Document-Based App Programming Guide for iOS

Developer

Contents

About Document-Based Applications in iOS 6
At a Glance 7
    Document Objects Are Model Controllers 7
    When Designing an Application, Consider Document-Data Format and Other Issues 7
    Creating a Subclass of UIDocument Requires Two Method Overrides 7
    An Application Manages a Document Through Its Life Cycle 7
    An Application Stores Document Files in iCloud Upon User Request 8
    An Application Ensures That Document Data is Saved Automatically 8
    An Application Resolves Conflicts Between Different Document Versions 8
How to Use This Document 8
Prerequisites 9
See Also 9

Designing a Document-Based Application 10
Why Create a Document-Based Application? 10
A Document in iOS 11
    A Document in iCloud Storage 12
    Properties of a UIDocument Object 12
Design Considerations for Document-Based Applications 13
    Defining Object Relationships 13
    Designing the User Interface 14
    Choosing Types, Formats, and Strategies for Document Data 15

Document-Based Application Preflight 19
What You Must Do to Make a Document-Based Application 19
Creating and Configuring the Project 20
    How iOS Identifies Your Application's Documents 21
    Declare a Document Type 21
    Exporting the Document UTI 23

Creating a Custom Document Object 24
Declaring the Document Class Interface 24
Loading Document Data 25
Supplying a Snapshot of Document Data 27

Storing Document Data in a File Package 28
Other Method Overrides You Might Make 29

Managing the Life Cycle of a Document 31
Setting the Preferred Storage Location for Document Files 31
Creating a New Document 32
    Document Filename Versus Document Name 32
    Composing the File URL and Saving the Document File 32
Opening and Closing a Document 34
    Discovering an Application's Documents 34
    Downloading Document Files from iCloud 37
    Opening a Document 37
    Closing a Document 39
Moving Documents to and from iCloud Storage 40
    Getting the Location of the iCloud Container Directory 40
    Moving a Document to iCloud Storage 41
    Removing a Document from iCloud Storage 42
Monitoring Document-State Changes and Handling Errors 43
Deleting a Document 46

Change Tracking and Undo Operations 48
How UIKit Saves Document Data Automatically 48
Implementing Undo and Redo 48
Implementing Change Tracking 50

Resolving Document Version Conflicts 51
Learning About Document Version Conflicts 51
Strategies for Resolving Document Version Conflicts 51
How to Tell iOS That a Document Version Conflict Is Resolved 52
An Example: Letting the User Pick the Version 53

Document Revision History 56

Figures, Tables, and Listings

Designing a Document-Based Application 10

Figure 1-1 Document file, document object, and model objects managed by the document object 12
Figure 1-2 A view controller manages both a document object and the view that presents document data 14

Document-Based Application Preflight 19

Figure 2-1 IOS looking up a document UTI from the extension of its file URL 21
Figure 2-2 Specification of a document type in Xcode 22
Figure 2-3 Exporting a custom document UTI in Xcode 23
Table 2-1 Properties for defining a document type (`CFBundleDocumentTypes`) 22
Table 2-2 Properties for exporting a document UTI (`UTExportedTypeDeclarations`) 23

Creating a Custom Document Object 24

Figure 3-1 Structure of a file package 28
Listing 3-1 Document subclass declarations (`NSData`) 24
Listing 3-2 Document subclass declarations (`NSFileWrapper`) 25
Listing 3-3 Loading a document's data (`NSData`) 26
Listing 3-4 Loading a document's data (`NSFileWrapper`) 26
Listing 3-5 Returning a snapshot of document data (`NSData`) 27
Listing 3-6 Returning a snapshot of document data (`NSFileWrapper`) 27

Managing the Life Cycle of a Document 31

Table 4-1 `UIDocumentState` constants 44
Listing 4-1 Getting a URL to the application's `Documents` directory in the local sandbox 33
Listing 4-2 Saving a new document to the file system 33
Listing 4-3 Getting the locations of documents stored locally and in iCloud storage 35
Listing 4-4 Collecting information about documents in iCloud storage 36
Listing 4-5 Responding to a request to open a document 37
Listing 4-6 Opening a document 38
Listing 4-7 Closing a document 39
Listing 4-8 Getting the iCloud container directory URL 40
Listing 4-9 Moving a document file to iCloud storage from local storage 41
Listing 4-10 Moving a document file from iCloud storage to local storage 42
Listing 4-11 Adding an observer of the `UIDocumentStateChangedNotification` notification 44

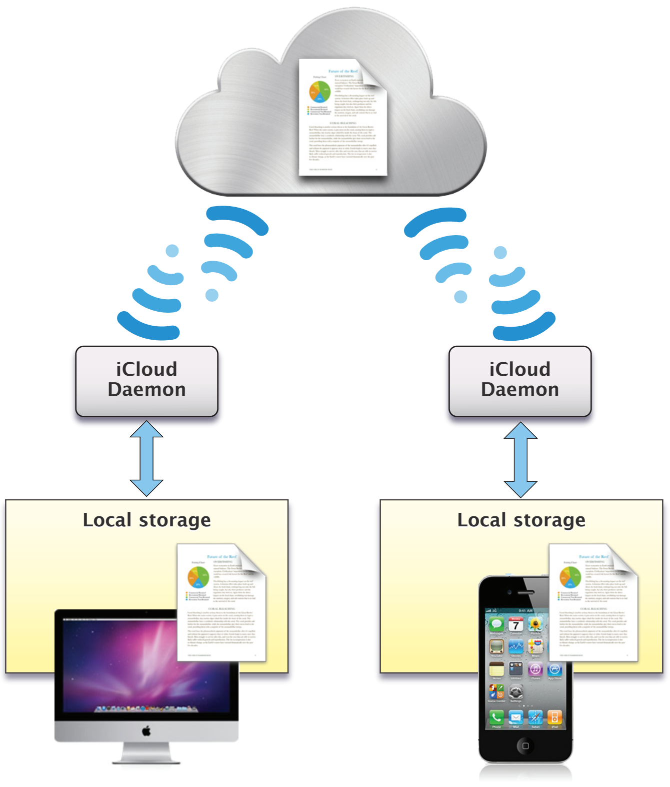
Listing 4-12 Evaluating the current document state 45
Listing 4-13 Updating a document’s user interface to reflect its state 45
Listing 4-14 Deleting a selected document 46

Change Tracking and Undo Operations 48
Listing 5-1 Implementing undo and redo for a text field 49
Listing 5-2 Updating the change count of a document 50

Resolving Document Version Conflicts 51
Listing 6-1 Detecting a conflict in document versions 53
Listing 6-2 Showing the user interface for resolving document version conflicts 54
Listing 6-3 Resolving a document version conflict 54

About Document-Based Applications in iOS

The UIKit framework offers support for applications that manage multiple documents, with each document containing a unique set of data that is stored in a file located either in the application sandbox or in iCloud.



Central to this support is the `UIDocument` class, introduced in iOS 5.0. A document-based application must create a subclass of `UIDocument` that loads document data into its in-memory data structures and supplies `UIDocument` with the data to write to the document file. `UIDocument` takes care of many details related to document management for you. Besides its integration with iCloud, `UIDocument` reads and writes document data in the background so that your application's user interface does not become unresponsive during these operations. It also saves document data automatically and periodically, freeing your users from the need to explicitly save.

At a Glance

Although a document-based application is responsible for a range of behaviors, making an application document-based is usually not a difficult task.

Document Objects Are Model Controllers

In the Model-View-Controller design pattern, document objects—that is, instances of subclasses of `UIDocument`—are model controllers. A document object manages the data associated with a document, specifically the model objects that internally represent what the user is viewing and editing. A document object, in turn, is typically managed by a view controller that presents a document to users.

Relevant Chapter: "Designing a Document-Based Application" (page 10)

When Designing an Application, Consider Document-Data Format and Other Issues

Before you write a line of code you should consider aspects of design specific to document-based applications. Most importantly, what is the best format of document data for your application, and how can you make that format work for your application in iOS *and* Mac OS X? What is the most appropriate document type?

You also need to plan for the view controllers (and views) managing such tasks as opening documents, indicating errors, and moving selected documents to and from iCloud storage.

Relevant Chapters: "Designing a Document-Based Application" (page 10), "Document-Based Application Preflight" (page 19)

Creating a Subclass of UIDocument Requires Two Method Overrides

The primary role of a document object is to be the "conduit" of data between a document file and the model objects that internally represent document data. It gives the `UIDocument` class the data to write to the document file and, after the document file is read, it initializes its model objects with the data that `UIDocument` gives it. To fulfill this role, your subclass of `UIDocument` must override the `contentsForType:error:` method and the `loadFromContents:ofType:error:` method, respectively.

Relevant Chapter: "Creating a Custom Document Object" (page 24)

An Application Manages a Document Through Its Life Cycle

An application is responsible for managing the following events during a document's lifetime:

- Creation of the document
- Opening and closing the document

- Monitoring changes in document state and responding to errors or version conflicts
- Moving documents to iCloud storage (and removing them from iCloud storage)
- Deletion of the document

Relevant Chapter: "Managing the Life Cycle of a Document" (page 31)

An Application Stores Document Files in iCloud Upon User Request

Applications give their users the option for putting all document files in iCloud storage or all document files in the local sandbox. To move document files to iCloud, they compose a file URL locating the document in an iCloud container directory of the application and then call a specific method of the `NSFileManager` class, passing in the file URL. Moving document files from iCloud storage to the application sandbox follows a similar procedure.

Relevant Chapter: "Managing the Life Cycle of a Document" (page 31)

An Application Ensures That Document Data is Saved Automatically

`UIDocument` follows the saveless model and automatically saves a document's data at specific intervals. A user usually never has to save a document explicitly. However, your application must play its part in order for the saveless model to work, either by implementing undo and redo or by tracking changes to the document.

Relevant Chapter: "Change Tracking and Undo Operations" (page 48)

An Application Resolves Conflicts Between Different Document Versions

When documents are stored in iCloud, conflicts between versions of a document can occur. When a conflict occurs, UIKit informs the application about it. The application must attempt to resolve the conflict itself or invite the user to pick the version he or she prefers.

Relevant Chapter: "Resolving Document Version Conflicts" (page 51)

How to Use This Document

Before you start writing any code for your document-based application, you should at least read the first two chapters, "Designing a Document-Based Application" (page 10) and "Document-Based Application Preflight" (page 19). These chapters talk about design and configuration issues, and give you an overview of the tasks required for well-designed document-based applications

Prerequisites

Before you read *Document-Based Application Programming Guide for iOS* you should become familiar with the information presented in *iOS App Programming Guide*, especially the material on iCloud integration.

See Also

The following documents are related in some way to *Document-Based Application Programming Guide for iOS*:

- *Document-Based App Programming Guide for Mac* describes how to create document-based applications for OSX. If you want documents to be shared between an iOS version of an application and an OSX version of the same application, you should become familiar with this document.
- *Uniform Type Identifiers Overview* and the related reference discuss Uniform Type Identifiers (UTIs), which are the primary identifiers of document types.
- *File Metadata Programming Guide* describes how to conduct searches using the `NSMetadataQuery` class and related classes. You use metadata queries to locate an application's documents stored in iCloud.

Designing a Document-Based Application

The `UIDocument` class, introduced in iOS 5.0, plays the principal role in document-based applications for iOS. It is an abstract base class—meaning that for you to have something useful, you must create a subclass of it that is tailored to the needs of your application. The application-specific code that you add to the subclass augments what `UIDocument` makes possible: efficient behavior of documents in a mobile environment integrated with iCloud storage.

Why Create a Document-Based Application?

When you have an idea for any application and sit down to design it, you must evaluate a multitude of options. What style of application should you use (master-detail, page-based utility, OpenGL game, and so on)? Will the application do its own drawing, will it respond to gestures or touch events, and will it incorporate audiovisual assets? What will the data model be—for example, should the application use Core Data? The decision whether to make your application document-based might seem to complicate that series of choices, but it really boils down to how you anticipate people will use your application.

Document-based applications are ideal, even necessary, when users expect to enter and edit content in a visual container and store that content under a name they specify. Each container of information—a document—is unique. You are no doubt familiar with several types of document-based applications found in both desktop and mobile systems. To name a few, there are word processors, spreadsheets, and drawing programs. With the iCloud technology, you have an even more compelling reason to make your application document-based. Your users, for example, could create a document using your application on a Mac OS X desktop system and then later, without having to do any syncing or copying, edit that document on an iPad using the iOS version of your application. This feature gives them an additional incentive to buy your application.

When you adopt the `UIDocument` approach to document-based applications, your application gets a lot of behavior "for free" or with minimal coding effort on your part.

- **Integration with iCloud storage.** `UIDocument` coordinates all reading and writing of document data from and to iCloud storage. It does this by adopting the `NSFilePresenter` protocol and by calling methods of the `NSFileCoordinator` and `NSFileManager` classes.
- **Background writing and reading of document data.** If your application reads and writes a document synchronously, it can become momentarily unresponsive. `UIDocument` avoids this problem by reading and writing document data asynchronously on a background dispatch queue.

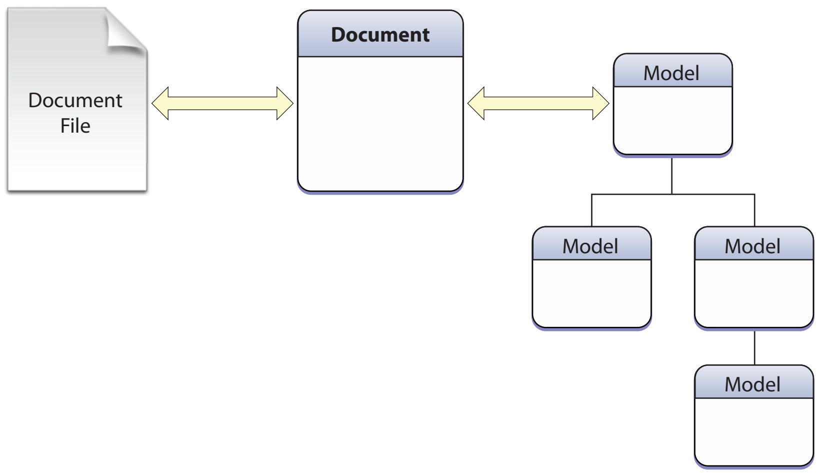
- **Saveless model.** In the saveless model, a user of a document-based application rarely has to save documents explicitly; `UIDocument` saves document data automatically at intervals optimized for what the user is doing with the document. You make your document-based application "saveless" by implementing undo management or change tracking (see "Change Tracking and Undo Operations" (page 48) for information).
- **Safe saving.** `UIDocument` saves document data safely; consequently, if some external event interrupts a save operation, document data won't be left in an inconsistent state. `UIDocument` implements safe saving by first writing the newest version of a document to a temporary file and then replacing the current document file with it.
- **Support for handling errors and version conflicts.** When `UIDocument` detects a conflict between different versions of a document, it notifies the application. The application can then attempt to resolve the conflict itself, or it can ask the user to pick the desired document version. `UIDocument` also notifies an application when a save operation does not succeed. "Managing the Life Cycle of a Document" (page 31) describes how to observe these notifications; "Resolving Document Version Conflicts" (page 51) discusses strategies for dealing with document-version conflicts.

A Document in iOS

Although the broad definition of a document is a "container of information," that container can be viewed in several ways. To a user, a document is the text, images, shapes, and other forms of information that he or she creates, edits, and saves under a unique name. A document can also refer to a document file: the persistent, on-disk representation of document data. And a document can mean a `UIDocument` object that represents and manages the in-memory representation of that same data.

A document object also plays a key role in converting document data between its representation on disk to its representation in memory. It cooperates with the `UIDocument` class in the writing of document data to a file and the reading of that data. For writing, a document typically provides a snapshot of data that can be written to the document file; for reading, it receives data and initializes the document's model objects with it. A document, in a sense, is a conduit between data stored in a file and the internal representation of that data. Figure 1-1 illustrates these relationships.

Figure 1-1 Document file, document object, and model objects managed by the document object



A Document in iCloud Storage

In iCloud, files reside locally in a container directory that is associated with an application. That directory has an internal structure that includes, most importantly, a `Documents` subdirectory. In the iCloud scheme, files and file packages written to the `Documents` subdirectory are considered document files—even if they don't originate from document-based applications. Files that are written to other locations in the container directory are considered data files.

Document objects are file presenters because the `UIDocument` class adopts the `NSFilePresenter` protocol. A file presenter is used together with `NSFileCoordinator` objects to coordinate access to a file or directory among the objects of an application and between an application and other processes. A file presenter is involved in other clients' access of the same presented file or directory.

When users request iCloud storage for a document, a document-based application should save document files (including file packages) to the `Documents` subdirectory of the iCloud container directory. See "Managing the Life Cycle of a Document" (page 31) for specifics. For more information about iCloud mobile containers, see "iCloud Storage" in *iOS App Programming Guide*.

Properties of a UIDocument Object

A document object has several defining properties, most of which relate to its role as a manager of document data. These properties are declared by the `UIDocument` class.

- **File URL.** A document must have a location where it can be stored, whether that location is in the local file system or in iCloud storage. The `fileURL` property identifies this location. When you create a `UIDocument` object, you must specify a file URL as the parameter of the `initWithFileURL:` initializer method of `UIDocument`.
- **Document name.** `UIDocument` obtains a default document name from the filename component of the file URL and stores it in the `localizedName` property. You can override the getter method of this property to provide a custom, localized document name.

> **Note** The display name of a document does not have to correspond to the filename of the document. In addition, users should not be required to specify a document name. See "Creating a New Document" (page 32) for more information on this subject.

- **File type.** The file type is a Uniform Type Identifier (UTI) derived from the extension of the file URL and assigned to the `fileType` property. For more information, see "How iOS Identifies Your Application's Documents" (page 21).
- **Modification date.** The date the document file was last modified. This value is stored in the `fileModificationDate` property. It can be use for (among other things) resolving document-version conflicts.
- **Document state.** A document is in one of several possible states during its runtime life. These states can indicate, for example, that there was an error in saving the document or that there are conflicting document versions. `UIDocument` stores the current document state in the `documentState` property. For information about observing changes in document state, see "Monitoring Document-State Changes and Handling Errors" (page 43).

Design Considerations for Document-Based Applications

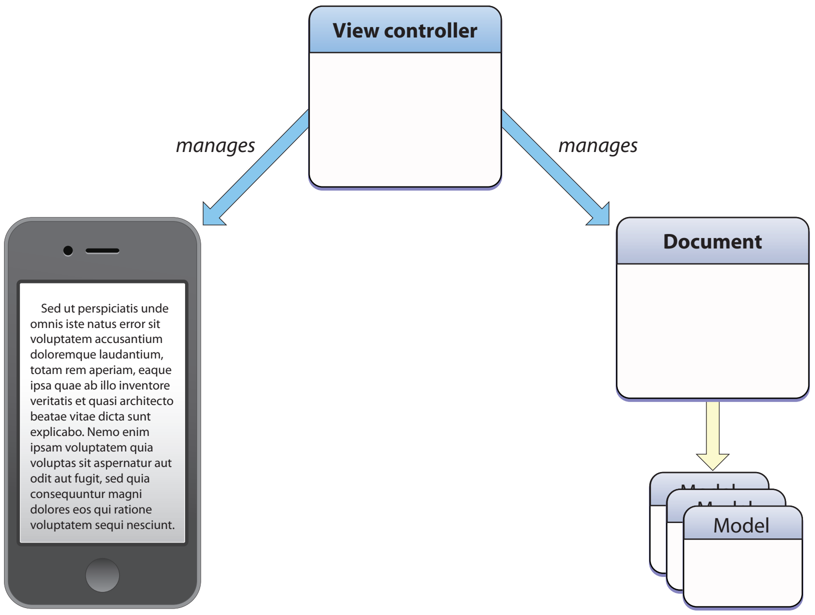
Before you write a line of code for your document-based application, it's worth your while to think about a few design issues.

Defining Object Relationships

As with all applications, you should devise an overall design for the objects of your document-based application that is based on the Model-View-Controller design pattern (MVC). Although the following MVC design for documents is recommended, you are free to come up with your own object-relationship designs.

In MVC terms, a document is a model controller; it "owns" and manages the model objects that represent the document's content. The document object itself is owned and managed by a view controller. The view controller, by definition, also manages a view that presents the content managed by the document object. It is thus a mediating controller in this network of relationships, obtaining the data it needs to present in the view from the document object and passing data entered or changed by users to the document object. Figure 1-2 depicts these object relationships.

Figure 1-2 A view controller manages both a document object and the view that presents document data


View controller
manages
manages
Document
Model
Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt.

Just as a view controller embeds its view, a view controller might embed the document object as a declared property. When your application instantiates the view controller, it initializes it with the document object or with the document's file URL (from which the view controller itself can create the document object).

Of course, a document-based application will have other view controllers (with their views) and possibly other model objects. To get an idea of what other view controllers might be required, see "Designing the User Interface" (page 14).

Designing the User Interface

Neither UIKit nor the developer tools provide any support for the user interface of a document-based application. For instance, there is no `UIDocumentView` class or `UIDocumentViewController` class. Rather than regret the absence, take it as an opportunity to create a user interface for your document-based application that makes it stand out from the competition.

Nonetheless, all document-based applications should enable their users to do certain things that require user interface elements; these include the following:

- Viewing and editing a document
- Creating a new document
- Selecting a document from a list of documents owned by the application
- Opening, closing, and deleting a selected document
- Putting a selected document in iCloud storage (and removing a selected document from iCloud storage)
- Indicating error conditions, including document-version conflicts
- Undoing and redoing changes (recommended but not required)

When you design your application, be sure to include the view controllers, views, and controls that are necessary to implement these actions.

Choosing Types, Formats, and Strategies for Document Data

When designing the data model for your document-based application, it's critical that you ask and answer the following questions:

What is the type of my document?

A document must have a document type, a collection of information that characterizes the document and associates it with an application. A document type has a name, an icon (optional), a handler rank, and a Uniform Type Identifier (UTI) that is paired with one or more filename extensions. For an application to edit the same document in iOS and in Mac OS X, the document-type information should be consistent.

"Creating and Configuring the Project" (page 20) explains how to specify document types in a document-based iOS application. See *Uniform Type Identifiers Overview* for a description of UTIs and *Uniform Type Identifiers Reference* for descriptions of common Uniform Type Identifiers.

How should I represent the document data that is written to a file?

You have basically three choices:

- **Core Data (database).** Core Data is a technology and framework for object-graph management and persistence. It can use the built-in SQLite data library as a database system. The `UIManagedDocument` class, a subclass of `UIDocument`, is intended for document-based applications that use Core Data.
- **Supported object formats.** `UIDocument` supports the `NSData` and `NSFileWrapper` objects as native types for document-data representation. `NSData` is intended for flat files, and `NSFileWrapper` is intended for file packages, which are directories that iOS treats as a single file. If you give `UIDocument` one of these objects, it saves it to the file system without further involvement on your part.

- **Custom object formats.** If you want the document data written to a file to be of a type other than `NSData` or `NSFileWrapper`, you can write it to the file yourself in an override of a specific `UIDocument` method. Keep in mind that your code will have to duplicate what `UIDocument` does for you, and so you must deal with greater complexity and a greater possibility of error. See *UIDocument Class Reference* for further information.

Why would I want to store my document data in a database instead of a file?

If the data set managed by your document object is primarily a large object graph but your application uses only a subsection of that graph at any time, then `UIManagedDocument` and Core Data are a good option. Core Data brings many benefits to a document-based application:

- Incremental reading and writing of document data
- Support for undo and redo operations
- Automatic support for resolving document-version conflicts
- Data compatibility for cross-platform applications

The "downside" to Core Data is that it is a complex technology; it takes time to become familiar with its concepts, classes, and techniques. To learn more, read *Core Data Programming Guide* and *UIManagedDocument Class Reference*.

Which of the supported object formats (NSData or NSFileWrapper) is best for my application?

Whether you use an `NSData` or `NSFileWrapper` object for document data depends on how complex that data is and on whether part of your document's model-object graph can be written out separately to a file. For example, if your document data is plain text and nothing else, then `NSData` is a suitable container for it. If, however, the data has multiple components—for example, text *and* images—use `NSFileWrapper` methods to create a file package for the data.

File wrapper objects—and file packages which they represent—offer some advantages over binary data objects.

- File wrappers support incremental saving. In contrast with a single binary data object, if a file wrapper contains your document data—for example, text and an image—and the text changes, only the file containing the text has to be written out to disk. This capability results in better performance.
- File wrappers (and file packages) make it easier to version documents. A file package, for example, can contain a property list file that holds version information and other metadata about a document.

"Storing Document Data in a File Package" (page 28) discusses (and illustrates) how you use an `NSFileWrapper` object to represent document data in a form that can be written out as a file package.

Can I archive my model-object graph and have that NSData object written to the document file?

Yes, that is possible, but some of the caveats previously mentioned apply. If any part of your document's model-object graph can be written to a separate file, don't archive that part. Instead, store the `NSData` archive and the partitioned-out items as separate components of a file package.

What if my document file is very large?

If the data stored in a document file can be large, you might have to write and read the data incrementally to ensure a good user experience. There are a few approaches you might take:

- Use `UIManagedDocument`. Remember, Core Data gives you incremental reading and writing "for free."
- Store the separable components of your document data in a file package.
- Override lower-level methods of `UIDocument` and do the incremental reading and writing of data yourself. (See *UIDocument Class Reference* for particulars.) You should always use an API for incremental reading and writing that is appropriate to the data type; for example, the AV Foundation framework has methods for incremental reading of large multimedia files.

What should I be thinking about if I want my documents to be editable on both platforms?

Your application will have a competitive advantage if your users can create and edit documents both on iOS mobile devices (iPhone, iPad, and iPad touch) and on Mac OS X desktop systems. But for this to be possible, the format of the document data on both platforms should be compatible. Some important considerations for document-data compatibility are:

- Some technologies are available on one platform but not the other. For example, if you're using RTF as a document format in Mac OS X, that format won't work in iOS because its text system doesn't support rich text.
- The corresponding classes in each platform are not compatible. This is especially true with colors (`UIColor` and `NSColor`), images (`UIImage` and `NSImage`), and Bezier paths (`UIBezierPath` and `NSBezierPath`). `NSColor` objects, for example, are defined in terms of a color space (`NSColorSpace`), but there is no color space class in UIKit.

  If you define a document property with one of these classes, you must devise a way (when preparing the document data for writing) to convert the property into a form that can be accurately reconstituted on the other platform. One way to do this is to "drop down" to a lower-level framework that is shared by both platforms. For example, `UIColor` defines a `CIColor` property holding a Core Image object representing the color; on the Mac OS X side, your application can create an `NSColor` object from the `CIColor` object using the `colorWithCIColor:` class method.
- The default coordinate system for each platform is different, and this difference can affect how content is drawn. (See "Default Coordinate Systems and Drawing in iOS" in *Drawing and Printing Guide for iOS* for a discussion of this topic).

- If you archive a document’s model-object graph, partially or entirely, then you might have to perform platform-sensitive conversions using `NSCoder` methods when you encode and decode the model objects.

Document-Based Application Preflight

For most developers, making a document-based application doesn't require much more effort than making an application that isn't document based. The basic difference is that you must create a custom subclass of `UIDocument` and then manage a document through the phases of its runtime life, including its integration with iCloud storage. This chapter outlines those document-specific tasks performed by most applications and describes the general steps for creating and configuring a document-based application project in iOS.

What You Must Do to Make a Document-Based Application

To create a document-based application, you should complete the following tasks:

- Create a custom subclass of `UIDocument` that provides the UIKit framework with a snapshot of document data and that initializes the document's model objects from the contents of the document file.

  "Creating a Custom Document Object" (page 24) describes the required method overrides and, if document data is to be stored as a file package, explains how to go about using `NSFileWrapper` objects for document data.

- Allow users to create new documents and select and open existing ones. You should also implement the complementary tasks of closing documents and deleting selected documents.

  Note that the follow-on task to creating or opening a document is displaying the document's content in a view.

  "Creating a New Document" (page 32), "Opening and Closing a Document" (page 34), and "Deleting a Document" (page 46) describe the requirements and procedures for these tasks. These discussions also include examples of managing the display of document data.

- Implement undo management or change tracking to enable the automatic saving of document data (saveless model).

  See "Change Tracking and Undo Operations" (page 48) for details.

- Put documents—typically those selected by users—into iCloud storage. Also remove documents from iCloud storage when users request this.

  "Moving Documents to and from iCloud Storage" (page 40) discusses these procedures.

- Observe notifications of changes in document state and, if an error occurs, respond appropriately.

  “Monitoring Document-State Changes and Handling Errors” (page 43) describes the general procedure for doing this.

- If conflicts between different versions of a document occur, notify the user and offer ways to resolve the conflicts.

  “Resolving Document Version Conflicts” (page 51) describes how to notify users of version conflicts and discusses strategies for resolving these conflicts.

You can also add extra features to your document-based application, such as the capability for printing the document, spell-checking it, or emailing it to others.

If your application has advanced requirements—for example, incremental reading and writing of large document files or dealing with document-data formats other than the supported ones—see *UIDocument Class Reference* . All of these advanced tasks involve overriding `UIDocument` methods.

Creating and Configuring the Project

When you create the Xcode project for your document-based application, choose a suitable template. (Note that there is no template specifically for document-based applications.) Generally, you want the first view of the application to be one in which users can choose existing documents and create new ones. Because the Master-Detail Xcode template is suitable for this purpose, it is used in the code examples throughout this document. This template gives you an initial table view for the iPhone and a split view for the iPad. Your project should use a storyboard, so be sure to select this option.

> **Note** If you are using Core Data to manage document data, be sure to select the “Use Core Data” option in the New Project assistant.

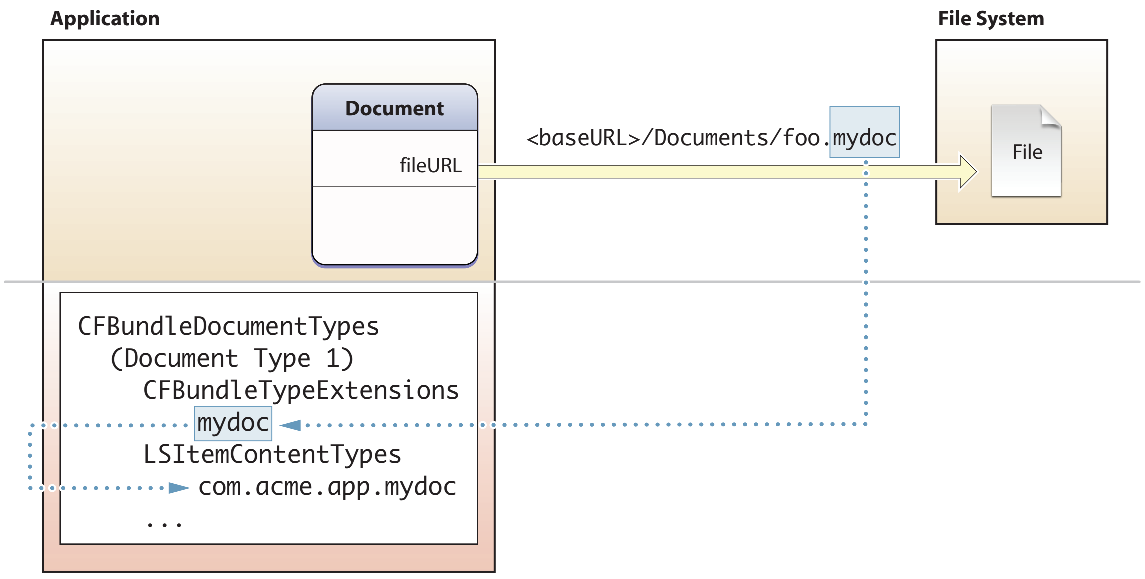
Based on the design for your application (see “Designing a Document-Based Application” (page 10)), create the view-controller subclasses that your application requires. All you need are minimally declared header and source files at this point. Then, create the user interface of your application in the project storyboard (or storyboards, if yours is a universal application); associate your custom view controllers with the view-controller placeholders in the storyboard.

Next, you need to configure the project for documents by specifying, in the Xcode target settings, the type or types of the documents that the application knows about.

How iOS Identifies Your Application's Documents

The most important attribute of a document object in iOS is its file URL (`fileURL`). The file URL is important, among other reasons, because it tells iOS which applications understand the document's format. The file URL ends with an extension (for example, `html`) and this extension is matched with a Uniform Type Identifier (for example, `public.html`). The Uniform Type Identifier (UTI) is the principal identifier of document type. Using the extension, `UIDocument` looks up the document-type UTI (as shown in Figure 2-1) and assigns it to the `fileType` property. Unlike document-based applications in OSX, those in iOS don't need to associate the `UIDocument` subclass with the document type.

Figure 2-1 IOS looking up a document UTI from the extension of its file URL



A document-type UTI can be defined by the system; see "System-Declared Uniform Type Identifiers" in *Uniform Type Identifiers Reference* for a list of these common identifiers. A document-based application can also define its own proprietary UTI for its documents (and often does). If it does declare a custom UTI, it must also export that UTI to make the operating system aware of it.

Declare a Document Type

To declare a document type in Xcode, start by clicking the Add button in the target's Info settings and choose Add Document Type from the pop-up menu. Click the triangle next to Untitled to disclose the property fields and add the properties in Table 2-1.

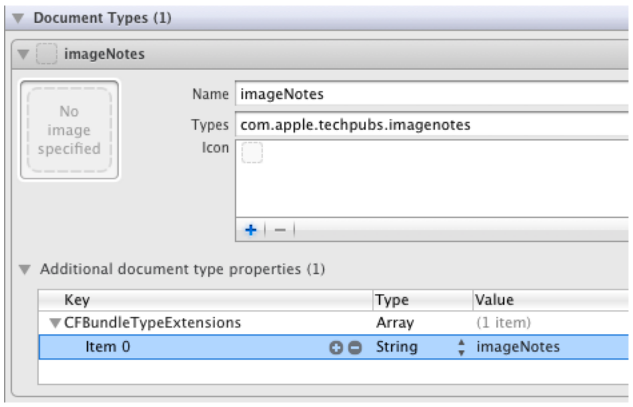
Table 2-1 Properties for defining a document type (`CFBundleDocumentTypes`)

Key	Xcode field	Value and comments
`LSItemContentTypes`	Types	An array of UTI strings. Only one is typically specified per document type.
`CFBundleTypeName`	Name	An optional name for the document type.
`CFBundleType-IconFiles`	Icon	An array of paths to icon image files in the application bundle.
`CFBundleType-Extensions`	In "Additional document type properties" table.	An array of filename extensions paired with the document UTI.
`LSHandlerRank`	In "Additional document type properties" table.	`Owner`, `Alternate`, `None`. (Typically `Owner`).
`LSTypeIsPackage`	In "Additional document type properties" table.	If document data is stored in a file package, set this property to `YES`. Otherwise omit.

For more information about these keys, see "CFBundleDocumentTypes" in *Information Property List Key Reference* .

When you have finished entering the properties for a document type, The Document Types area of Xcode should look similar to the example in Figure 2-2.

Figure 2-2 Specification of a document type in Xcode



Name: imageNotes
Types: com.apple.techpubs.imagenotes

Key	Type	Value
CFBundleTypeExtensions	Array	(1 item)
Item 0	String	imageNotes

An application could have multiple types of documents—for example, a word-processing application could have a type for regular (blank) documents and another type for pre-formatted documents. For each type, you need to go through the procedure given above .

Exporting the Document UTI

If you define a custom UTI for your documents, you must also export it. To export a document type in Xcode, start by clicking the Add button in the target's Info settings and choose Add Exported UTI from the pop-up menu. Click the triangle next to Untitled to disclose the property fields and add the properties in Table 2-2.

Note The procedure for declaring and exporting UTIs is described in "Declaring New Uniform Type Identifiers" in *Uniform Type Identifiers Overview*.

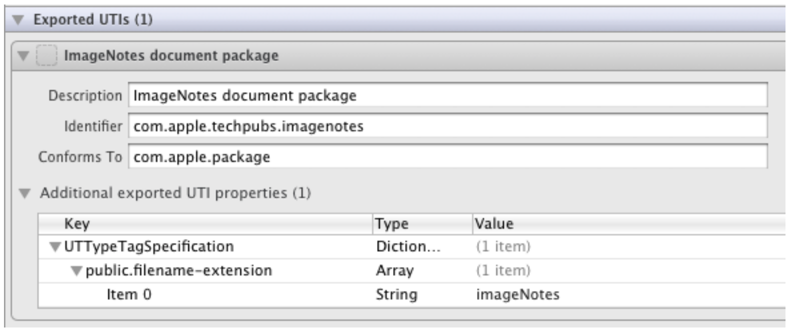
Table 2-2 Properties for exporting a document UTI (`UTExportedTypeDeclarations`)

Key	Xcode field	Value and comments
`UITypeIdentifier`	Identifier	The custom document UTI, a string.
`UTTypeConformsTo`	Conforms to	The UTI that the custom document UTI conforms to. If the data representation is a file package, specify `com.apple.package`.
`UTTypeDescription`	Description	Description of the exported type (optional).
`UTTypeTag-Specification`	In "Additional exported UTI properties" table.	Create an array named `public.filename-extension`. Then add as items all extensions of the document file.

For more information about these keys, see "CFBundleDocumentTypes" in *Information Property List Key Reference*.

When you have finished entering the properties for a document type, The Exported UTIs area of Xcode should look similar to the example in Figure 2-3.

Figure 2-3 Exporting a custom document UTI in Xcode



Exported UTIs (1)

ImageNotes document package

Description: ImageNotes document package
Identifier: com.apple.techpubs.imagenotes
Conforms To: com.apple.package

Additional exported UTI properties (1)

Key	Type	Value
▼UTTypeTagSpecification	Diction...	(1 item)
▼public.filename-extension	Array	(1 item)
Item 0	String	imageNotes

Creating a Custom Document Object

A document-based application must have an instance of a subclass of `UIDocument` that represents and manages document data. This chapter discusses the method overrides most applications need to make and offers suggestions for overriding other methods. For the core override points—the `loadFromContents:ofType:error:` and `contentsForType:error:` methods—examples are given for both `NSData` and `NSFileWrapper` as types of document data read from and written to a file. "Storing Document Data in a File Package" (page 28) further explains how to use file-wrapper objects for document data.

You can override methods of `UIDocument` other than the ones discussed in this chapter to read and write document data for particular purposes—for example, to write and read document data incrementally. However, these more advanced overrides have more complex requirements and should be avoided if possible. See *UIDocument Class Reference* for discussions of these overrides.

Declaring the Document Class Interface

In Xcode, add new Objective-C source and header files to your project, naming them appropriately (suggestion: work "Document" into the name). In the header file, change the superclass to `UIDocument` and add properties to hold the document data. In Listing 3-1, the document data is plain text, so an `NSString` property is all that is needed to hold it. (The text will be converted to an `NSData` object that is written to the document file.)

Listing 3-1 Document subclass declarations (`NSData`)

```
@interface MyDocument : UIDocument {
}
@property(nonatomic, strong) NSString *documentText;
@end
```

Listing 3-2 illustrates set of declarations for another application that uses an `NSFileWrapper` object as the data-representation type. (The code examples in the chapter alternate between the two applications.) Not only is there a property to hold the file-wrapper object, there are properties to hold the text and image components of the represented file package.

Listing 3-2 Document subclass declarations (`NSFileWrapper`)

```
@interface ImageNotesDocument : UIDocument

@property (nonatomic, strong) NSString* text;
@property (nonatomic, strong) UIImage* image;
@property (nonatomic, strong) NSFileWrapper *fileWrapper;

@property (nonatomic, weak) id <ImageNotesDocumentDelegate> delegate;
@end

@protocol ImageNotesDocumentDelegate <NSObject>
-(void)noteDocumentContentsUpdated:(ImageNotesDocument*)noteDocument;
@end
```

This code shows additional declarations for a delegate and the protocol that it adopts. The document object's view controller makes itself the delegate of the document object (and adopts the protocol) so that it can be notified (via `noteDocumentContentsUpdated:` messages) of modifications to the document file. Listing 3-4 (page 26) shows when and how the `noteDocumentContentsUpdated:` message is sent.

Loading Document Data

When an application opens a document (at the user's request), `UIDocument` reads the contents of the document file and calls the `loadFromContents:ofType:error:` method, passing in an object encapsulating the document data. That object can be an `NSData` object or an `NSFileWrapper` object. In your override of the method, initialize the document's internal data structures (that is, its model objects) from the contents of the passed-in object.

The example in Listing 3-3 creates a string from the passed-in `NSData` object and assigns it to the `documentText` property. It also informs its delegate (in this case, the document's view controller) of the updated document contents by invoking a protocol method. The motivation behind this delegation message is that the `loadFromContents:ofType:error:` method is called not only as the result of opening a document, but also because of iCloud updates and reversion operations (`revertToContentsOfURL:completionHandler:`).

Listing 3-3 Loading a document's data (`NSData`)

```
- (BOOL)loadFromContents:(id)contents ofType:(NSString *)typeName error:(NSError
**)outError {
    if ([contents length] > 0) {
        self.documentText = [[NSString alloc] initWithData:(NSData *)contents
encoding:NSUTF8StringEncoding];
    } else {
        self.documentText = @"";
    }
    if ([_delegate respondsToSelector:@selector(noteDocumentContentsUpdated:)]) {
        [_delegate noteDocumentContentsUpdated:self];
    }
    return YES;
}
```

If you have more than one document type, check the `typeName` parameter; a different document type might affect how your code handles the document-data object. If your code experiences an error that prevents it from loading document data, return `NO`; optionally, you can return by reference an `NSError` object that describes the error.

The example in Listing 3-4 handles document data in the form of an `NSFileWrapper` object. It simply assigns this object to its property.

Listing 3-4 Loading a document's data (`NSFileWrapper`)

```
-(BOOL)loadFromContents:(id)contents ofType:(NSString *)typeName error:(NSError
**)outError {
    self.fileWrapper = (NSFileWrapper *)contents;
    if ([_delegate respondsToSelector:@selector(noteDocumentContentsUpdated:)]) {
        [_delegate noteDocumentContentsUpdated:self];
    }
    return YES;
}
```

In this code, the method implementation does not extract the text and image components of the file wrapper and assign them to their properties. That is done lazily in the getter methods for the `text` and `image` properties.

Supplying a Snapshot of Document Data

When a document is closed or when it is automatically saved, `UIDocument` sends the document object a `contentsForType:error:` message. You must override this method to return a snapshot of the document's data to `UIDocument`, which then writes it to the document file. Listing 3-5 gives an example of returning a snapshot of document data in the form of an `NSData` object.

Listing 3-5 Returning a snapshot of document data (`NSData`)

```
- (id)contentsForType:(NSString *)typeName error:(NSError **)outError {
    if (!self.documentText) {
        self.documentText = @"";
    }
    NSData *docData = [self.documentText dataUsingEncoding:NSUTF8StringEncoding
allowLossyConversion:NO];
    return docData;
}
```

If the `documentText` property has not yet been assigned any string value yet, it is assigned an empty string before it's used to create an `NSData` object.

Listing 3-6 shows an implementation of the same method that returns an `NSFileWrapper` object. Basically, if a top-level (directory) file-wrapper object doesn't exist, the code creates it; and if the two contained (regular file) file-wrapper objects do not exist, the code creates them from the values of the `text` and `image` properties. Then, it returns the top-level file wrapper to `UIDocument`, which creates a file package in the file system. See "Storing Document Data in a File Package" (page 28) for a more detailed explanation of file packages and documents.

Listing 3-6 Returning a snapshot of document data (`NSFileWrapper`)

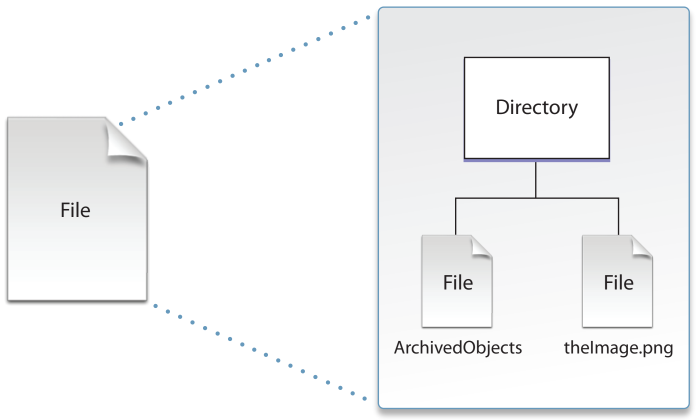
```
-(id)contentsForType:(NSString *)typeName error:(NSError **)outError {

    if (self.fileWrapper == nil) {
       self.fileWrapper = [[NSFileWrapper alloc] initDirectoryWithFileWrappers:nil];
    }
    NSDictionary *fileWrappers = [self.fileWrapper fileWrappers];
    if (([fileWrappers objectForKey:TextFileName] == nil) && (self.text != nil))
{
        NSData *textData = [self.text dataUsingEncoding:TextFileEncoding];
        NSFileWrapper *textFileWrapper = [[NSFileWrapper alloc]
initRegularFileWithContents:textData];
        [textFileWrapper setPreferredFilename:TextFileName];
        [self.fileWrapper addFileWrapper:textFileWrapper];
    }
    if (([fileWrappers objectForKey:ImageFileName] == nil) && (self.image != nil))
 {
        @autoreleasepool {
            NSData *imageData = UIImagePNGRepresentation(self.image);
            NSFileWrapper *imageFileWrapper = [[NSFileWrapper alloc]
initRegularFileWithContents:imageData];
            [imageFileWrapper setPreferredFilename:ImageFileName];
            [self.fileWrapper addFileWrapper:imageFileWrapper];
        }
    }
    return  self.fileWrapper;
}
```

Storing Document Data in a File Package

A file package has an internal structure that is reflected in the methods of the `NSFileWrapper` class. A file wrapper is a runtime representation of a file-system node, which is either a directory, a regular file, or a symbolic link. As shown in Figure 3-1, a file package is a file-system node, typically a directory and its contents, that the operating system treats as a single, opaque entity. It is similar in concept to a bundle.

Figure 3-1 Structure of a file package



You programmatically compose a file package by creating a top-level directory file wrapper and then adding to that container regular files and subdirectories, each represented by other `NSFileWrapper` objects. File wrappers inside the top-level directory should have preferred names associated with them.

With this brief overview in mind, look again at the following lines of code from the `contentsForType:error:` method in Listing 3-6 (page 27). The file package created in this method has two components, a text file and an image file. (The creation of the image file wrapper is not shown in the snippet.)

```
    if (self.fileWrapper == nil) {
       self.fileWrapper = [[NSFileWrapper alloc] initDirectoryWithFileWrappers:nil];
    }
    NSDictionary *fileWrappers = [self.fileWrapper fileWrappers];
    if (([fileWrappers objectForKey:TextFileName] == nil) && (self.text != nil))
{
        NSData *textData = [self.text dataUsingEncoding:TextFileEncoding];
        NSFileWrapper *textFileWrapper = [[NSFileWrapper alloc]
initRegularFileWithContents:textData];
        [textFileWrapper setPreferredFilename:TextFileName];
        [self.fileWrapper addFileWrapper:textFileWrapper];
    }
```

The code creates a top-level directory if it doesn't exist. If a file wrapper doesn't exist for the text file, it creates one from the string contents of the text property. It gives this file wrapper a preferred filename and then adds it to the top-level directory file wrapper.

For more on `NSFileWrapper`, see *NSFileWrapper Class Reference*; also see "Exporting the Document UTI" (page 23) for the required `Info.plist` property for document file packages.

Other Method Overrides You Might Make

There are a few other `UIDocument` overrides that many document-based applications might want to make:

- `disableEditing`...`enableEditing`—`UIDocument` calls the first method when it is unsafe for the user to make changes to document content, such as when there are updates from iCloud or a revert operation is underway. You can implement this method to prevent editing during this period. When editing becomes safe again, `UIDocument` calls the second method.

Note As an alternative to these overrides, you can observe notifications posted when the document state changes and, if the new document state is `UIDocumentStateEditingDisabled`, prevent editing until the document state changes again. More more on this topic, see "Monitoring Document-State Changes and Handling Errors" (page 43).

- `savingFileType`—This method by default returns the value of the `fileType` property. If the current document should be saved under a different file type for any reason, you can override this method to return the replacement file-type UTI. An example (from Mac OS X) is that when an image is added to an RTF file, it should be saved as an RTFD file package.

Managing the Life Cycle of a Document

A document goes through a typical life cycle. A document-based application is responsible for managing its progress through that cycle. As you can see from the following list, most of these life-cycle events are initiated by the user:

- The user first creates a document.
- The user opens an existing document and the application displays it in the document's view or views.
- The user edits the document.
- A user may ask to put a document in iCloud storage or may request the removal of a document from iCloud storage.
- During editing, saving, or other actions, errors or conflicts can happen; the application should learn about these errors and conflicts and either attempt to handle them or inform the user.
- The user closes a selected document.
- The user deletes an existing document.

The following sections discuss the procedures a document-based application must complete for these life-cycle operations.

Setting the Preferred Storage Location for Document Files

All documents of an application are stored either in the local sandbox or in an iCloud container directory. A user should not be able to select individual documents for storage in iCloud.

When an application launches for the first time on a device, it should do the following:

- If iCloud is not configured, ask users if they want to configure it (and, preferably, transfer them to Launch Settings if they want to configure iCloud).
- If iCloud is configured but not enabled for the application, ask users if they want to enable iCloud—in other words, ask if they want all of their documents saved to iCloud. Store the response as a user preference.

Based on this preference, an application writes document files either to the local application sandbox or the iCloud container directory. (For details, see "Moving Documents to and from iCloud Storage" (page 40).) An application should expose a switch in the Settings application that enables users to move documents between local storage and iCloud storage.

Creating a New Document

A document object (that is, an instance of your custom `UIDocument` subclass) must have a file URL that locates the document file either in the local application sandbox or in an iCloud container directory, whichever is the user's preference. In addition, a new document can be given a name. The following discusses guidelines and procedures related to file URLs, document names, and the creation of new documents.

Document Filename Versus Document Name

The `UIDocument` class assumes a correspondence between the filename of a document and the document name (also known the display name). By default, `UIDocument` stores the filename as the value of the `localizedName` property. However, an application should not require a user to provide the document filename or display name when he or she creates a new document.

For your application, you should devise some convention for automatically generating the filenames for your new documents. Some suggestions are:

- Generate a UUID (universally unique identifier) for each document, optionally with an application-specific prefix.
- Generate a timestamp (date and time) for each document, optionally with an application-specific prefix.
- Use a sequential numbering system, for example: "Notes 1", "Notes 2", and so on.

For the document (display) name, you might initially use the document filename if that makes sense (such as with "Notes 1"). Or, if the document contains text and the user enters some text in the document, you might use the first line (or some part of the first line) as the display name. Your application can give users some way to customize the document name after the document has been created.

Composing the File URL and Saving the Document File

You cannot create a document object without a valid file URL. The file URL has three parts of interest: the path to the `Documents` directory in the user's preferred document location, the document filename, and the extension of the document file. You can get a URL representing the path to the `Documents` directory in the local application sandbox through a method such as the one in "Document Filename Versus Document Name."

Listing 4-1 Getting a URL to the application's `Documents` directory in the local sandbox

```
-(NSURL*)localDocumentsDirectoryURL {
    static NSURL *localDocumentsDirectoryURL = nil;
    if (localDocumentsDirectoryURL == nil) {
        NSString *documentsDirectoryPath = [NSSearchPathForDirectoriesInDomains(
NSDocumentDirectory,
            NSUserDomainMask, YES ) objectAtIndex:0];
       localDocumentsDirectoryURL = [NSURL fileURLWithPath:documentsDirectoryPath];
    }
    return localDocumentsDirectoryURL;
}
```

The file extension must be one that you specified for the document type (see "Creating and Configuring the Project" (page 20)). You can declare a global string to represent the extension. For example:

```
static NSString *FileExtension = @"imageNotes";
```

The final part of a document's file URL is the filename component. As "Document Filename Versus Document Name" (page 32) explains, the application should initially generate the document filename according to some convention that makes sense for the application. This generated filename can be used as the document name, or the first line (or part thereof) can be used as the document name. The application can give the user the option of customizing the document name after the document object has been created.

After you concatenate the base URL, the document filename, and the file extension, you can allocate an instance of your custom `UIDocument` subclass and initialize it with the `initWithFileURL:` method, passing in the constructed file URL. The final step in creating a new document is to save it to the preferred document storage location (even though there is no content at this point). As illustrated by "Setting the Preferred Storage Location for Document Files ," you do this by calling the `saveToURL:forSaveOperation:completionHandler:` method on the document object.

Listing 4-2 Saving a new document to the file system

```
-(void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    if (_createFile) {
        [self.document saveToURL:self.document.fileURL
            forSaveOperation:UIDocumentSaveForCreating completionHandler:^(BOOL
success) {
```

```
            if (success)
                _textView.text = self.document.text;
        }];
        _createFile = NO;
    }
    // .....
}
```

The save-operation parameter of the method call should be `UIDocumentSaveForCreating`. The final parameter of the call is a completion hander: a block that is invoked after the save operation concludes. The parameter of the block tells you whether the operation succeeded. If it did succeed, this code assigns the document text to the `text` property of the text view displaying the document content.

Note If you want to save a new document to the application's iCloud container directory, it is recommended that you first save it locally and then call the `NSFileManager` method `setUbiquitous:itemAtURL:destinationURL:error:` to move the document file to iCloud storage. (This call could be made in the completion handler of the `saveToURL:forSaveOperation:completionHandler:` method.) See "Moving Documents to and from iCloud Storage" (page 40) for further information.

Opening and Closing a Document

Opening a document might at first glance seem to be a fairly easy procedure. Your application scans the contents of its `Documents` directory for files having the document's extension and presents those documents to the user for selection. However, when iCloud storage is factored in, things get a bit more complicated. Your application's documents could be in the `Documents` directory of the application sandbox *or* they could be in the `Documents` directory of the iCloud container directory.

Discovering an Application's Documents

To obtain a list of an application's documents in iCloud storage, run a metadata query. A query is an instance of the `NSMetadataQuery` class. After creating a `NSMetadataQuery` object, you give it a scope and a predicate. For iCloud storage, the scope should be `NSMetadataQueryUbiquitousDocumentsScope`. A predicate is an `NSPredicate` object that, in this case, constrains a search by filename extension. Before you start running the query, register to observe the `NSMetadataQueryDidFinishGatheringNotification` and `NSMetadataQueryDidUpdateNotification` notifications. The method accepting delivery of these notifications processes the results of the query.

Listing 4-3 illustrates how you set up and run a metadata query to get the list of application documents in the iCloud mobile container. The method first tests the user's preferred storage location for documents (the `documentsInCloud` property). If that location is the mobile container, it runs a metadata query. If the location is the application sandbox, it iterates through the contents of the application's `Documents` directory to get the names and locations of all local document files.

Listing 4-3 Getting the locations of documents stored locally and in iCloud storage

```
-(void)viewDidLoad {
    [super viewDidLoad];
    // set up Add and Edit navigation items here....

   if (self.documentsInCloud) {
        _query = [[NSMetadataQuery alloc] init];
        [_query setSearchScopes:[NSArray
arrayWithObjects:NSMetadataQueryUbiquitousDocumentsScope, nil]];
        [_query setPredicate:[NSPredicate predicateWithFormat:@"%K LIKE '*.txt'",
 NSMetadataItemFSNameKey]];
        NSNotificationCenter* notificationCenter = [NSNotificationCenter
defaultCenter];
        [notificationCenter addObserver:self selector:@selector(fileListReceived)
            name:NSMetadataQueryDidFinishGatheringNotification object:nil];
        [notificationCenter addObserver:self selector:@selector(fileListReceived)
            name:NSMetadataQueryDidUpdateNotification object:nil];
        [_query startQuery];
    } else {
        NSArray* localDocuments = [[NSFileManager defaultManager]
contentsOfDirectoryAtPath:
            [self.documentsDir path] error:nil];
        for (NSString* document in localDocuments) {
            [_fileList addObject:[[[FileRepresentation alloc]
initWithFileName:[document lastPathComponent]
                url:[NSURL fileURLWithPath:[[self.documentsDir path]
                stringByAppendingPathComponent:document]]] autorelease]];
        }
    }
}
```

In this example, the predicate format is `@"%K LIKE '*.txt’"`, which means to return all filenames (the `NSMetadataItemFSNameKey` key) that have a extension of `txt`, the file extension of this application's document files.

After the initial query concludes, and again if there are subsequent updates, the notification method specified in Listing 4-3 (`fileListReceived`) is invoked again. Listing 4-4 shows this method's implementation. If query updates arrive after the user has made a selection, the code also tracks the current selection.

Listing 4-4 Collecting information about documents in iCloud storage

```
-(void)fileListReceived {

 NSString* selectedFileName=nil;
    NSInteger newSelectionRow = [self.tableView indexPathForSelectedRow].row;
    if (newSelectionRow != NSNotFound) {
        selectedFileName = [[_fileList objectAtIndex:newSelectionRow] fileName];
    }
    [_fileList removeAllObjects];
    NSArray* queryResults = [_query results];
    for (NSMetadataItem* result in queryResults) {
        NSString* fileName = [result valueForAttribute:NSMetadataItemFSNameKey];
        if (selectedFileName && [selectedFileName isEqualToString:fileName]) {
            newSelectionRow = [_fileList count];
        }
       [_fileList addObject:[[[FileRepresentation alloc] initWithFileName:fileName
            url:[result valueForAttribute:NSMetadataItemURLKey]] autorelease]];
    }
    [self.tableView reloadData];
    if (newSelectionRow != NSNotFound) {
        NSIndexPath* selectionPath = [NSIndexPath indexPathForRow:newSelectionRow
 inSection:0];
        [self.tableView selectRowAtIndexPath:selectionPath animated:NO
scrollPosition:UITableViewScrollPositionNone];
    }
}
```

The example application now has an array (`_fileList`) of custom model objects that encapsulate the name and file URL of each of the application's documents. (`FileRepresentation` is the custom class of those objects.) The root view controller populates a plain table view with the document names

Note You should leave metadata queries running only while your application is in the foreground. You should stop the queries when your application moves to the background.

Downloading Document Files from iCloud

When you run a metadata query to learn about an application's iCloud documents, the query results are placeholder items (`NSMetadataItem` objects) for document files. The items contain metadata about the file, such as its URL and its modification date. The document file is not in the iCloud container directory.

The actual data for a document is not downloaded until one of the following happens:

- Your application attempts to open or access the file, such as by calling `openWithCompletionHandler:`.
- Your application calls the `NSFileManager` method `startDownloadingUbiquitousItemAtURL:error:` to download the data explicitly.

Because downloading large document files from iCloud might result in a perceptible delay in displaying the document data, you should indicate to the user that the download has begun (for example, show "loading" or "updating") and that the file is not currently accessible. Remove this indication when the download has completed.

Opening a Document

The sample document-based application lists known documents in a table view. When the user taps a listed document to open it, `UITableView` invokes the `tableView:didSelectRowAtIndexPath:` method of its delegate. The implementation of this method, shown in Listing 4-5, is typical for the navigation pattern: The root view controller allocates the next view controller in the sequence—in this case, the view controller presenting document data—and initializes with essential data—in this case, the document's file URL. Based on whether the device idiom is iPad or iPhone (or iPhone touch), the root view controller adds the view controller to the split view or pushes it on the navigation controller's stack.

Listing 4-5 Responding to a request to open a document

```
-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath
```

```
{
    [self selectFileAtIndexPath:indexPath create:NO];
}

-(void)selectFileAtIndexPath:(NSIndexPath*)indexPath create:(BOOL)create
{
    NSArray* fileList = indexPath.section == 0 ? _localFileList :
_ubiquitousFileList;
    DetailViewController* detailViewController = [[DetailViewController alloc]
        initWithFileURL:[[fileList objectAtIndex:indexPath.row] url]
createNewFile:create];

    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPad)
 {
        self.splitViewController.viewControllers =
            [NSArray arrayWithObjects:self.navigationController,
detailViewController, nil];
    }
    else {
        [self.navigationController pushViewController:detailViewController
animated:YES];
    }
    [detailViewController release];
}
```

In its initializer method (not shown), the document's view controller (`DetailViewController` in the example) allocates an instance of the `UIDocument` subclass and initializes it by calling the `initWithFileURL:` method, passing in the file URL. It assigns the newly created document object to a `document` property.

The final step in opening a document is to call the `openWithCompletionHandler:` method on the `UIDocument` object; the document's view controller in our sample application calls this method in `viewWillAppear:`, as shown in Listing 4-6. The code checks the document state to verify that the document is closed before attempting to open it—there's no need to open an already opened document.

Listing 4-6 Opening a document

```
-(void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    if (_createFile) {
        [self.document saveToURL:self.document.fileURL
forSaveOperation:UIDocumentSaveForCreating
            completionHandler:^(BOOL success) {
                _textView.text = self.document.text;
        }];
        _createFile = NO;
    }
    else {
        if (self.document.documentState & UIDocumentStateClosed) {
            [self.document openWithCompletionHandler:nil];
        }
    }
}
```

When `openWithCompletionHandler:` is called, `UIDocument` reads data from the document file, and the document object itself creates its model objects from the data. At the conclusion of this sequence of actions, the completion handler of the `openWithCompletionHandler:` method is executed. Although the view controller in the example does not implement the completion block, the completion handler is sometimes used to assign the document data to the document's view or views for display. (To recall what `DetailViewController` does instead to update document views, see Listing 3-4 (page 26) and accompanying text.)

Closing a Document

To close a document, send a `closeWithCompletionHandler:` method to the document object. This method saves the document data, if necessary, and then executes the completion handler in its sole parameter.

A good time to close a document is when the document's view controller is dismissed, such as when the user taps the back button. Before the view controller's view disappears, the `viewWillDisappear:` method is invoked. Your view controller subclass can override this method in order to call `closeWithCompletionHandler:` on the document object, as shown in Listing 4-7.

Listing 4-7 Closing a document

```
-(void)viewDidDisappear:(BOOL)animated {
    [super viewDidDisappear:animated];
    [self.document closeWithCompletionHandler:nil];
}
```

Moving Documents to and from iCloud Storage

As noted in "Setting the Preferred Storage Location for Document Files " (page 31), an application should give its users the option of storing all documents in the local file system (the application sandbox) or in iCloud (the container directory). It stores this option as a user preference and refers to this preference when saving and opening documents. When the user changes the preference, the application should move all document files in the application sandbox to iCloud or move all files in the other direction, depending on the nature of the change.

Getting the Location of the iCloud Container Directory

When you move a document file from local storage to the `Documents` subdirectory of the iCloud container directory, its filename is unchanged. The only part of the file-URL path that is different is the part leading up to `Documents`. To get that part of the path, you need to call the `NSFileManager` method `URLForUbiquityContainerIdentifier:`, passing in a valid iCloud container identifier. You can copy your application's container identifier—a concatenation of team ID and application bundle ID, separated by a period—from the Identifier field of the target's Summary view in Xcode. (This is the applications primary container identifier; an application can have additional container directories.) It might be a good idea to declare a string constant for the container identifier, as in this example:

```
static NSString *UbiquityContainerIdentifier =
@"A93A5CM278.com.acme.document.billabong";
```

The two methods in Listing 4-8 get the iCloud container identifier and append "/Documents" to it.

Listing 4-8 Getting the iCloud container directory URL

```
-(NSURL*)ubiquitousContainerURL {
    return [[NSFileManager defaultManager]
URLForUbiquityContainerIdentifier:UbiquityContainerIdentifier];
}


-(NSURL*)ubiquitousDocumentsDirectoryURL {
    return [[self ubiquitousContainerURL] URLByAppendingPathComponent:@"Documents"];
}
```

Note For an example of getting the base URL for the application sandbox, see "Composing the File URL and Saving the Document File" (page 32).

Moving a Document to iCloud Storage

Programmatically, you put a document in iCloud storage by calling the `NSFileManager` method `setUbiquitous:itemAtURL:destinationURL:error:`. This method requires the fille URL of the document file in the application sandbox (source URL) and the destination file URL of the document file in the application's iCloud container directory. The first parameter takes a Boolean value, which should be `YES`.

Important You should not call `setUbiquitous:itemAtURL:destinationURL:error:` from your application's main thread, especially if the document is not closed. Because this method performs a coordinated write operation on the specified file, calling this method from the main thread can trigger a deadlock with any file presenter monitoring the file. (In addition, this method executing on the main thread can take an indeterminate amount of time to complete.) Instead, call the method in a block running in a dispatch queue other than the main-thread queue. You can always message your main thread after the call finishes to update the rest of your application's data structures.

The method in Listing 4-9 illustrates how to move a document file from an application sandbox to iCloud storage. In the sample application, when the user's preferred storage location (iCloud or local) changes, this method is called for every document file in the application sandbox. There are roughly three parts to this method:

- Compose the source URL and the destination URL.
- On a secondary dispatch queue: Call the `setUbiquitous:itemAtURL:destinationURL:error:` method and cache the result, a Boolean value (`success`) that indicates whether the document file successfully moved to the iCloud container directory.
- On the main dispatch queue: If the call succeeds, update the document's model objects and its presentation of those objects; if the call does not succeed, log the error (or otherwise handle it).

Listing 4-9 Moving a document file to iCloud storage from local storage

```
- (void)moveFileToiCloud:(FileRepresentation *)fileToMove {
    NSURL *sourceURL = fileToMove.url;
    NSString *destinationFileName = fileToMove.fileName;
    NSURL *destinationURL = [self.documentsDir
URLByAppendingPathComponent:destinationFileName];


    dispatch_queue_t q_default;
    q_default = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_async(q_default, ^(void) {
        NSFileManager *fileManager = [[[NSFileManager alloc] init] autorelease];
        NSError *error = nil;
        BOOL success = [fileManager setUbiquitous:YES itemAtURL:sourceURL
            destinationURL:destinationURL error:&error];
        dispatch_queue_t q_main = dispatch_get_main_queue();
        dispatch_async(q_main, ^(void) {
            if (success) {
                FileRepresentation *fileRepresentation = [[FileRepresentation
alloc]
                    initWithFileName:fileToMove.fileName url:destinationURL];
                [_fileList removeObject:fileToMove];
                [_fileList addObject:fileRepresentation];
                NSLog(@"moved file to cloud: %@", fileRepresentation);
            }
            if (!success) {
                 NSLog(@"Couldn't move file to iCloud: %@", fileToMove);
            }
        });
    });
}
```

Removing a Document from iCloud Storage

To move a document file from an iCloud container directory to the `Documents` directory of the application sandbox, follow the same procedure described in "Moving a Document to iCloud Storage" (page 41), except switch the source URL (now the document file in the iCloud container directory) and the destination URL (now the document file in the application sandbox). In addition, the first parameter of the `setUbiquitous:itemAtURL:destinationURL:error:` method should now be `NO`. Listing 4-10 shows a method implementing this procedure; it is called for each file in the iCloud container directory, moving it to the application sandbox.

Listing 4-10 Moving a document file from iCloud storage to local storage

```
- (void)moveFileToLocal:(FileRepresentation *)fileToMove {
    NSURL *sourceURL = fileToMove.url;
    NSString *destinationFileName = fileToMove.fileName;
    NSURL *destinationURL = [self.documentsDir
URLByAppendingPathComponent:destinationFileName];


    dispatch_queue_t q_default;
    q_default = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_async(q_default, ^(void) {
        NSFileManager *fileManager = [[[NSFileManager alloc] init] autorelease];
        NSError *error = nil;
        BOOL success = [fileManager setUbiquitous:NO itemAtURL:sourceURL
destinationURL:destinationURL
            error:&error];
        dispatch_queue_t q_main = dispatch_get_main_queue();
        dispatch_async(q_main, ^(void) {
            if (success) {
                FileRepresentation *fileRepresentation = [[FileRepresentation
alloc]
                    initWithFileName:fileToMove.fileName url:destinationURL];
                [_fileList removeObject:fileToMove];
                [_fileList addObject:fileRepresentation];
                NSLog(@"moved file to local storage: %@", fileRepresentation);
            }
            if (!success) {
                NSLog(@"Couldn't move file to local storage: %@", fileToMove);
            }
        });
    });
}
```

Monitoring Document-State Changes and Handling Errors

A document can go through different states during its runtime life. A state can tell you whether a document is experiencing an error, a version conflict, or some other condition that is not normal. `UIDocument` declares constants (of type `UIDocumentState`) to represent document states and sets the `documentState` property with one of these constants when a change is a document's state occurs. Table 4-1 describes the state constants.

Table 4-1 `UIDocumentState` constants

Document state constant	What it means
`UIDocumentStateNormal`	The document is open and is experiencing no conflicts or other problems.
`UIDocumentStateClosed`	The document is closed. A document is in this state if `UIDocument` cannot open a document, in which case document properties might not be valid.
`UIDocumentStateInConflict`	There are versions of the document that are in conflict.
`UIDocumentStateSavingError`	An error prevents `UIDocument` from saving the document.
`UIDocumentStateEditingDisabled`	It is not currently safe to allow users to edit the document.

`UIDocument` also posts a notification of type `UIDocumentStateChangedNotification` when a change in document state occurs. Your application should observe this notification and respond appropriately. The initializer method of the document's view controller is a good place to add an observer, as shown in Listing 4-11. The observer in this case is the view controller.

Listing 4-11 Adding an observer of the `UIDocumentStateChangedNotification` notification

```
-(id)initWithFileURL:(NSURL*)url createNewFile:(BOOL)createNewFile {
    NSString* nibName = [[UIDevice currentDevice] userInterfaceIdiom] ==
        UIUserInterfaceIdiomPad ? @"DetailViewController_iPad" :
@"DetailViewController_iPhone";
    self = [super initWithNibName:nibName bundle:nil];
    if (self) {
        _document = [[ImageNotesDocument alloc] initWithFileURL:url];
        // other code here....
        [[NSNotificationCenter defaultCenter] addObserver:self
            selector:@selector(documentStateChanged)
            name:UIDocumentStateChangedNotification object:_document];
    }
    return self;
}
```

Be sure to remove the observer from the notification center in the class's `dealloc` method.

When the document's state changes, `UIDocument` posts the `UIDocumentStateChangedNotification` notification, and the notification center delivers it by invoking the notification method (`documentStateChanged` in the example). In Listing 4-12, the observing view controller gets the current state from the `documentState` property and evaluates it. If the state is `UIDocumentStateEditingDisabled`, it hides the keyboard. If there are conflicts between different versions of the document (`UIDocumentStateInConflict`), it displays a Show Conflicts button in the document view's toolbar. (For detailed information on handling document-version conflicts, see "Resolving Document Version Conflicts" (page 51).)

Listing 4-12 Evaluating the current document state

```
-(void)documentStateChanged {
    UIDocumentState state = _document.documentState;
    [_statusView setDocumentState:state];
    if (state & UIDocumentStateEditingDisabled) {
        [_textView resignFirstResponder];
    }
    if (state & UIDocumentStateInConflict) {
        [self showConflictButton];
    }
    else {
        [self hideConflictButton];
        [self dismissModalViewControllerAnimated:YES];
    }
}
```

The notification-handling method also calls a `setDocumentState:` method implemented by a private view class. This method, shown in Listing 4-13, changes other items of the document view's toolbar depending on the document state.

Listing 4-13 Updating a document's user interface to reflect its state

```
-(void)setDocumentState:(UIDocumentState)documentState {
    if (documentState & UIDocumentStateSavingError) {
        self.unsavedLabel.hidden = NO;
        self.circleView.image = [UIImage imageNamed:@"Red"];
    }
    else {
        self.unsavedLabel.hidden = YES;
        if (documentState & UIDocumentStateInConflict) {
            self.circleView.image = [UIImage imageNamed:@"Yellow"];
        }
        else {
            self.circleView.image = [UIImage imageNamed:@"Green"];
        }
    }
}
```

If the document could not be saved (`UIDocumentStateSavingError`), the view controller changes the status indicator to red and displays Unsaved next to it. If there are conflicting document versions, it makes the status indicator yellow (this is in addition to the Show Conflicts button mentioned earlier). Otherwise, the status indicator is green.

Deleting a Document

Just as you want to allow users to create a document, you also want to let them delete selected documents. Deletion of a document requires you to do three things:

- Remove the document file from storage (either from the local sandbox or the iCloud container directory).
- Remove the model objects used to represent the document data in memory.
- Remove the document data presented in the document view.

When you delete a document from storage, your code should approximate what `UIDocument` does for reading and writing operations. It should perform the deletion asynchronously on a background queue, and it should use file coordination. Listing 4-14 illustrates this procedure. It dispatches a task on a background queue that creates an `NSFileCoordinator` object and calls the `coordinateWritingItemAtURL:options:error:byAccessor:` method on it. The `byAccessor` block of this method calls the `NSFileManager` method for deleting the file, `removeItemAtURL:error:`.

Listing 4-14 Deleting a selected document

```
-(void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle

       forRowAtIndexPath:(NSIndexPath *)indexPath {
```

```
    NSMutableArray* fileList = nil;
    if (indexPath.section == 0) {
        fileList = self.localFileList;
    }
    else {
        fileList = self.ubiquitousFileList;
    }
    NSURL* fileURL = [[fileList objectAtIndex:indexPath.row] url];
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
 ^(void) {
        NSFileCoordinator* fileCoordinator = [[NSFileCoordinator alloc]
initWithFilePresenter:nil];
        [fileCoordinator coordinateWritingItemAtURL:fileURL
options:NSFileCoordinatorWritingForDeleting
            error:nil byAccessor:^(NSURL* writingURL) {
            NSFileManager* fileManager = [[NSFileManager alloc] init];
            [fileManager removeItemAtURL:writingURL error:nil];
        }];
    });
    [fileList removeObjectAtIndex:indexPath.row];
    [tableView deleteRowsAtIndexPaths:[[NSArray alloc] initWithObjects:&indexPath
 count:1]
        withRowAnimation:UITableViewRowAnimationLeft];
}
```

In this example, the user triggers the invocation of the method when he or she taps the Delete button in a row while the table view is in editing mode.

Change Tracking and Undo Operations

The saveless-model feature of the `UIDocument` class ensures that document data is automatically saved at frequent intervals, relieving users of the need to explicitly save their documents. `UIDocument` implements much of the behavior for the saveless model, but a document-based application must play its own part to make the feature work.

How UIKit Saves Document Data Automatically

The saveless model implemented by the UIKit framework for documents has two main parts: a mechanism for marking a document as needing to be saved and a variable period for when the framework checks that flag. Periodically, UIKit calls the `hasUnsavedChanges` method of a `UIDocument` object and evaluates the returned value. If the value is `YES`, it saves the document data to the document file. The period between checks of the `hasUnsavedChanges` value varies according to several factors, including the rate of input by the user.

A document-based application sets the value returned by `hasUnsavedChanges` indirectly, either by implementing undo and redo or by tracking changes to the document. Change tracking requires the application to call the `updateChangeCount:` method, passing in `UIDocumentChangeDone` (a constant of type `UIDocumentChangeKind`). When an application registers an undo action and then sends `undo` or `redo` messages to the document's undo manager, `UIDocument` calls `updateChangeCount:` on its behalf.

Because giving users the ability to undo and redo changes can be a differentiating feature, that approach is recommended for most applications.

Implementing Undo and Redo

You can implement undo and redo operations in your application by following the procedures and recommendations in *Undo Architecture* . Note that `UIDocument` defines an `undoManager` property. You can get the default `NSUndoManager` object by accessing this property, or you can assign your own `NSUndoManager` object to it. The undo manager must be associated with the `UIDocument` object through the property in order to enable change tracking and thus automatic saving of document data.

Listing 5-1 illustrates an implementation of undo and redo for a text field.

Listing 5-1 Implementing undo and redo for a text field

```
- (void)textFieldDidEndEditing:(UITextField *)textField {
    self.undoButton.enabled = YES;
    self.redoButton.enabled = YES;

    if (textField.tag == 1) {
        [self setLocationText:textField.text];
    }
    // code for other text fields here....
}

- (void)setLocationText:(NSString *)newText {
    NSString *currentText = _document.location;
    if (newText != currentText) {
        [_document.undoManager registerUndoWithTarget:self
            selector:@selector(setLocationText:)
            object:currentText];
        _document.location = newText;
        self.locationField.text = newText;
    }
}

- (IBAction)handleUndo:(id)sender {
    [_document.undoManager undo];
    if (![_document.undoManager canUndo]) self.undoButton.enabled = NO;
}

- (IBAction)handleRedo:(id)sender {
    [_document.undoManager redo];
    if (![_document.undoManager canRedo]) self.redoButton.enabled = NO;
}
```

Implementing Change Tracking

To implement change tracking instead of implementing undo/redo, call the `updateChangeCount:` method on the `UIDocument` object at the appropriate points in your code. Just as when you register an undo action, it's typically at the point where you update the document's model object with data the user has just entered. The parameter passed in should be a `UIDocumentChangeDone` constant.

Listing 5-2 shows how you might call `updateChangeCount:` from within a `UITextViewDelegate` method that is called when a change is made in a text view.

Listing 5-2 Updating the change count of a document

```
-(void)textViewDidChange:(UITextView *)textView {
    _document.documentText = textView.text;
    [_document updateChangeCount:UIDocumentChangeDone];
}
```

Resolving Document Version Conflicts

In an iCloud world, when a user has installed a document-based application on multiple devices or desktop systems, there can be conflicts between different versions of the same document. Recall that an application updates a document file in the local container directory and those changes are then transmitted—usually immediately—to iCloud. But what if this transmission is not immediate? For example, you edit a document using the Mac OS X version of your application, but you've also edited the same document using the iPad version of the application—and you did so while the device was in Airplane Mode. When you switch off Airplane Mode, the local change to the document is transferred to iCloud. iCloud notices a conflict and notifies the application.

Learning About Document Version Conflicts

As "Monitoring Document-State Changes and Handling Errors" (page 43) describes, your application becomes aware of document-version conflicts by observing the `UIDocumentStateChangedNotification` notification. If the `documentState` property changes to `UIDocumentStateInConflict`, multiple versions of the same document exist. The application is responsible for resolving those conflicts as soon as possible, with or without the user's help.

You learn about the conflicting versions of a document through two class methods of the `NSFileVersion` class. The `currentVersionOfItemAtURL:` method returns an `NSFileVersion` object representing what's referred to as the *current file*; the current file is chosen by iCloud on some basis as the current "conflict winner" and is the same across all devices. By calling the `unresolvedConflictVersionsOfItemAtURL:` method, you get an array of `NSFileVersion` objects; these objects are called *conflict versions*, and each represents an unresolved version conflict for the file located at the specified URL. `NSFileVersion` objects can give you information helpful in resolving conflicts, such as modification dates, localized document names, and localized names of saving computers.

Strategies for Resolving Document Version Conflicts

Your application can follow one of three strategies for resolving document-version conflicts:

- Merge the changes from the conflicting versions.

- Choose one of the document versions based on some pertinent factor, such as the version with the latest modification date.
- Enable the user to view conflicting versions of a document and select the one to use.

Which strategy is best to use depends a lot upon your document data. If you can merge the contents of different document versions without introducing contradictory elements, then follow that strategy. Or choose the document version with the latest modification date if your application doesn't suffer any loss of data as a result.

Generally, you should try to resolve the conflict without involving the user, but for some applications that might not be possible. If an application takes the user-centered approach, it should discreetly inform the user about the version conflict and expose a button or other control that initiates the resolution procedure. "An Example: Letting the User Pick the Version" (page 53) examines the code of an application that lets the user select the document version to use.

How to Tell iOS That a Document Version Conflict Is Resolved

When your application or its users resolve a document version conflict by picking a version of a document, your application should complete the following steps:

- If the chosen version is a conflict version, replace the current document file with the conflict-version document file.

  To to this, call the `replaceItemAtURL:options:error:` method on the `NSFileVersion` object representing the version, passing in the document's current-file URL.
- If the chosen version is a conflict version, revert the document so that it displays the new data in the document file

  To do this, call the `UIDocument` method `revertToContentsOfURL:completionHandler:` on the document object, passing in the document's current-file URL.
- Disassociate all conflict versions with the document's file URL.

  To do this, call the `NSFileVersion` class method `removeOtherVersionsOfItemAtURL:error:`, passing in the document's file URL.
- Mark each conflict version as resolved so that iOS doesn't raise it again as a conflicting version.

  To do this, set the `resolved` property of each `NSFileVersion` object representing a conflict version to `YES`. This step should always be done last.

An Example: Letting the User Pick the Version

Our sample document-based application is a simple text editor. It would be difficult for such an application to locate and merge textual differences in conflicting versions of the document, and even if it did, the resulting document might not be what the user wants. The application could pick the document version with the most recent modification date, but then again there's no way to be certain that is the version the user wants. A good conflict-resolution strategy in this case is to let the user, who is most familiar with the document's contents, pick the version she or he wants.

You might recall the code shown in Listing 6-1 from "Monitoring Document-State Changes and Handling Errors" (page 43). This code shows the method of the document's view controller that handles the `UIDocumentStateChangedNotification` notification posted by `UIDocument` when there is a change in document state. If the new document state is `UIDocumentStateInConflict`, the view controller shows a Resolve Conflicts button in a custom status view. (It also sets the color of the status indicator to red.)

Listing 6-1 Detecting a conflict in document versions

```
-(void)documentStateChanged {
    UIDocumentState state = _document.documentState;
    [_statusView setDocumentState:state];
    if (state & UIDocumentStateEditingDisabled) {
        [_textView resignFirstResponder];
    }

    if (state & UIDocumentStateInConflict) {
        [self showConflictButton];           // <------ Shows "Resolve Conflicts"
 button
    }
    else {
        [self hideConflictButton];
        [self dismissModalViewControllerAnimated:YES];
    }
}
```

When the user taps the button, UIKit invokes the method in Listing 6-2. This method displays modally the view of a custom conflict-resolver view controller.

Listing 6-2 Showing the user interface for resolving document version conflicts

```
-(void)conflictButtonPushed
{
    ConflictResolverViewController* conflictResolver =
[[ConflictResolverViewController alloc]
        initWithURL:_document.fileURL delegate:self];
    [self presentViewController:conflictResolver animated:YES completion:nil];
    [conflictResolver release];
}
```

The `ConflictResolverViewController` object creates a page view controller (`UIPageViewController` object) that allows user to page between, and examine, the current-file document and each conflict-version document. In the tool bar of each document view is a Select Version button. If the user taps that button, one of the two custom delegation methods shown in Listing 6-3 is called, depending on whether the chosen document is the current-file document or a conflict-version document.

Listing 6-3 Resolving a document version conflict

```
-(void)conflictResolver:(ConflictResolverViewController *)conflictResolver
       didResolveWithFileVersion:(NSFileVersion *)fileVersion {
    [self dismissViewControllerAnimated:YES completion:nil];
    [fileVersion replaceItemAtURL:_document.fileURL options:0 error:nil];
    [NSFileVersion removeOtherVersionsOfItemAtURL:_document.fileURL error:nil];
    [_document revertToContentsOfURL:_document.fileURL completionHandler:nil];
    NSArray* conflictVersions = [NSFileVersion
unresolvedConflictVersionsOfItemAtURL:_document.fileURL];
    for (NSFileVersion* fileVersion in conflictVersions) {
        fileVersion.resolved = YES;
    }
}


-(void)conflictResolverDidResolveWithCurrentVersion:(ConflictResolverViewController*)conflictResolver
 {
    [self dismissViewControllerAnimated:YES completion:nil];
    [NSFileVersion removeOtherVersionsOfItemAtURL:_document.fileURL error:nil];
    NSArray* conflictVersions = [NSFileVersion
unresolvedConflictVersionsOfItemAtURL:_document.fileURL];
```

```
    for (NSFileVersion* fileVersion in conflictVersions) {
        fileVersion.resolved = YES;
    }
}
```

These methods illustrate the steps described in "How to Tell iOS That a Document Version Conflict Is Resolved" (page 52). If the chosen document is a conflict version, the delegate calls `replaceItemAtURL:options:error:` on the passed-in `NSFileVersion` object to replace the document file in the iCloud container directory with the chosen document. The delegate then enumerates the array containing `NSFileVersion` objects representing all conflict versions of the document and sets the `resolved` property of each object to `YES`. It then asks `NSFileVersion` to remove all other conflict versions of the document associated with the document's file URL and calls `revertToContentsOfURL:completionHandler:` to revert the displayed document to the new contents of the document file.

The second delegation method, invoked when the current document file is selected, is much simpler. It sets the `resolved` property of all `NSFileVersion` objects representing conflict versions to `YES` and removes all conflict versions associated with the document file URL.

Document Revision History

This table describes the changes to *Document-Based App Programming Guide for iOS* .

Date	Notes
2012-01-09	Changed the operator used in the predicate of metadata query searches to LIKE.
2011-10-12	New document that explains how to create an iOS application whose documents are integrated with iCloud storage.