud for your app are to obtain an appropriate provisioning profile for your development device and to request the appropriate entitlements in your Xcode project.

Note: iCloud entitlements are available only to apps submitted to the App Store or to the Mac App Store.

Entitlements are key-value pairs that request capabilities for your app—such as the capability to use iCloud. Your iCloud entitlement values define where your app can place data and they ensure that only your apps are allowed to access that data. You request separate entitlements for document storage and key-value storage. When you code sign your app, these requests become part of your app's code signature.

Relevant Sections: "First, Provision Your Development Devices" (page 9), "Request Access to iCloud Storage By Using Entitlements" (page 11)

How to Use This Document

Whether you are developing for iOS, OS X, or both, and no matter which sort of app you are developing, start by reading the entire "iCloud Fundamentals" (page 9) chapter to get the foundation that all iCloud developers need.

Next, read "Designing for Key-Value Data in iCloud" (page 24). Any app that provides user settings or maintains user state—that is, nearly every app—should adopt iCloud key-value storage.

The iOS and OS X document architectures automatically provide most of the iCloud functionality needed by document-based apps. If your app works with file-based information, you'll want to read "Designing for Documents in iCloud" (page 30).

If you are developing a Core Data app, read "Designing for Core Data in iCloud" (page 43) for an overview of iCloud considerations for Core Data.

No matter which iCloud storage APIs you adopt in your app, testing is critical. To get started on creating a test plan for your app, read "Testing and Debugging Your iCloud App" (page 48).

See Also

This document describes the pieces you need to support iCloud in your app, but does not teach you how to develop apps. For that, start with *Start Developing iOS Apps Today* or *Start Developing Mac Apps Today*, and read the following documents:

- iOS apps: *iOS App Programming Guide*
- Mac apps: *Mac App Programming Guide*

If you plan to use Core Data with iCloud, learn about this technology in *Core Data Programming Guide* and be sure to read *Using Core Data with iCloud Programming Notes*.

For a tutorial introduction to implementing a document-based iCloud app for iOS, read *Your Third iOS App: iCloud*.

iCloud Fundamentals

From a user's perspective, iCloud is a simple feature that automatically makes their personal content available on all their devices. To make your app participate in this "magic," you need to design and implement your app somewhat differently, and for this you need to learn about your app's roles when it participates with iCloud.

These roles, and the specifics of your iCloud adoption process, depend on your app. You design how your app manages its data, so only you can decide which iCloud supporting technologies your app needs and which ones it does not.

This chapter gets you started with the fundamental elements of iCloud that all developers need to know.

First, Provision Your Development Devices

To start developing an iCloud app, you must have an appropriate device provisioning profile and app ID. If you don't already have these in place, learn about setting up a provisioning profile and app ID in "Provisioning Your App for Store Technologies" in *App Distribution Guide*.

iCloud Data Transfer Proceeds Automatically and Securely

When you adopt iCloud, the operating system initiates and manages uploading and downloading of data for the devices attached to an iCloud account. Your app does not directly communicate with iCloud servers and, in most cases, does not invoke upload or download of data. At a very high level, the process works as follows:

1. You configure your app to gain access to special local file system locations known as *ubiquity containers*.
2. You design your app to respond appropriately to changes in the availability of iCloud (such as if a user signs out of iCloud), and to changes in the locations of files (because instances of your app on other devices can rename, move, duplicate, or delete files).
3. Your app reads and writes to its ubiquity containers using APIs that provide file coordination, as explained in "How iCloud Document Storage Works" (page 31).
4. The operating system automatically transfers data to and from iCloud as needed.

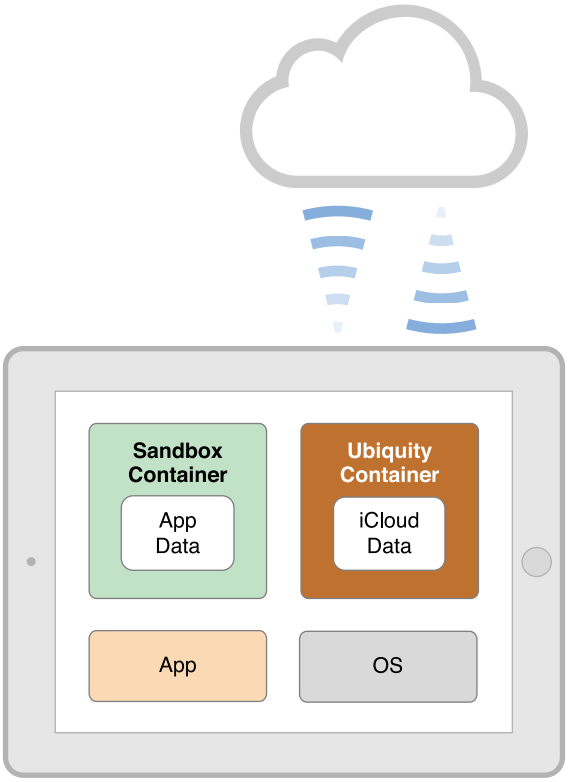
In iOS, there is an exception to automatic iCloud data transfer. For the first-time download of an iCloud-based document in iOS, your app actively requests the document. You learn about this process in "How iCloud Document Storage Works" (page 31).

iCloud secures user data with encryption in transit and on the iCloud servers, and by using secure tokens for authentication. For details, refer to iCloud security and privacy overview. Key-value storage employs the same security as iCloud uses for "Documents in the Cloud," as it is described in that document.

The Ubiquity Container, iCloud Storage, and Entitlements

To save data to iCloud, your app places the data in special file system locations known as *ubiquity containers*. A **ubiquity container** serves as the local representation of corresponding iCloud storage. It is outside of your app's sandbox container, as shown in Figure 1-1.

Figure 1-1 Your app's main ubiquity container in context



To enable your app to access ubiquity containers, you request the appropriate iCloud entitlements.

Request Access to iCloud Storage By Using Entitlements

Entitlements are key-value pairs that request specific capabilities or security permissions for your app. When the system grants your app iCloud entitlements, your app gains access to its ubiquity containers and to its iCloud key-value storage.

To use iCloud entitlements in an app, first enable entitlements in the Xcode target editor. Xcode responds by creating a `.entitlements` property list file which you can see in the Xcode Project navigator.

You then request the specific iCloud entitlements you need for your app, associating them with a bundle identifier. (See "CFBundleIdentifier" in *Information Property List Key Reference*.)

There are two types of iCloud entitlement, corresponding to the two types of iCloud storage:

- **iCloud key-value storage**. To enable key-value storage (appropriate for preferences or small amounts of data), select the iCloud Key-Value Store checkbox in the Xcode target editor. This requests the `com.apple.developer.ubiquity-kvstore-identifier` entitlement. Xcode responds by filling in the value for the entitlement with your app's bundle identifier, which represents the iCloud key-value storage location for your app.

  If you look in the `.entitlements` file, you see the `com.apple.developer.ubiquity-kvstore-identifier` entitlement key along with a fully-qualified value, which includes your unique team identifier string. The entitlement value follows this pattern:

  ```
  <TEAM_ID>.<BUNDLE_IDENTIFIER>
  ```

- **iCloud document storage**. To enable document storage (appropriate for user-visible documents, Core Data storage, or other file-based data), click the '+' button below the iCloud Containers list in the Xcode target editor. Xcode responds by adding a first string to the list. The string identifies the primary ubiquity container for your app, and, by default, is based on your app's bundle identifier.

  If you look in the `.entitlements` file, you see the `com.apple.developer.ubiquity-container-identifiers` key along with the fully-qualified default ubiquity container identifier value.

The first ubiquity container identified in the iCloud Containers list is special in the following ways:

- It is your app's primary ubiquity container, and, in OS X, it is the container whose contents are displayed in the `NSDocument` app-specific open and save dialogs.
- Its identifier string must be the bundle identifier for the current target, or the bundle identifier for another app of yours that was previously submitted for distribution in the App Store and whose entitlements use the same team ID.

Important: Container identifier strings must not contain any wildcard ('*') characters.

This chapter provides a full discussion about how to choose between key-value and document storage, in "Choose the Proper iCloud Storage API" (page 21).

Configuring a Common Ubiquity Container for Multiple Apps

In the Xcode target editor's Summary tab, you can request access to as many ubiquity containers as you need for your app. For example, say you provide a free and paid version of your app. You'd want users, who upgrade, to retain access to their iCloud documents. Or, perhaps you provide two apps that interoperate and need access to each other's files. In both of these examples, you obtain the needed access by specifying a common ubiquity container and then requesting access to it from each app.

To configure a common ubiquity container

1. Pick one of your iCloud-enabled apps to serve as the primary app for the common ubiquity container.

   The app you pick can be the current one you are developing, or another app of yours submitted for distribution in the App Store and whose entitlements use the same team ID.

2. In the first row of the Xcode target editor's iCloud Containers list, enter the bundle identifier of your primary iCloud-enabled app.

   If you then look in the `.entitlements` property list file, you'll see that Xcode has automatically qualified the identifier string by prefixing it with your team ID.

To retrieve a URL for a ubiquity container, you must pass the fully qualified string to the `NSFileManager` method `URLForUbiquityContainerIdentifier:`. That is, you must pass the complete container identifier string, which includes your team ID, that you see in the `.entitlements` property list file. You can pass `nil` to this method to retrieve the URL for the first container in the list.

For more information about how to configure entitlements, see "Configuring Store Technologies in Xcode and iTunes Connect" in *App Distribution Guide*.

Configuring Common Key-Value Storage for Multiple Apps

If you provide a free and paid version of your app, and want to use the same key-value storage for both, you can do that. The procedure is similar to that for configuring a common ubiquity container.

To configure common key-value storage for multiple apps

1. Pick one of your iCloud-enabled apps to serve as the primary app for the common key-value storage.

The app you pick can be the current one you are developing, or another app of yours submitted for distribution in the App Store and whose entitlements use the same team ID.

2. In the Xcode target editor's iCloud Key-Value Store field, enter the bundle identifier of your primary iCloud-enabled app.

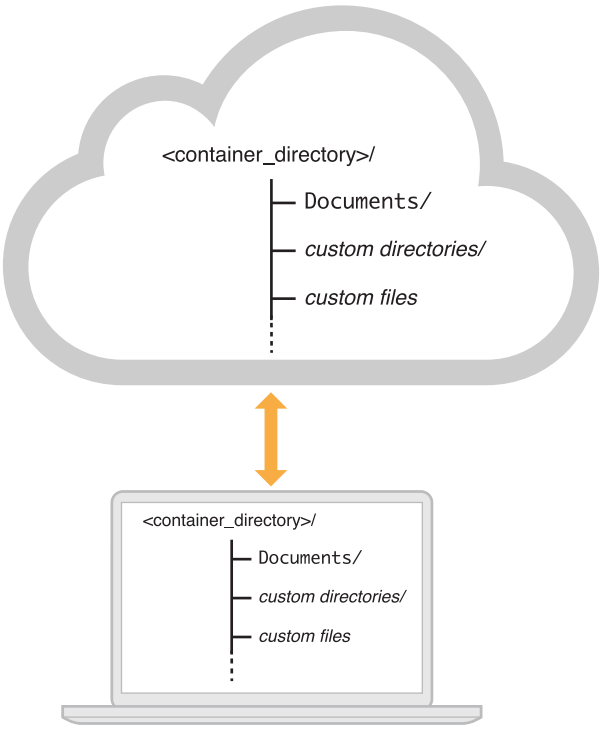
   If you then look in the `.entitlements` property list file, you'll see that Xcode has automatically qualified the identifier string by prefixing it with your team ID.

3. Ensure that, in the Xcode projects for each of your other apps that are to use the common key-value store, the iCloud Key-Value Store field contains the bundle identifier of your specified, primary iCloud-enabled app.

Ubiquity Containers Have Minimal Structure

As you can see in Figure 1-2, the structure of a newly-created ubiquity container is minimal—having only a `Documents` subdirectory. This allows you to define the structure as needed for your app, such as by adding custom directories and custom files at the top level of the container, as indicated in Figure 1-2.

Figure 1-2 The structure of a ubiquity container



You can write files and create subdirectories within the `Documents` subdirectory. You can create files or additional subdirectories in any directory you create. Perform all such operations using an `NSFileManager` object using file coordination. See "The Role of File Coordination" in *File System Programming Guide*.

The `Documents` subdirectory is the public face of a ubiquity container. When a user examines the iCloud storage for your app (using Settings in iOS or System Preferences in OS X), files or file packages in the `Documents` subdirectory are listed and can be deleted individually. Files outside of the `Documents` subdirectory are treated as private to your app. If a user wants to delete anything outside of the `Documents` subdirectories of your ubiquity containers, they must delete *everything* outside of those subdirectories.

To see the user's view of iCloud storage, do the following, first ensuring that you have at least one iCloud-enabled app installed:

- In iOS, open Settings. Then navigate to iCloud > Storage & Backup > Manage Storage.
- In OS X, open System Preferences. Then choose the iCloud preference pane and click Manage.

A User's iCloud Storage is Limited

Each iCloud user receives an allotment of complimentary storage space and can purchase more as needed. Because this space is shared by a user's iCloud-enabled iOS and Mac apps, a user with many apps can run out of space. For this reason, to be a good iCloud citizen, it's important that your app saves to iCloud only what is needed in iCloud. Specifically:

- **DO** store the following in iCloud:
  - User documents
  - App-specific files containing user-created data
  - Preferences and app state (using key-value storage, which does not count against a user's iCloud storage allotment)
  - Change log files for a SQLite database (a SQLite database's store file must never be stored in iCloud)
- **DO NOT** store the following in iCloud:
  - Cache files
  - Temporary files
  - App support files that your app creates and can recreate
  - Large downloaded data files

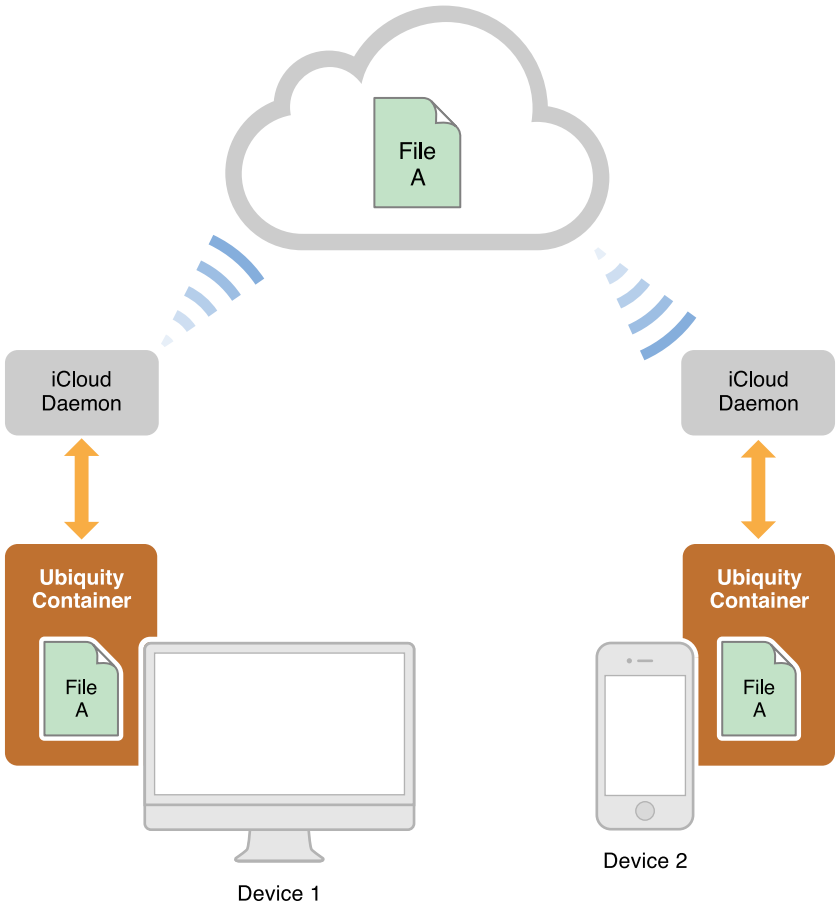
There may be times when a user wants to delete content from iCloud. Provide UI to help your users understand that deleting a document from iCloud removes it from the user's iCloud account *and* from all of their iCloud-enabled devices. Provide users with the opportunity to confirm or cancel deletion.

The System Manages Local iCloud Storage

A user's iCloud data lives on Apple's iCloud servers, and a cache lives locally on each of the user's devices, as shown in Figure 1-3.

Local caching of iCloud data allows a user to continue working even when the network is unavailable, such as when they turn on Airplane mode.

Figure 1-3 iCloud files are cached on local devices and stored in iCloud



Because the local cache of iCloud data shares space with the other files on a device, in some cases there is not sufficient local storage available for all of a user's iCloud data. The system addresses this issue, automatically, by maintaining an optimized subset of files locally. At the same time, the system keeps all metadata local, thereby ensuring that your app's users can access all their files, local or not. For example, the system might

evict a file from its ubiquity container if that file is not being used and local space is needed for another file that the user wants now; but updated metadata for the evicted file remains local. The user can still see the name and other information for the evicted file, and, if connected to the network, can open it.

Your App Can Help Manage Local Storage in Some Cases

Apps usually do not need to manage the local availability of iCloud files and should let the system handle eviction of files. There are two exceptions:

- If a user file is a) not currently needed and b) unlikely to be needed soon, you can help the system by explicitly evicting that file from the ubiquity container by calling the `NSFileManager` method `evictUbiquitousItemAtURL:error:`.

  > **Note:** Use this method with caution, keeping user needs in mind. Once a file is evicted, the system must download the file again before your app can use it.

- Conversely, if you explicitly want to ensure that a file is available locally, you can initiate a download to a ubiquity container by calling the `NSFileManager` method `startDownloadingUbiquitousItemAtURL:error:`. You learn more about this process in "App Responsibilities for Using iCloud Documents" (page 34).

Prepare Your App to Use iCloud

When a user launches your iCloud-enabled app for the first time, invite them to use iCloud. The choice should be all-or-none. In particular, it is best practice to:

- Use iCloud exclusively or use local storage exclusively; in other words, do not attempt to mirror documents between your ubiquity container and your sandbox container.
- Never prompt the user again about whether they want to use iCloud vs. local storage, unless they delete and reinstall your app.

Early in your app launch process—in the `application:didFinishLaunchingWithOptions:` method (iOS) or `applicationDidFinishLaunching:` method (OS X), check for iCloud availability by calling the `NSFileManager` method `ubiquityIdentityToken`, as shown in Listing 1-1.

Listing 1-1 Obtaining the iCloud token

```
id currentiCloudToken = [[NSFileManager defaultManager] ubiquityIdentityToken];
```

Call this lightweight, synchronous method from your app's main thread. The return value is a unique token representing the user's currently active iCloud account. You can compare tokens to detect if the current account is different from the previously-used one, as explained in the next section. To enable comparisons, archive the newly-acquired token in the user defaults database, using code like that shown in Listing 1-2.

Listing 1-2 Archiving iCloud availability in the user defaults database

```
 if (currentiCloudToken) {
    NSData *newTokenData =
            [NSKeyedArchiver archivedDataWithRootObject: currentiCloudToken];
    [[NSUserDefaults standardUserDefaults]
            setObject: newTokenData
               forKey: @"com.apple.MyAppName.UbiquityIdentityToken"];
} else {
    [[NSUserDefaults standardUserDefaults]
            removeObjectForKey: @"com.apple.MyAppName.UbiquityIdentityToken"];
}
```

This code takes advantage of the fact that the `ubiquityIdentityToken` method conforms to the `NSCoding` protocol.

If a signed-in user enables Airplane mode, iCloud is of course inaccessible but the account remains signed in; the `ubiquityIdentityToken` method continues to return the token for the account.

If a user signs out of iCloud, such as by turning off Documents & Data in Settings, the `ubiquityIdentityToken` method returns `nil`. To enable your app to detect when a user signs out and signs back in, register for changes in iCloud account availability. In your app's launch sequence, add an app object as an observer of the `NSUbiquityIdentityDidChangeNotification` notification, using code such as that shown in Listing 1-3.

Listing 1-3 Registering for iCloud availability change notifications

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

After obtaining and archiving the iCloud token, and registering for the iCloud notification, your app is ready to invite the user to use iCloud. If this is the user's first launch of your app with an iCloud account available, display an alert by using code like that shown in Listing 1-4.

Listing 1-4 Inviting the user to use iCloud

```
if (currentiCloudToken && firstLaunchWithiCloudAvailable) {
    UIAlertView *alert = [[UIAlertView alloc]
                          initWithTitle: @"Choose Storage Option"
                                message: @"Should documents be stored in iCloud
 and
                                                available on all your devices?"
                               delegate: self
                      cancelButtonTitle: @"Local Only"
                      otherButtonTitles: @"Use iCloud", nil];
    [alert show];
}
```

(This code is simplified to focus on the sort of language you would display. In an app you intend to provide to customers, you would internationalize this code by using the `NSLocalizedString` (or similar) macro, rather than using strings directly.)

Store the user's choice as a Boolean value in the user defaults database, from within a call to the `alertView:didDismissWithButtonIndex:` method. To do this, you can use code similar to that shown in Listing 1-2 (page 17). This stored value lets you determine, each time your app launches, a value for the `firstLaunchWithiCloudAvailable` variable in the `if` statement's condition (in Listing 1-4).

Although the `ubiquityIdentityToken` method tells you if a user is signed in to an iCloud account, it does not prepare iCloud for use by your app.

In iOS, make your ubiquity containers available by calling the `NSFileManager` method `URLForUbiquityContainerIdentifier:` for each of your app's ubiquity containers.

In OS X v10.8 and later, the `NSDocument` class automatically calls the `ubiquityIdentityToken` method as needed.

Always call the `URLForUbiquityContainerIdentifier:` method from a background thread—not from your app's main thread. This method depends on local and remote services and, for this reason, does not always return immediately. For example, you can use code such as that shown in Listing 1-5.

Listing 1-5 Obtaining the URL to your ubiquity container

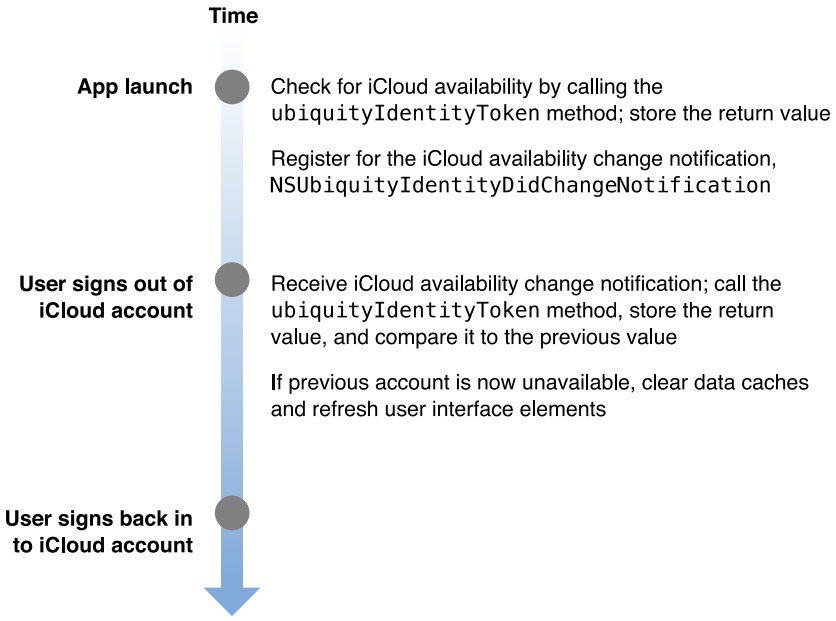
```
dispatch_async (dispatch_get_global_queue (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
^(void) {
    myUbiquityContainer = [[NSFileManager defaultManager]
            URLForUbiquityContainerIdentifier: nil];
    if (myUbiquityContainer != nil) {
        // Your app can write to the ubiquity container

    dispatch_async (dispatch_get_main_queue (), ^(void) {
        // On the main thread, update UI and state as appropriate
    });
    }
});
```

This example assumes that you have previously defined `myUbiquityContainer` as an `NSURL*` instance variable in the class containing this code.

Handle Changes in iCloud Availability

There are circumstances when iCloud is not available to your app, such as if a user disables the Documents & Data feature or signs out of iCloud. If the current iCloud account becomes unavailable while your app is running or in the background, your app must be prepared to remove references to user iCloud files and to reset or refresh user interface elements that show data from those files, as depicted in Figure 1-4.

Figure 1-4 Timeline for responding to changes in iCloud availability



To handle changes in iCloud availability, implement a method to be invoked on receiving an `NSUbiquityIdentityDidChangeNotification` notification. Your method needs to perform the following work:

- Call the `ubiquityIdentityToken` method and store its return value.
- Compare the new value to the previous value, to find out if the user signed out of their account or signed in to a different account.
- If the previously-used account is now unavailable, save the current state locally as needed, empty your iCloud-related data caches, and refresh all iCloud-related user interface elements.

  If you want to allow users to continue creating content with iCloud unavailable, store that content in your app's sandbox container. When the account is again available, move the new content to iCloud. It's usually best to do this without notifying the user or requiring any interaction from the user.

Note: If the `ubiquityIdentityToken` method and the `NSUbiquityIdentityDidChangeNotification` notification are not available based on your Xcode project settings, you need to play a more active role in keeping up with iCloud availability. For example, you would call the `URLForUbiquityContainerIdentifier:` method on a secondary thread, for each of your ubiquity containers, each time your app moves to the foreground.

Choose the Proper iCloud Storage API

iCloud provides two different iCloud storage APIs, each for a different purpose:

- **Key-value storage** is for discrete values such as preferences, settings, and simple app state.

  Use iCloud key-value storage for small amounts of data: stocks or weather information, locations, bookmarks, a recent documents list, settings and preferences, and simple game state. Every app submitted to the App Store or Mac App Store should take advantage of key-value storage.

- **Document storage** is for user-visible file-based content, Core Data storage, or for other complex file-based content.

  Use iCloud document storage for apps that work with presentations, word-processing documents, diagrams or drawings, or games that need to keep track of complex game state. If your document-based app needs to store state for each document, do so with each document and not by using key-value storage.

Many apps benefit from using both types of storage. For example, say you develop a task management app that lets a user apply keywords for organizing their tasks. You would employ iCloud document storage for the task items, and key-value storage for the list of user-entered keywords.

If your app uses Core Data, either for documents or for a shoebox-style app like iPhoto, use iCloud document storage. To learn how to adopt iCloud in your Core Data app, see "Designing for Core Data in iCloud" (page 43).

If your app needs to store passwords, do not use iCloud storage APIs for that. The correct API for storing and managing passwords is Keychain Services, as described in *Keychain Services Reference*.

Use Table 1-1 to help you pick the iCloud storage scheme that is right for each of your app's needs.

Table 1-1 Differences between document and key-value storage

Element	Document storage	Key-value storage
Purpose	User documents, complex private app data, and files containing complex app- or user-generated data.	Preferences and configuration data that can be expressed using simple data types.

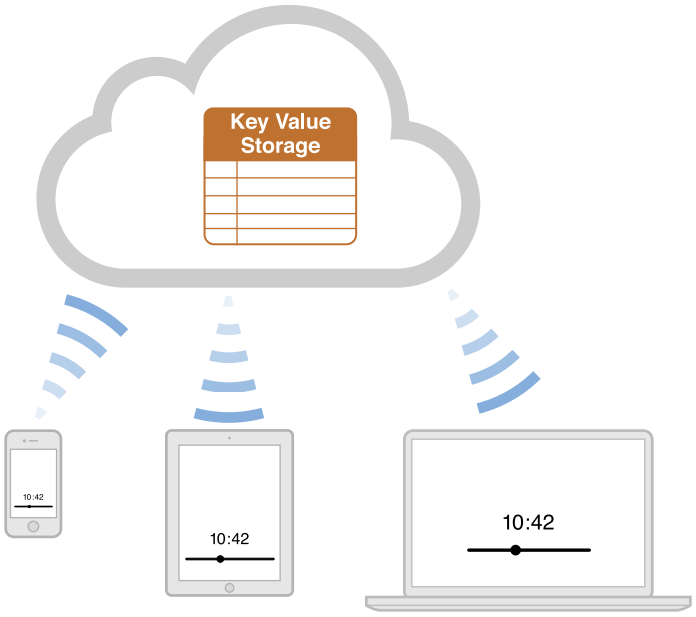
Element	Document storage	Key-value storage
Entitlement key	`com.apple.developer.ubiquity-container-identifiers`	`com.apple.developer.ubiquity-kvstore-identifier`
Data format	Files and file packages	Property-list data types only (numbers, strings, dates, and so on)
Capacity	Limited only by the space available in the user's iCloud account.	Limited to a total of 1 MB per app, with a per-key limit of 1 MB.
Detecting availability	Call the `URLForUbiquityContainerIdentifier:` method for one of your ubiquity containers. If the method returns `nil`, document storage is not available.	Always effectively available. If a device is not attached to an account, changes created on the device are pushed to iCloud as soon as the device is attached to the account.
Locating data	Use an `NSMetadataQuery` object to obtain live-updated information on available iCloud files.	Use the shared `NSUbiquitousKeyValueStore` object to retrieve values.
Managing data	Use the `NSFileManager` class to work directly with files and directories.	Use the default `NSUbiquitousKeyValueStore` object to manipulate values.
Resolving conflicts	Documents, as file presenters, automatically handle conflicts; in OS X, `NSDocument` presents versions to the user if necessary. For files, you manually resolve conflicts using file presenters.	The most recent value set for a key wins and is pushed to all devices attached to the same iCloud account. The timestamps provided by each device are used to compare modification times.
Data transfer	In iOS, perform a coordinated read to get a local copy of an iCloud file; data is then automatically pushed to iCloud in response to local file system changes. In OS X, iCloud files are always automatically pushed to iCloud in response to local file system changes.	Automatic, in response to local file system changes.
Metadata transfer	Automatic, in response to local file system changes.	Not applicable (key-value storage doesn't use metadata).

Element	Document storage	Key-value storage
User interface	None provided by iOS: Your app is responsible for displaying information about iCloud data, if desired; do so seamlessly and with minimal changes to your app's pre-iCloud UI. In OS X, `NSDocument` provides iCloud UI.	Not applicable. Users don't need to know that key-value data is stored in iCloud.

Designing for Key-Value Data in iCloud

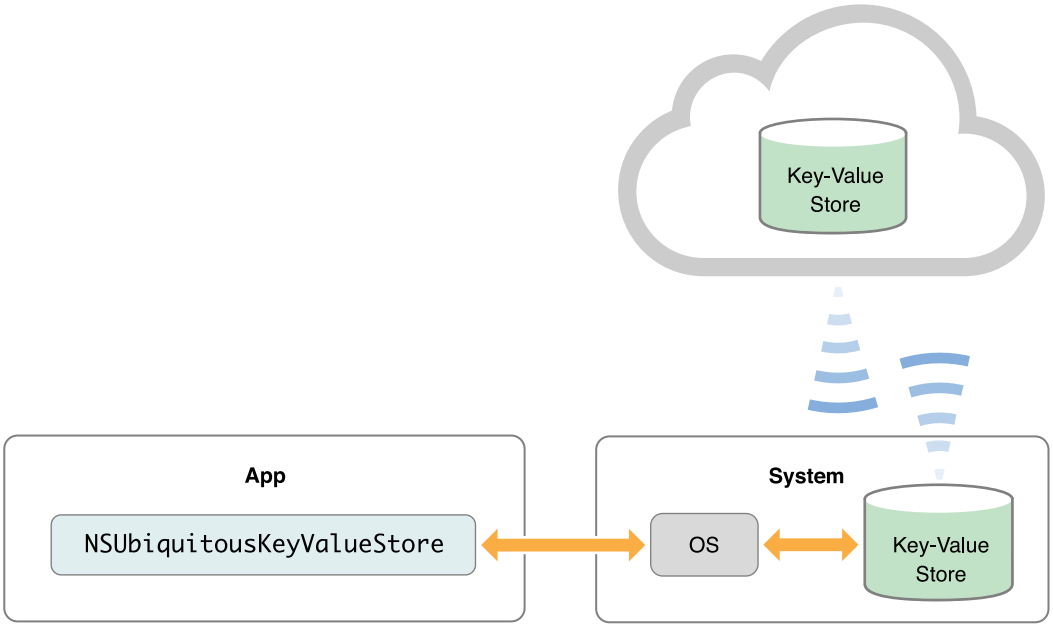
To store discrete values in iCloud for app preferences, app configuration, or app state, use iCloud key-value storage. **Key-value storage** is similar to the local user defaults database; but values that you place in key-value storage are available to every instance of your app on all of a user's various devices. If one instance of your app changes a value, the other instances see that change and can use it to update their configuration, as depicted in Figure 2-1.

Figure 2-1 iCloud key-value storage



Zooming in on the key-value storage interactions for a single device, as shown in Figure 2-2, you can see that your app accesses key-value storage by employing the shared `NSUbiquitousKeyValueStore` object.

Figure 2-2 iCloud key-value storage access



As you do with an `NSUserDefaults` object, use the iCloud key-value store to save and retrieve scalar values (such as `BOOL`) and property-list object types: `NSNumber`, `NSString`, `NSDate`, `NSData`, `NSArray`, and `NSDictionary`. Array and dictionary values can hold any of these value types.

The `NSUbiquitousKeyValueStore` class provides methods for reading and writing each of these types, as described in *NSUbiquitousKeyValueStore Class Reference*.

Each time you write key-value data, the operation succeeds or fails atomically; either all of the data is written or none of it is. You can take advantage of this behavior when your app needs to ensure that a set of values is saved together to ensure validity: place the mutually-dependent values within a dictionary and call the `setDictionary:forKey:` method.

Enable Key-Value Storage

Before you can use key-value storage, your app must have the appropriate entitlement, as explained in "Request Access to iCloud Storage By Using Entitlements" (page 11). And that is all the setup you need.

Prepare Your App to Use the iCloud Key-Value Store

Any device running your app, and attached to a user's iCloud account, can upload key-value changes to that account. To keep track of such changes, register for the `NSUbiquitousKeyValueStoreDidChangeExternallyNotification` notification during app launch. Then, to ensure your app starts off with the newest available data, obtain the keys and values from iCloud by calling the `synchronize` method. (You need never call the `synchronize` method again during your app's life cycle, unless your app design requires fast-as-possible upload to iCloud after you change a value.)

The following code snippet shows how to prepare your app to use the iCloud key-value store. Place code like this within your `application:didFinishLaunchingWithOptions:` method (iOS) or `applicationDidFinishLaunching:` method (OS X).

```
// register to observe notifications from the store
[[NSNotificationCenter defaultCenter]
    addObserver: self
       selector: @selector (storeDidChange:)
           name: NSUbiquitousKeyValueStoreDidChangeExternallyNotification
         object: [NSUbiquitousKeyValueStore defaultStore]];

// get changes that might have happened while this
// instance of your app wasn't running
[[NSUbiquitousKeyValueStore defaultStore] synchronize];
```

In your handler method for the `NSUbiquitousKeyValueStoreDidChangeExternallyNotification` notification, examine the user info dictionary and determine if you want to write the changes to your app's local user defaults database. It's important to decide deliberately whether or not to change your app's settings based on iCloud-originated changes, as explained next.

Actively Resolve Key-Value Conflicts

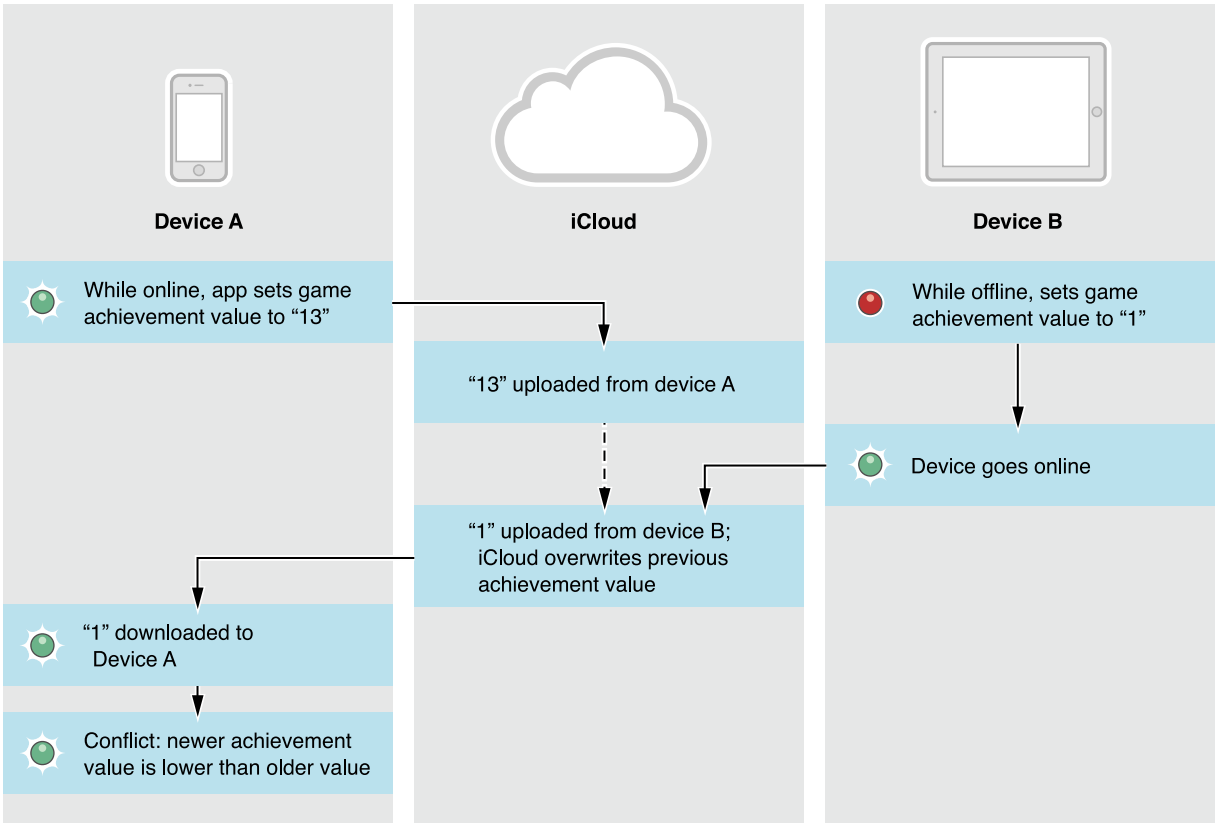
The implementation of iCloud key-value storage makes conflicts unlikely, but you must design your app to handle them should they occur. This section describes a key-value conflict scenario and explains how to handle it.

When a device attached to an iCloud account uploads a value to key-value storage, iCloud overwrites the corresponding, older value on the server. To determine which of two competing instances of a value is the newest, iCloud compares their universal-time timestamps as provided by the devices that uploaded the values. Usually, this simple model provides what a user wants.

However, it may be the case that an older value, not its replacement, newer one, is the "correct" value. It is your app's responsibility to Identify and resolve this situation. To allow your app to do that, use the `NSUbiquitousKeyValueStore` object in concert with—not as a replacement for—user defaults.

For example, say you develop a game that keeps track of the highest level achieved, and a user has reached level 13 on their iPhone—shown as the first step in the Device A column in Figure 2-3.

Figure 2-3 Creation of a key-value conflict



The user then opens your app for the first time on an iPad (shown as the first step in the Device B column in the figure) that is not currently connected to iCloud. So, the initial conditions shown in the figure are as follows:

- On device A, connected to iCloud (as indicated by the green light), a user has reached a high achievement level in your game.
- On device B, which has previously connected to the same iCloud account but is not now connected (indicated by the red light), the user plays your game for the first time and completes level one.

Continuing with this example, the user then connects the iPad to iCloud, shown as the "Device goes online" event in the Device B column in the figure; the device uploads the level-achievement value of 1.

iCloud sees that the *most recent* level-achievement value is 1, and, because it is newer, overwrites the previous value of 13. iCloud then pushes the value 1 to all connected devices. As shown in the final step in the Device A column of Figure 2-3 (page 27), this is certainly not what the user wants. And this is where your app design comes in.

When your app's state or configuration changes (such as, in this example, when the user reaches a new level in the game), first write the value to your user defaults database using the `NSUserDefaults` class. Then compare that local value to the value in key-value storage. If the values differ, your app needs to resolve the conflict by picking the appropriate winner.

In this example, you have stored the user's true highest level reached—13—in user defaults on Device A; so your app has the information it needs to do what the user expects. When the instance of your app on Device A receives an `NSUbiquitousKeyValueStoreDidChangeExternallyNotification` notification, compare the (lower) achievement value that it sees as newly-downloaded from iCloud, with the (higher) value in user defaults. Recognizing that in a level-based game, a user wants to keep building on their highest level achieved, write the true highest level (13 in this example) to key-value storage. iCloud then overwrites the value of 1 with 13 on the iCloud server, and then pushes that (true) highest achievement level to Device B. The iPad's app instance should then recognize the pushed value of 13 as appropriate, and should overwrite the local user defaults achievement value of "1" with "13."

Data Size Limits for Key-Value Storage

The total space available in your app's iCloud key-value storage is 1 MB per user. The maximum number of keys you can specify is 1024, and the size limit for each value associated with a key is 1 MB. For example, if you store a single large value of exactly 1 MB for a single key, that fully consumes your quota for a given user of your app. If you store 1 KB of data for each key, you can use 1,000 key-value pairs.

The maximum length for a key string is 64 bytes using UTF8 encoding. The data size of your cumulative key strings does *not* count against your 1 MB total quota for iCloud key-value storage; rather, your key strings (which at maximum consume 64 KB) count against a user's total iCloud allotment.

If your app has exceeded its quota in key-value storage, the iCloud key-value store posts the `NSUbiquitousKeyValueStoreDidChangeExternallyNotification` notification with a value of `NSUbiquitousKeyValueStoreQuotaViolationChange` in its user info dictionary.

For more on how to use iCloud key-value storage in your app, see "Storing Preferences in iCloud" in *Preferences and Settings Programming Guide*, and refer to *NSUbiquitousKeyValueStore Class Reference*.

Exercise Care When Using NSData Objects as Values

Using an `NSData` object lets you store arbitrary data as a single value in key-value storage. For example, in a game app, you can use it to upload complex game state to iCloud, as long as it fits within the 1 MB quota.

However, a data object in iCloud is available to be read, modified, and written by every instance of your app on a device attached to a user's account. Some of these instances could be older versions or running on another platform. Because of this, your data format requires the same careful design and care in handling as you would apply when using document storage, as explained in "Design for Robustness and Cross-Platform Compatibility" (page 39).

In addition, the larger your data objects, the more data gets uploaded to iCloud, and down to each attached device, each time you make a change.

For these reasons, Apple recommends that you employ other property list objects, such as `NSDictionary`, for your key-value storage values if possible.

Don't Use Key-Value Storage in Certain Situations

Every app submitted to the App Store or Mac App Store should adopt key-value storage, but some types of data are not appropriate for key-value storage. In a document-based app, for example, it is usually *not* appropriate to use key-value storage for state information about each document, such as current page or current selection. Instead, store document-specific state, as needed, with each document, as described in "Design for Persistent Document State" (page 38).

In addition, avoid using key-value storage for data essential to your app's behavior when offline; instead, store such data directly into the local user defaults database.

Designing for Documents in iCloud

Adopting iCloud document storage makes your app's documents available on all of a user's devices.

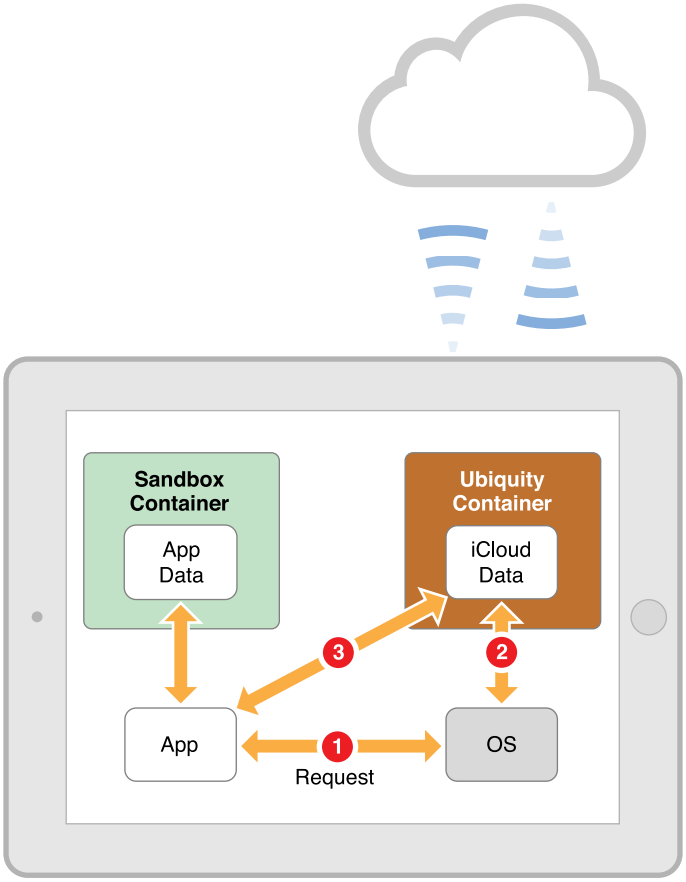
To your app, a **document** (based on the `UIDocument` class in iOS or the `NSDocument` class in OS X) is an assemblage of related data that can be written to disk as a single file or as a file package. A **file package**, in turn, is a directory presented to the user as a single file, accessible to your app using an `NSFileWrapper` object.

Documents automatically implement most of the behavior expected of iCloud apps. Specifically, a document automatically ensures that local changes are safely coordinated with iCloud-originated changes. It does this by employing a file coordinator (`NSFileCoordinator`) object and by adopting the file presenter (`NSFilePresenter`) protocol. In OS X, starting in version 10.8 Mountain Lion, documents automatically provide open/save/rename UI and functionality; in iOS your app must implement these things.

How iCloud Document Storage Works

Your app uses iCloud document storage by reading and writing to a `Documents` subdirectory of one of its ubiquity containers. Figure 3-1 shows a simplified representation of local storage on a device, and shows data transfer to and from the iCloud servers.

Figure 3-1 Data and messaging interactions for iCloud apps



The system automatically grants your app access to the data in your app's own sandbox container, but your app must request entitlements and employ iCloud storage APIs to use a ubiquity container. Figure 3-1 assumes that you have previously requested the appropriate entitlements, as explained in "Request Access to iCloud Storage By Using Entitlements" (page 11).

At step 1 in Figure 3-1, your app calls the `NSFileManager` method `URLForUbiquityContainerIdentifier:`, on a background thread, to set up iCloud storage and to obtain the URL for the specified ubiquity container. You must perform this step explicitly, except in the case of a document-based app in OS X, for which your document objects do this automatically on your behalf.

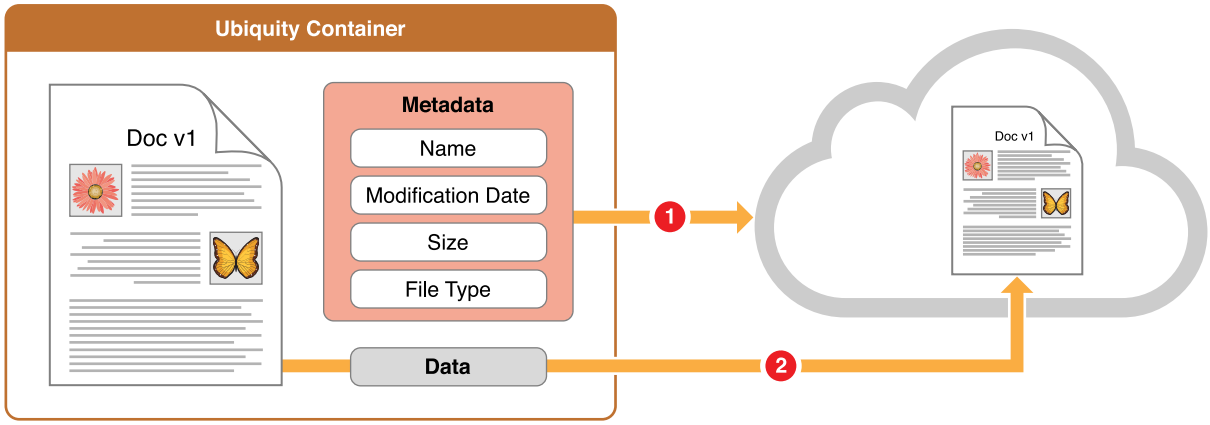
At step 2, the system grants your app access to the ubiquity container. Finally, at step 3, your app can read and write data to iCloud by accessing the ubiquity container.

Behind the scenes, the system manages the transfer of your app's iCloud data to and from the iCloud servers in a way that preserves data integrity and consistency. In particular, the system serializes those transfers with your app's file-system access, preventing conflicts from simultaneous changes. To support this serialization, iCloud apps must use special objects called *file coordinators* and *file presenters*. The name for serialized file system access using these objects is **file coordination**.

A document automatically uses a file coordinator (an instance of the `NSFileCoordinator` class) and adopts the file presenter (`NSFilePresenter`) protocol to manage its underlying file or file package. If you directly access a file or file package, such as to create it or delete it, you must use file coordination explicitly. For more information about using file coordination, see "The Role of File Coordination" in *File System Programming Guide*.

The first time your app adds a document's on-disk representation to a ubiquity container, the system transfers the entire file or file package to the iCloud server, as shown in Figure 3-2.

Figure 3-2 Transferring a file to iCloud the first time

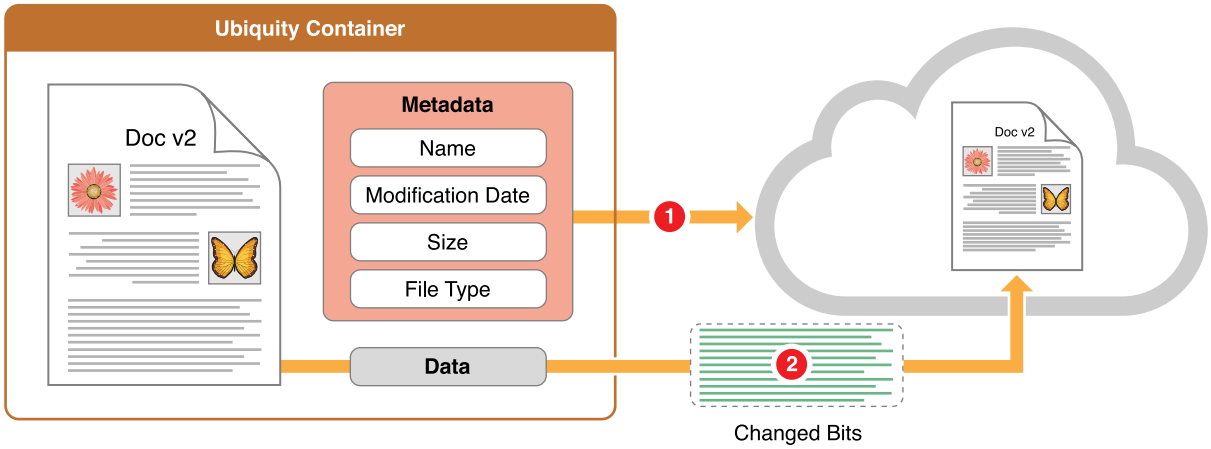


This transfer proceeds in two steps. In step 1 in the figure, the system sends the document's metadata, which includes information such as the document name, modification date, file size, and file type. This metadata transfer takes place quickly. At some later point, represented in step 2, the system sends the document's data.

Sending document metadata first lets iCloud quickly know that a new document exists. In response, the iCloud server propagates the metadata quickly to all other available devices attached to the same iCloud account. This lets the devices know that a new document is available.

After a document's data is on the iCloud server, iCloud optimizes future transfers to and from devices. Instead of sending the entire file or file package each time it changes, iCloud sends only the metadata and the pieces that changed. Figure 3-3 shows a visual representation of an incremental upload.

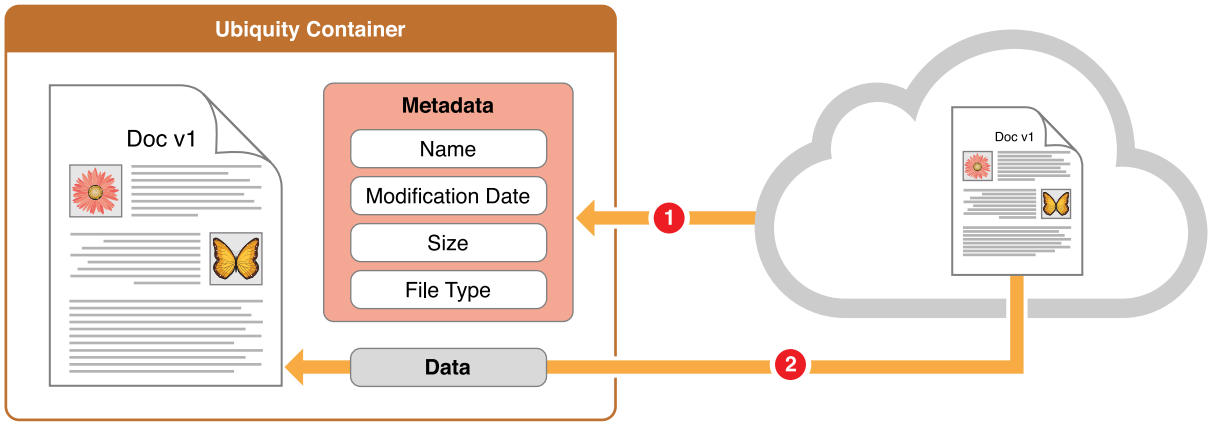
Figure 3-3 Transferring just the file changes to iCloud



Again, file metadata is sent quickly to the iCloud servers, as shown in step 1, so that it can be propagated to other devices. Because the system tracks changes to the document, it is able to upload only the parts that changed, as shown in step 2. This optimization reduces iCloud network traffic and also reduces the amount of power consumed by the device, which is important for battery-based devices. As you learn later in this chapter, in "Design for Network Transfer Efficiency" (page 37), good file format design is essential to supporting incremental transfer.

Downloading of a document from iCloud to a device works as follows: Say a user creates a document on one device, as you saw in Figure 3-2 (page 32). iCloud quickly sends that document's metadata to all other devices attached to the same account. Step 1 of Figure 3-4 represents this transfer of metadata to one of those other devices.

Figure 3-4 Receiving a file from iCloud for the first time



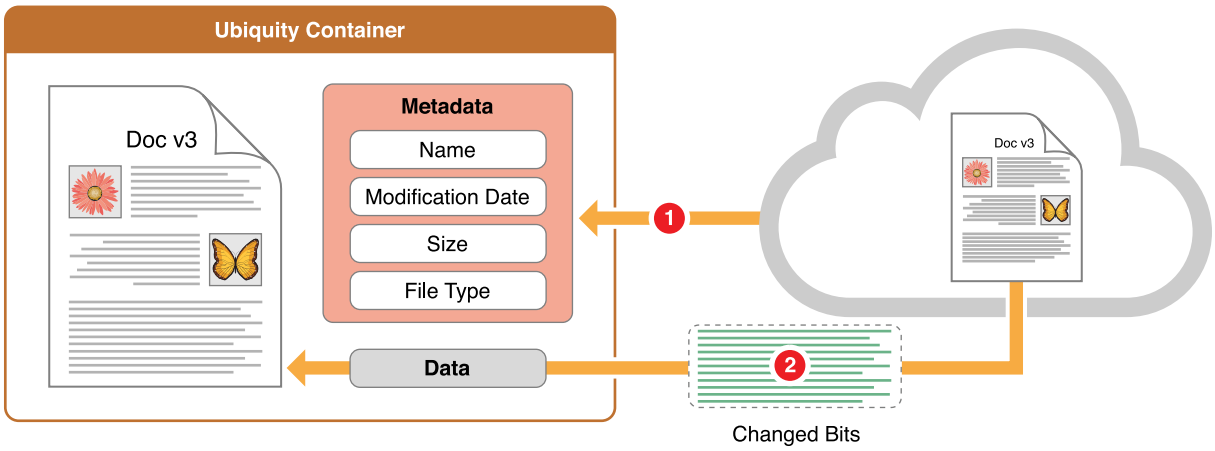
The details of step 2 in this figure depend on the receiving device's type.

An iOS device does not automatically download the new file, so your app must help. The steps to perform are described in "App Responsibilities for Using iCloud Documents" (page 34).

On a Mac, download is automatic for files created by other devices on the same account. For this reason, a Mac is sometimes referred to as a "greedy peer." If your Mac app sees metadata for a new document but the document itself is not yet local, it is likely already being downloaded by the system on your app's behalf.

Figure 3-5 depicts a device receiving updated metadata for a changed document that the device has previously downloaded. New metadata is received, in this scenario, because the user made a change to the document on another of their devices.

Figure 3-5 Receiving changed data from iCloud



When the device receives updated metadata (step 1), the system responds by asking iCloud to send the corresponding incremental data right away (step 2).

App Responsibilities for Using iCloud Documents

Changes to your app's documents can arrive from iCloud at any time, so your app must be prepared to handle them. The `NSDocument` class in OS X does most of this work for you, while in iOS there is more for your app to handle explicitly. Follow these guidelines:

- **Enable Auto Save.** For your app to participate with iCloud, you must enable Auto Save.

  In iOS, enable Auto Save by registering with the default `NSUndoManager` object or else by calling the `UIDocument` method `updateChangeCount:` at appropriate points in your code, such as when a view moves off the screen or your app transitions to the background.

In OS X, enable Auto Save by overriding the `NSDocument` class method `autosavesInPlace` to return `YES`.

- **In iOS, actively track document location.** Each instance of your iOS app must be prepared for another instance to move, rename, or delete iCloud-based documents. If your app persistently stores a URL or path information to a file or file package, do not assume that the item will still be there the next time you attempt to access it.

  In iOS, employ an `NSMetadataQuery` object, along with file coordination, to actively track the locations of your documents. Early in your app's launch process, instantiate and configure a metadata query object, start it, and register for its `NSMetadataQueryDidUpdateNotification` notification. Implement the `presentedItemDidMoveToURL:` method and the `presentedItemURL` property to allow your app to respond to pushed changes from iCloud. Refresh your app's model layer and update your app's user interface elements as needed.

  For a working example of using a metadata query in iOS, see "Searching for iCloud Documents" in *Your Third iOS App: iCloud* . For details on using metadata queries, see *File Metadata Search Programming Guide* .

- **In OS X, do not actively track document location.** The Open and Save dialogs in a document-based Mac app automatically track the locations of iCloud-based documents; you typically *do not* need to use an `NSMetadataQuery` object. For example, if a user renames or moves a document while working on one device, instances of your app running on other devices automatically pick up those changes by way of the document architecture.

- **In iOS, actively download files when required.** Items in iCloud but not yet local are not automatically downloaded by iOS; only their metadata is automatically downloaded. The initial download of new iCloud-based documents requires your attention and careful design in your app. After you explicitly download such an item, the system takes care of downloading changes arriving from iCloud.

  Consider keeping track of file download status as part of your iOS app's model layer. Having this information lets you provide a better user experience: you can design your app to not surprise users with long delays when they want to open a document that is not yet local. For each file (or file package) URL provided by your app's metadata query, get the value of the `NSURLUbiquitousItemIsDownloadedKey` key by calling the `NSURL` method `getResourceValue:forKey:error:`. Reading a file that has not been downloaded can take a long time, because the coordinated read operation blocks until the file finishes downloading (or fails to download).

  For a file (or file package) that is not yet local, you can initiate download either when, or before, the user requests it. If your app's user files are not large or great in number, one strategy to consider is to actively download all the files indicated by your metadata query. For each file (or file package) URL provided by the query, make the corresponding item local by calling the `NSFileManager` method `startDownloadingUbiquitousItemAtURL:error:`. If you pass this method a URL for an item that is already local, the method performs no work and returns `YES`.

- **In iOS, respond to, and resolve, document version conflicts as needed.** The document architecture supports conflict resolution by nominating a winning `NSFileVersion` object, but it is your iOS app's responsibility to accept the suggestion or override it. You can rely on this automatic nomination most of the time, but your app should be prepared to help as needed.

  A conflict arises when two instances of your iOS app, running on two different devices, attempt to change a document. This can happen, for example, if two devices are not connected to the network, a user makes changes on both of them, and then reconnects both devices to the network.

  When an iOS document's state changes, it posts a `UIDocumentStateChangedNotification` notification. On receiving this notification, query the document's `documentState` property and check if the value is `UIDocumentStateInConflict`. If you determine that explicit resolution is needed, resolve the conflict by using the `NSFileVersion` class. Enlist the user's help, if necessary; but when possible, resolve conflicts without user involvement. Keep in mind that another instance of your app, running on another device attached to the same iCloud account, might resolve the conflict before the local instance does.

  When done resolving a conflict, be sure to delete any out-of-date document versions; if you don't, you needlessly consume capacity in the user's iCloud storage.
- **In OS X, rely on the system to resolve document conflicts.** OS X manages conflict resolution for you when you use documents.
- **In OS X, avoid deadlocks resulting from modal UI elements.** It is possible for a document change to arrive from iCloud while your app is presenting a modal user interface element, such as a printing dialog, for the same document. If that incoming change requires its own modal interface, such as for conflict resolution, your app can deadlock.

  > **Important:** To avoid deadlocks when you present a document-modal interface element, ensure that your modal call is wrapped within a call to the `performActivityWithSynchronousWaiting:usingBlock:` method.

- **Always use a file coordinator to access an iCloud file or file package.** A document uses a file coordinator automatically, which is one of the great benefits of using documents when designing for iCloud.

  A document-based iOS app, however, must use a file coordinator explicitly when operating on a document's underlying file; that is, when moving, renaming, duplicating, or deleting the file. For these operations, use methods from the `NSFileManager` class within the context of a file coordinator writing method. For more information, see "The Role of File Coordination" in *File System Programming Guide*.

  Most document-based OS X apps should employ the built-in app-centric document viewing UI that appears when opening or saving documents. Even if your OS X app needs to present documents programmatically to the user, do not programmatically move, rename, or delete documents. It's up to the user to perform those operations using the Finder, the built-in document renaming UI, or the built-in iCloud "move" menu items in the File menu.

- **Don't let users unintentionally share information.** Unlike iOS users, OS X users have direct access to the file system. For example, if you keep an undo stack within your document format's file package, that information is accessible to a user viewing your app's document in OS X.

  This issue predates iCloud, but may not be familiar to you if you are primarily an iOS developer.

  When designing your document format, carefully consider which information your users would not want to share—for example, in an emailed version of the document. Instead of placing such information within the file package, associate it with the document but store it outside of the `Documents` subdirectory in the ubiquity container (see Figure 1-2 (page 13)).

- **Make your documents user-manageable, when appropriate.** Place files in the `Documents` subdirectory of an iCloud container to make them visible to the user and make it possible for the user to delete them individually. Files you place outside of the `Documents` subdirectory are grouped together as "data." When a user visits System Preferences (OS X) or Settings (iOS), they can delete content from iCloud. Files that you place outside of the `Documents` subdirectory can be deleted by the user only as a monolithic group.

  The choice of whether or not to use the `Documents` subdirectory depends on your app design. For example, if your app supports a user creating and naming a document, put the corresponding document file in the `Documents` subdirectory. If your app does not let users interact with files as discrete documents, it's more appropriate to place them outside of the `Documents` subdirectory.

Designing a Document File Format for iCloud

Choices you make in designing your document format can impact network transfer performance for your app's documents. The most important choice is to be sure to use a file package for your document format.

If you provide versions of your app for iOS and Mac, design your document file format to be cross-platform compatible.

Design for Network Transfer Efficiency

If your document data format consists of multiple distinct pieces, use a file package for your document file format. A **file package**, which you access by way of an `NSFileWrapper` object, lets you store the elements of a document as individual files and folders that can be read and written separately—while still appearing to the user as a single file. The iCloud upload and download machinery makes use of this factoring of content within a file package; only changed elements are uploaded or downloaded.

Register your document file format, and its associated filename extension, in your app's `Info.plist` property list file in Xcode. Specifically, use the "CFBundleDocumentTypes" in *Information Property List Key Reference* key to specify the file formats that your app recognizes and is able to open. Specify both a filename extension

and a uniform type identifier (UTI) corresponding to the file contents. The system uses this information to associate file packages to your app, and, in OS X, to display file packages to the user as though they were normal files.

Design for Persistent Document State

Many document-based apps benefit from maintaining state on a per-document basis. For example, a user of a vector-based drawing app would want each document to remember which drawing tool was most recently used and which element of the drawing was selected.

There are two places you can store document-specific state:

- Within the file package (or flat-file format) for a document. This choice supports keeping the state together with the document if, for example, a user sends it by email.
- Associated with the document but outside of its file package (or file format). This choice supports cases in which a user would not want to share information. But, being outside of the data managed by your document objects, such state is not tracked by a document's conflict resolution functionality. It is up to you to do so.

> **Note:** Key-value storage, in most cases, is not appropriate for storing document-specific state. This is because of the different ways in which the system uploads and downloads data to the two types of storage, and the different APIs used to access them.

Whichever scheme you choose, take care, in an app that supports editing, to never save document state unless document content was edited. Otherwise, you invite trivial and unhelpful conflict scenarios that consume network bandwidth and battery power.

For example, imagine that a user had been editing a long text document on their iPad, working on page 1. A bit later, they open the document on their iPhone and scroll to the last page. This badly-behaved example app aggressively saves the end-of-document scroll position—even though the user made no other changes. When the user later opens the document on their iPad to resume editing, there is a needless conflict due to scroll position data. The system has marked the document as in conflict (`UIDocumentStateInConflict`), and the newer version (automatically nominated as the conflict winner) is the one scrolled to the last page. To be a well-behaved iCloud app, this text editor should have ignored the change in scroll position on the iPhone because the user did not edit the content.

Think through various usage scenarios for your app, and design accordingly to improve user experience. Take care with state like:

- Document scroll position

- Element selection
- Last-opened timestamps
- Table sort order
- Window size (in OS X)

A strategy to consider is to save such state but only when the user has also made a change that deserves saving. In OS X, the built-in Resume feature provides all the state saving behavior that most document-based apps need. If you want additional control, you can take advantage of the `NSChangeDiscardable` document change type.

Design for Robustness and Cross-Platform Compatibility

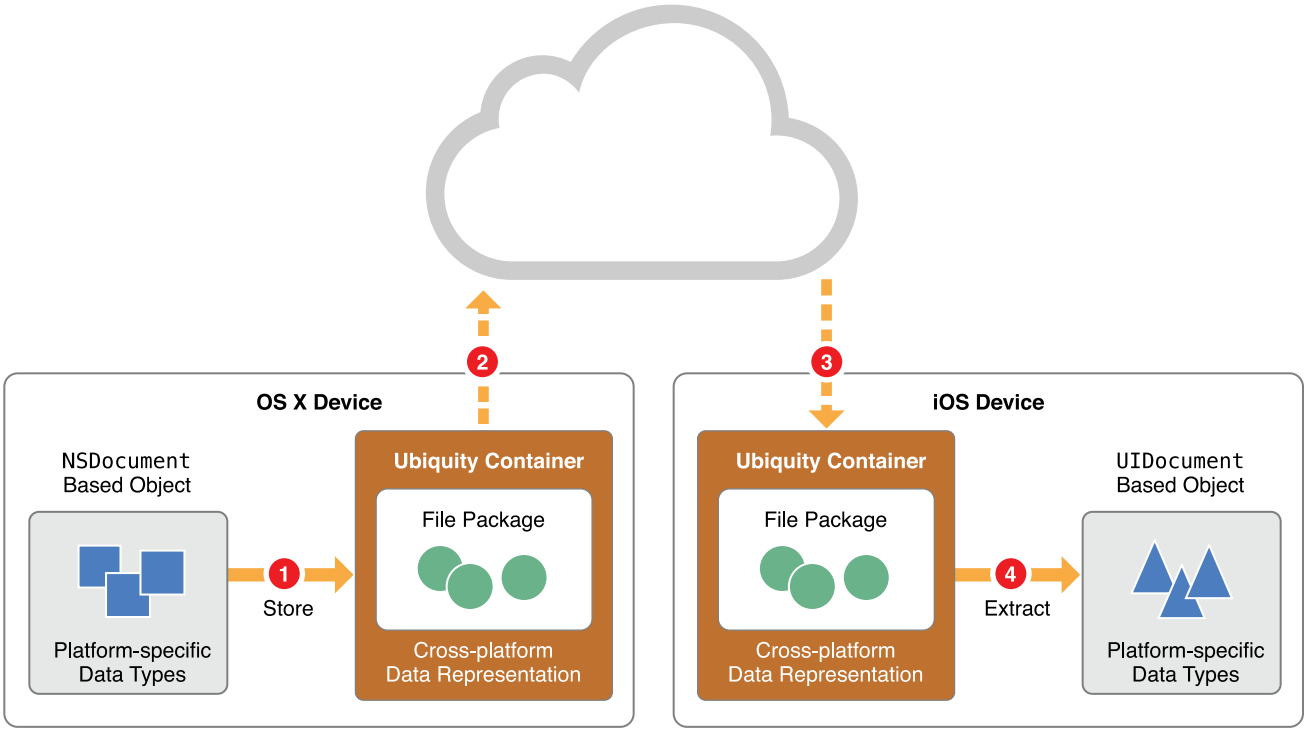
Keep the following factors in mind when designing a document file format for iCloud:

- **Employ a cross-platform data representation.** Like-named classes—those whose prefixes are "UI" for iOS and "NS" for OS X—are not directly compatible. This is true for colors (`UIColor` and `NSColor`), images (`UIImage` and `NSImage`), and Bezier paths (`UIBezierPath` and `NSBezierPath`), as well as for other classes.

  For example, an OS X `NSColor` object is defined in terms of a color space (`NSColorSpace`), but there is no color space class in iOS.

If you employ such a class in a document format property, you must devise an intermediate iCloud representation that you can faithfully reconstitute into the native representation on either platform, as depicted in Figure 3-6.

Figure 3-6 Using a cross-platform data representation



In step 1, your OS X app stores an in-memory document object to disk, converting each platform-specific data type to an appropriate cross-platform representation. Steps 2 and 3 show the data uploaded and then downloaded to an iOS device. In step 4, the iOS version of your app extracts the data, converting from the intermediate representation into iOS-specific data types, thereby constructing a `UIDocument`-based version of the original `NSDocument`-based object.

When storing a document from your iOS app for later viewing on either platform, you'd reverse the process shown in Figure 3-6—starting by storing your iOS document using a cross-platform data representation.

For each platform-specific data type you employ in your document format, check if there is an appropriate, lower-level data type shared by both platforms that you can use in the intermediate representation. For example, each color class (`UIColor` and `NSColor`) has an initialization method that lets you create a color object from a Core Image `CIColor` instance.

When preparing data for writing to iCloud, convert *to* the intermediate representation; when reading an iCloud-based file, convert *from* the intermediate representation. If you call methods from `NSCoder` or its concrete subclasses to encode and decode your document's object graph, perform these conversions within those methods.

- **Take platform-specific coordinate systems into account.** The default screen coordinate systems in iOS and OS X are different, leading to differences in how you draw and in how you position views. Take this into account when storing and extracting screen coordinate information with your cross-platform representation. For more information, see "Coordinate Systems and Drawing in iOS" in *Drawing and Printing Guide for iOS*.
- **Always use case-insensitive filenames.** OS X, by default, and for nearly all users, employs a case-*insensitive* file system. For example, the file `mydoc.txt` and the file `MyDoc.TXT` cannot both exist within the same directory in OS X. By contrast, iOS treats those file names as being different.

  To make your document file format cross-platform compatible, you must read and write files in a case-insensitive manner.
- **Use a format version number.** Because you may want to change your document format in the future, design a format version numbering scheme and apply the version number as a property of the document format.

  Versioning your document format has always been a good idea, but with iCloud and cross-platform formats, it's even more important. For example, an iCloud user is likely to have multiple devices on which they can view their content; but the user might not conscientiously update your app on all of their devices at once. So, for example, a user's Mac might have the newest version of your OS X app but their iPad might have a year-old version of your iOS app.

  An early version of your app might be iOS-only; when you later distribute an OS X version, you'll likely want to modify the document format. By embedding the format version within each document, your code can gracefully handle documents regardless of their version. You might, for example, make an old, iOS-only format read-only in OS X. Think through various scenarios involving multiple devices on each platform, each running a different version of your app. Make sure that you provide the best possible user experience for iCloud-based documents.

For more information about techniques for designing file formats, see "Choosing Types, Formats, and Strategies for Document Data" in *Document-Based App Programming Guide for iOS* or "Handling a Shared Data Model in OS X and iOS" in *Document-Based App Programming Guide for Mac*.

Document-Based Workflows

Table 3-1 lists some typical workflows for document-based apps. For each workflow, the table lists the primary classes you typically use, along with a brief description of the task.

Table 3-1 Typical document-based app workflows

Workflow	Implementation	Description
Create a new standard document	`UIDocument` (iOS) `NSDocument` (OS X)	Use a document object to create and manage the data structures in your document format. The document classes automatically support saving new documents to a ubiquity container or to local storage.
Create a new Core Data document	`UIManagedDocument` (iOS) `NSPersistentDocument` (OS X)	Use the Core Data document subclasses to create and manage your Core Data stores. For details, see "Designing for Core Data in iCloud" (page 43).
Obtain URLs to iCloud documents	`NSMetadataQuery` (iOS) Automatic (OS X)	In iOS, use a metadata query object to locate and obtain live-updated information on iCloud documents. In OS X v10.8 or later, a document Open dialog uses a metadata query automatically.
Prompt the user to open a document.	Custom UI (iOS) Automatic as part of the document architecture (OS X)	In iOS, your app is directly responsible for presenting a selection UI for user documents in a simple and clean way that fits well with the rest of the app design. In OS X v10.8 or later, the Open command in a document-based app presents a dialog that lets the user select iCloud files.
Handle version conflicts	`UIDocument`, `NSFileVersion` (iOS) Automatic (OS X)	In iOS, documents detect and notify you about conflicts. Use `NSFileVersion` objects (one per revision of a document) to resolve them, as needed.
Move, duplicate, and delete iCloud-based documents	`NSFileCoordinator` `NSFileManager`	Manipulate files on disk using the `NSFileManager` class, and always within the context of a file coordinator object.

For details on how to perform the preceding workflows, look in the document-based programming guide for the platform you are targeting. For iOS, see *Document-Based App Programming Guide for iOS*. For Mac, see *Document-Based App Programming Guide for Mac*.

Designing for Core Data in iCloud

Adopting iCloud Core Data storage makes the content in a shoebox-style app (like iPhoto) or a database-style app (like Bento) available on all of a user's devices.

> **Note:** To create a Core Data app, you design a managed object model and work with managed objects. If you are not already familiar with Core Data, read *Core Data Programming Guide*. For the latest information about using iCloud with Core Data, read *Using Core Data with iCloud Programming Notes*.

Although Core Data can work with a variety of persistent store file types, this chapter focuses on using Core Data with SQLite stores. One advantage of using SQLite in an iCloud-enabled app is that it can minimize network traffic by sending incremental changes to iCloud.

Core Data Sends Incremental Changes to iCloud

Each instance of your app, on each device attached to an iCloud account, maintains its own local Core Data store file. When data changes locally, Core Data writes change log files to your app's default ubiquity container.

> **Note:** You can specify a non-default container for Core Data change log files by configuring your instance of the `NSPersistentStoreCoordinator` class—specifically by using the `NSPersistentStoreUbiquitousContentURLKey` key.

The change log files, not the store file, are uploaded to iCloud and downloaded to each of a user's other devices. When a change log arrives from another device attached to the same iCloud account, Core Data updates your app's local copy of the SQLite database, based on the received change log. iCloud and Core Data ensure that each local database is updated with the same set of changes.

Early in your app's life cycle, register for the `NSPersistentStoreCoordinator` notification `NSPersistentStoreDidImportUbiquitousContentChangesNotification`. When a Core Data store has imported changes from iCloud, it posts this notification. On receiving it, refresh the affected records and update the user interface.

When you use a SQLite Core Data store with iCloud, keep the following points in mind:

- Place your SQLite Core Data store within a `<my_folder>.nosync` subdirectory of one of your app's ubiquity containers. This placement ensures that, if a user switches iCloud accounts, the system takes care of keeping each account's data associated with the correct account.

  > **Note:** If your app must be able to operate when a user is signed out of iCloud, place the store, instead, in your sandbox container. If you do this, however, you must explicitly manage data access.

- Do not prepopulate the contents of a new SQLite store, such as to provide seed records the first time a user launches your app, by moving or copying an existing store file to a new location.

  If your existing store file size is small, migrate its contents using the `NSPersistentStoreCoordinator` method `migratePersistentStore:toURL:options:withType:error:`.

  If your store file is large, take care regarding the amount of memory consumed during migration, which is on the order of twice the size of the store. Transfer data in batches by using your app's `NSManagedObjectContext` instance along with the `NSFetchRequest` class, setting the amount of data-per-batch with the `setFetchBatchSize:` method.

- When using iCloud, the SQLite store on the device is a cache that represents data imported from the Core Data change log files. If there are pending transactions in change logs that Core Data has not yet processed, the store does not have the most current data.

- To delete a store file, you must first tear down your Core Data stack. Then delete both the file package for the store and the directory containing the store's transaction logs.

  Each of these deletions requires you to perform a coordinated write operation using an `NSFileCoordinator` object. For more information about deleting a SQLite store, see *Using Core Data with iCloud Programming Notes* .

Binary Stores for iCloud Apps Have Limitations

Core Data binary stores, and Core Data XML stores (OS X only), work differently with iCloud than do SQLite stores. Binary (and XML) store files are themselves transferred to the iCloud servers; so, whenever a change is made to the data, the system uploads the entire store and pushes it to all connected devices. For a large data set, or for a data set that changes frequently, these types of stores result in more data transfer to and from the iCloud servers and are consequently less efficient than using a SQLite store.

Because of this inherent reduction in efficiency, using binary or XML stores for all but small and infrequently-changing data sets results in excess network traffic. For small data sets that change infrequently, however, a binary store can work well in an iCloud-enabled app.

The rest of this chapter discusses only apps that use Core Data SQLite stores.

iOS Supports Core Data Documents in iCloud

A managed document is represented on disk as a file package, and includes its own Core Data store within the package. Using Core Data APIs, you can establish formal relationships among the elements in a managed document.

To use Core Data documents in iOS, use the `UIManagedDocument` class, a concrete subclass of the abstract `UIDocument`. You can use the `UIManagedDocument` class directly or you can subclass it.

> **Note:** You cannot use the additional-content APIs (described in "Customizing Read and Write Operations" in *UIManagedDocument Class Reference*) for iCloud.

When using iCloud, Core Data attempts to resolve and merge changes between versions of a managed document. Just as with an iCloud-enabled Core Data app that does not use managed documents, register for the `NSPersistentStoreCoordinator` notification `NSPersistentStoreDidImportUbiquitousContentChangesNotification`. When a managed document has imported data from iCloud, it posts this notification. On receiving it, refresh the affected records and update the document's display.

When you create a managed document, you need to specify a unique location for its change log files so that later, if you need to delete the document, you can find those files (to delete them as well).

To set the location for a managed document's change log files, specify a unique name (using the `NSPersistentStoreUbiquitousContentNameKey` key) and a unique URL (using the `NSPersistentStoreUbiquitousContentURLKey` key) in the managed document's `persistentStoreOptions` property.

For the URL, choose a location within the ubiquity container that you want to use, but outside of the `Documents` subdirectory. The document saves the specified name and URL in its `DocumentMetadata.plist` property list file, which is present inside the document's file package.

To save a managed document in iOS, you can use any of the following three approaches, listed with the most recommended approach first:

- Use the inherited Auto Save behavior provided by the `UIDocument` superclass. For most apps, this provides good performance and low network traffic.
- If your app design requires explicit control over when pending changes are committed, use the `UIDocument` method `saveToURL:forSaveOperation:completionHandler:`.

If you perform an explicit save operation in an iCloud-enabled app, be aware that you are generating additional network traffic—multiplied by the number of devices connected to the iCloud account.

- If you have a specific case in which neither of the two preceding approaches works, such as importing a large quantity of data in the background, you can explicitly save the document's managed object context.

  A managed document has two contexts: The one it presents to you is a child of a second context that the document uses internally to communicate with the document's Core Data store.

  If you save only the document's public context, you are not committing changes to the store; you still end up relying on Auto Save. To explicitly save a document's managed object context, explicitly save *both* contexts. For more on this, read "Saving Changes" in *Core Data Programming Guide*.

To delete a managed document, you must first close the document. Then you must delete two directories, using coordinated write operations:

- The directory for the Core Data change logs for the managed document
- The managed document's file package

To obtain the location of the directory containing the change log files, look at the document's `DocumentMetadata.plist` property list file. Retrieve the name and URL that you set when creating the document.

In OS X, the NSPersistentDocument Class Does Not Support iCloud

In OS X, Core Data integrates with the document architecture through the `NSPersistentDocument` class. However, in OS X v10.8, instances of this class do not provide specific support for iCloud.

Design the Launch Sequence for Your iCloud Core Data App

When you adopt iCloud, take special care when designing the launch sequence for your app. The following factors come into play and you must account for them:

- The user might or might not have previously indicated a preference to use iCloud in your app; the local instance of your app might or might not have already established its initial store in a ubiquity container.

  As a first step in your launch sequence, read the local user defaults database using the shared `NSUserDefaults` object. During operation of your app, use that object to save user choices that you'll need on next launch.

- iCloud might or might not be available.

The user might have set the device to Airplane mode, or the network might be otherwise unavailable. Determine how you want your app to behave when iCloud is unavailable. For example, you could allow the creation of new records but keep existing records read-only. If you allow editing of existing records, be prepared for additional data reconciliation when your app next connects to iCloud.

- The user might log out of iCloud or switch to another account.

  If a user logs out of iCloud, or switches to another account, the ubiquity containers for the previously-used account are no longer available to your app.

- The local Core Data store might be newer or older than the store on another device owned by the same user.

  During app launch, Core Data might need to reconcile the local store with change logs from iCloud. This can involve detection and resolution of duplicate records and conflicts. Testing is critical. To get started with some tips, refer to "Testing and Debugging Your iCloud App" (page 48).

Testing and Debugging Your iCloud App

Although no app is finished until it has been thoroughly tested and debugged, these steps are perhaps more important when you adopt iCloud: the number of user scenarios that your app must gracefully handle is greater. A user can have different versions of your app on different devices, or run your app on more than one device simultaneously. A user can turn on Airplane mode while a large file is uploading to iCloud. In rare cases, a user might log out of iCloud or switch to a different iCloud account.

This chapter helps orient you toward thinking about iCloud when testing and debugging your app.

Make Sure Your Device is Configured for iCloud

If things aren't working as expected, first make sure that the preliminaries are correctly in place.

- **Ensure that your test devices are correctly provisioned for iCloud.**

  Your device provisioning profile and app ID must be correctly configured for iCloud. Review "Adding Capabilities" in *App Distribution Guide*.

- **Ensure that your iCloud entitlement requests are correct.**

  Without the appropriate entitlement requests in place in your Xcode project, your app has no access to its ubiquity containers or to key-value storage. Carefully review the information in "Request Access to iCloud Storage By Using Entitlements" (page 11).

  Ensure that when you access your app's ubiquity containers, you are using fully-qualified entitlement values.

Take Latency Into Account

When testing, don't expect changes to instantly propagate from one device to another.

You can programmatically determine if a file has made it to the iCloud servers by checking the value of its URL's `NSURLUbiquitousItemIsUploadedKey` key. Do this by calling the `NSURL` method `getResourceValue:forKey:error:`. The key's Boolean `NSNumber` value contains `true` if the file has been successfully uploaded.

Likewise, you can programmatically determine if an iCloud file is present locally. Check the value of the `NSURLUbiquitousItemIsDownloadedKey` key of the corresponding URL by calling the `NSURL` method `getResourceValue:forKey:error:`.

In both cases, obtain the file URLs from your app's `NSMetadataQuery` object, as described in "App Responsibilities for Using iCloud Documents" (page 34).

Monitor Your App's Network Traffic

Keep an eye on network activity while testing your app, checking for undue amounts of network traffic. For iOS, use the "Network Activity Instrument", described in *Instruments User Reference*. On a Mac, you can use the OS X Network Utility app.

If your app's network traffic seems excessive, look for optimizations in your app design. For example, you may be able to design your document file package in a way to better support incremental uploads and downloads, as described in "Design for Network Transfer Efficiency" (page 37).

Use Two Devices to Test Document Conflicts

Real world use of your iCloud-enabled app can result in a wide range of conflict scenarios between versions of a document. Beyond that unavoidable truth, issues in an app's logic can cause needless conflicts, as described in "Design for Persistent Document State" (page 38).

Here are some ideas for simulating real-world conflicts. To prepare for these tests, set up two devices to be connected to a single iCloud account. Implement your app to handle these scenarios and others that come to mind for your particular app.

To test document version conflicts with one device offline

1. On device 1, create and open a new document using your app, and then open the document on device 2.

   The document is now open on both devices.

2. On device 1, turn on Airplane mode and then edit the document.
3. On device 2 (which is still online), edit the document as well.
4. On device 1, turn off Airplane mode.
5. On each device, check whether the conflict resolution behavior is as you intend it to be.

To test document version conflicts with two devices offline

1. On device 1, create and open a new document using your app, then open the document on device 2.

   The document is now open on both devices.

2. Turn on Airplane mode on both devices, then edit the document on both devices.

   Try varying this procedure by finishing your changes on the two devices at the same time or at different times.

3. On both devices, turn off Airplane mode.

   For this step as well, try varying this procedure by turning off Airplane mode at the same time or at different times for the two devices.

4. On each device, check whether the conflict resolution behavior is as you intend it to be.

To test forward and backward compatibility of your document format

1. Install the oldest supported version of your app on device 1; install the newest version on device 2.
2. Create a document on each device. Allow enough time to elapse that each doc appears on the other device..
3. Open each document on the device that was not used to create it. Or, in the case of a version mismatch in which you do not support opening, ensure that the behavior is as you intend it to be. Check for issues.

To test document management

1. Save a document to iCloud, then open that document on two devices.
2. Using device 1, rename (or, alternatively, delete) the document.
3. Check on device 2 whether the behavior is as you intend it to be.

Start Fresh if Your iCloud Data Becomes Inconsistent During Development

While you are developing an app, it is possible for data in the app's ubiquity container to become inconsistent. If this happens, your app's behavior can become inconsistent as well. If previously-working code is now not working, try emptying the ubiquity container for your app to start fresh.

You can empty your app's ubiquity container using an iOS device or a Mac.

To empty the ubiquity container for your app, using an iOS device

1. On each device containing an instance of your app, delete that instance.
2. In iOS, navigate to Settings > iCloud > Storage & Backup > Manage Storage.
3. If your app is not shown in the Documents & Data group, tap Show All.
4. Tap your app's name in the Manage Storage screen.
5. On your app's storage Info screen, tap Edit, then tap Delete All.

   In the confirmation alert that appears, tap Delete All.

After completing these steps, wait to ensure that deletion of your ubiquity container's contents has propagated to all devices attached to your iCloud account. If the container had a large amount of data, this takes longer. On a Mac, you can use the Finder to see if the ubiquity container has been emptied. The path to the container on a Mac is:

```
~/Library/Mobile Documents/<bundle-identifier-for-app>
```

Then reinstall your app on each device.

To empty the ubiquity container for your app, using a Mac

1. On each device containing an instance of your app, delete that instance.
2. In OS X, open iCloud preferences in System Preferences and click Manage.
3. In the Manage Storage dialog that appears, locate your app in the list and click it.
4. Click Delete All.

   In the confirmation alert that appears, click Delete. Then click Done.

After completing these steps, wait to ensure that deletion of your ubiquity container's contents has propagated to all devices attached to your iCloud account. If the container had a large amount of data, this takes longer. On a Mac, you can use the Finder to see if the ubiquity container has been emptied. The path to the container on a Mac is:

```
~/Library/Mobile Documents/<bundle-identifier-for-app>
```

Then reinstall your app on each device.

If you provide an OS X version of your iCloud-enabled app, and you want to start completely fresh, also delete the contents of the app's App Sandbox container on each Mac on which your app was installed. This ensures that any other app-specific data, that might have become inconsistent, is also removed. The path for an App Sandbox container on a Mac is:

```
~/Library/Containers/<bundle-name-for-app>
```

Document Revision History

This table describes the changes to *iCloud Design Guide*.

Date	Notes
2013-09-18	Minor corrections and additions. Corrected text in Figure 1-4 (page 20). Added information about coordinated reads of files that have not yet been downloaded to "Designing for Documents in iCloud" (page 30).
2012-09-19	Added material throughout the document. Changed guidance on using wildcard characters in ubiquity container entitlement values, in "Request Access to iCloud Storage By Using Entitlements" (page 11). Reorganized and expanded the material in "Prepare Your App to Use iCloud" (page 16). Corrected Figure 1-4 (page 20). Simplified the code listing in "Prepare Your App to Use the iCloud Key-Value Store" (page 26). Clarified the discussion around Figure 2-3 (page 27). Added information to "Data Size Limits for Key-Value Storage" (page 28). Added two new sections in "Designing for Key-Value Data in iCloud" (page 24): "Exercise Care When Using NSData Objects as Values" (page 29) and "Don't Use Key-Value Storage in Certain Situations" (page 29). Improved the discussion in "App Responsibilities for Using iCloud Documents" (page 34). Improved the discussion in "Designing for Core Data in iCloud" (page 43). Clarified the task steps in "Use Two Devices to Test Document Conflicts" (page 49).

Date	Notes
	Added a new section about testing, "Start Fresh if Your iCloud Data Becomes Inconsistent During Development" (page 50).
2012-07-21	Added material throughout the document.
2012-07-10	New document that introduces the steps for incorporating iCloud support into your app.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010