LEARNING iCloud DATA MANAGEMENT

A Hands-On Guide to Structuring Data for iOS and OS X

JESSE FEILER

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Learning iCloud Data Management

Addison-Wesley Learning Series

Visit informit.com/learningseries for a complete list of available publications.

The Addison-Wesley Learning Series is a collection of hands-on programming guides that help you quickly learn a new technology or language so you can apply what you've learned right away.

Each title comes with sample code for the application or applications built in the text. This code is fully annotated and can be reused in your own projects with no strings attached. Many chapters end with a series of exercises to encourage you to reexamine what you have just learned, and to tweak or adjust the code as a way of learning.

Titles in this series take a simple approach: they get you going right away and leave you with the ability to walk off and build your own application and apply the language or technology to whatever you are working on.

Learning iCloud Data Management

A Hands-On Guide to Structuring Data for iOS and OS X

Jesse Feiler

Addison-Wesley

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

Library of Congress Cataloging-in-Publication Data

Feiler, Jesse.
Learning iCloud data management : a hands-on guide to structuring data for iOS and OS X / Jesse Feiler.
pages cm
Includes bibliographical references and index.
ISBN 978-0-321-88911-9 (paperback : alkaline paper)
1. iCloud—Handbooks, manuals, etc. 2. Cloud computing—Handbooks, manuals, etc. 3. Database management—Handbooks, manuals, etc. 4. iOS (Electronic resource)—Handbooks, manuals, etc. 5. Mac OS—Handbooks, manuals, etc. I. Title.
QA76.585.F45 2014
004.67'82—dc23

2013043333

ISBN-13: 978-0-321-88911-9
ISBN-10: 0-321-88911-8
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, February 2014

Editor-in-Chief
Mark L. Taub

Senior Acquisitions Editor
Trina MacDonald

Development Editor
Michael Thurston

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Project Editor
Anna Popick

Copy Editor
Carol Lallier

Indexer
Jack Lewis

Proofreader
Anna Popick

Technical Reviewers
Jon Bell
Erik Buck
Rod Strougo

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
Shepherd, Inc.

Contents at a Glance

Contents

This page intentionally left blank

Preface

When Apple announces new products or new versions of its operating systems, there is usually a big press event, and frequently there are lines of people waiting at Apple stores. There's generally a pattern to these announcements. In the case of the operating systems, the major announcements are made at the Apple Worldwide Developers Conference in June. In some years, developer previews of one or both operating systems are made available earlier in the spring. Over the course of the summer, developer releases are made available. Rumors of the availability of the new iPhone begin circulating, and, sometime in the fall, Apple sends invitations to a media event to be held in a week. At that event, a new version of iOS is shown to the public along with a new iPhone. The public release of iOS comes a week later, followed by the availability of the new iPhone. Later (often the following month) the process is repeated for the iPad, Macs, and OS X.

This has been the schedule over the past few years, but there is no guarantee it will be repeated. What is important to note is that there are specific dates for the announcement and release of the products and operating systems. iCloud is a very different matter. Over a number of years, Apple has built a significant hardware and telecommunications support structure to power iCloud and its other network operations. As is the case with many such infrastructures, the details of it are kept confidential. We know the location of some of Apple's data centers because they often require building permits and other public documents and permissions, but they are usually kept out of the public view. There has been no ribbon cutting or turning of a key to launch iCloud—it has been a years-long process (and it will continue for years to come).

In addition to the hardware infrastructure, iCloud has a software component. However, that, too, has been a years-long development process. As you will see in this book, parts of iCloud are implemented in the user interface of the operating systems, and other parts of it are implemented with relatively small changes to existing frameworks and APIs. For developers as well as consumers, public announcements about iCloud have been part of the announcements of new versions of the operating systems as well as of hardware.

In short, iCloud is not a product: it's a pervasive technology and a companywide strategy for Apple. Unlike Apple's hardware and software products, iCloud has no part number and no version. It is part of products across the company.

For that reason, it is not easy to write about iCloud or to learn to develop for it. This book was first envisioned in early 2012, but as it took shape, it became clear that some of the most powerful pieces of iCloud were not yet in place. Rather than rushing out a partial book and relying on the possibility of a revised edition sometime in the future, Trina MacDonald and Addison-Wesley agreed to push back the publication date so as to include the information from WWDC in June 2013, and I'm very grateful to them for doing that.

As you will see, the book culminates in what I call the iCloud Round Trip. In the final chapter, you'll see how to build an iOS app and an OS X app that let you share data via iCloud on both OS X and iOS. Having the tools to be able to implement the Round Trip seems to me to be a good time to publish the book. That's as close to a product launch event as you can get in the world of iCloud.

Who Should Read This Book

This book is written for developers who want to explore iCloud. Because iCloud is implemented in so many areas of the operating systems, you need a bit of familiarity with many parts of Cocoa and Cocoa Touch. As the book presents iCloud, an attempt has been made to at least summarize the various components that it touches. This means that the discussion of a topic such as notifications is at a fairly high level: some people will think "everyone knows that" and other people may think that more details are needed.

The attempt has been to provide a medium road for both experts and novices in the various Cocoa technologies that interact with iCloud. Apple's documentation on developer.apple.com provides the primary resource for more details if you feel you need them. If you hit an area where you feel that you already know the topic, feel free to skip to the details of iCloud. Even among engineers at Apple, there are many areas of Cocoa that they know inside out (and may have written) and other areas with which they're not familiar.

In terms of skills and knowledge, you should have a basic knowledge of Cocoa and/or Cocoa Touch as well as of Xcode. Objective-C is a must for understanding the code. The author's *Sams Teach Yourself Objective-C in 24 Hours* provides an introduction to that topic.

In addition, you should have experience in using iCloud. It is always amazing how many people attempt to develop for a technology that they have not used. There's nothing like hands-on user experience.

Downloading the Example Files

The example files for each chapter that has them can be downloaded from the author's site at http://northcountryconsulting.com and from http://informit.com/title/9780321889119. In addition to the examples, you will find any updates and

corrections on both sites. Some of the downloadable examples contain additional code, such as an iPad interface in addition to the iPhone interface for Chapter 14, "Working with File Wrappers in iCloud."

The files are arranged by chapter, and they represent the code as of the end of the chapter. Thus, in the cases where one chapter builds on the previous chapter's code, download the previous chapter and work through it to add the new chapter's code.

iCloud requires code signing, so you'll see in this book how to set up your projects to accomplish that. Note that the code in this book and in the downloadable code contains code signing that will not work on your computer. You must use your own developer credentials. Rather than leaving the code signing information blank, I have used my own credentials (the password is not provided, and even the developer account name has been changed). This means that the code will not run unless you customize it for your own developer account. This is deliberate and necessary.

The code has been written against Xcode 5.0 and OS X Mavericks (10.9).

How This Book Is Organized

There are four parts to this book.

Part I: Introducing iCloud

The first part provides perspectives on iCloud from the user's point of view and from that of the developer.

- Chapter 1, "Exploring iCloud and Its User Experience": As iCloud has evolved, it has been incorporated into apps such as the iWork suite. You'll see the user interface aspects of iCloud for apps and the operating systems.
- Chapter 2, "Setting Up iCloud for Development": This chapter provides an overview of the API structure of iCloud. It's a roadmap to the rest of the book.

Part II: Using the APIs

This part explores how you use iCloud data that the user enters and maintains. For many users, iCloud plays some role with the storage of their music and with the synchronization of their calendars and contacts. There are APIs that allow developers to tap into this synchronized user data, and they are described in this part of the book. This use of iCloud can reap big payoffs for the developer: the engineers at Apple and the users have done all the work—all you have to do is empower the users to employ their own data in new and imaginative ways.

- Chapter 3, "Introducing the APIs and the First Apps": The simplest part of iCloud consists of the APIs that manage user data. This chapter provides the roadmap to this part of the book.

- Chapter 4, "Working with the AddressBook API for Contacts": The AddressBook API lets developers access and update address book data. This chapter shows you the basics of doing so.
- Chapter 5, "Managing Calendars and Reminders with the Event Kit API": You'll see how to leverage calendars and reminders in this chapter.
- Chapter 6, "Protecting the Privacy of User Data": iCloud brings up many privacy issues that you need to address in your apps. This is user data, and you have to play by the rules described in this chapter.

Part III: Using the Technologies

Various data management technologies and design patterns are integrated with iCloud. Using these technologies can mean that your apps can take the most advantage of iCloud synchronization. These technologies are integrated with iCloud, but they existed long before iCloud came to be. It's the integration that's new.

- Chapter 7, "Introducing Blocks, Threads, and Notifications": This chapter provides a roadmap to the technologies in the context of iCloud. Even if you know the technologies, it's important to review them in the iCloud world.
- Chapter 8, "Using Key-Value Coding (KVC)": Key-value coding has been used in Cocoa for years. It's a very efficient way of storing relatively small amounts of data. And it works very easily for you and your users with iCloud.
- Chapter 9, "Using Preferences, Settings, and Keychains with iCloud": Preferences (OS X) and Settings (iOS) are a special case of key-value coding. This chapter shows how you can add them to your apps so that they apply to all of a user's devices. You'll also see how to exclude certain preferences and settings from iCloud if they don't make sense for a specific device.
- Chapter 10, "Managing Persistent Storage with Core Data": Core Data is the major data persistence tool in Cocoa and Cocoa Touch. This chapter provides a high-level overview. It is followed on by Chapter 17, "Working with Core Data and iCloud."
- Chapter 11, "Using Xcode Project Workspaces for Shared Development": Introduced in Xcode 4, Xcode workspaces make it easy to set up multiple targets within a project and to share certain files among the targets. For example, this will enable you to share a Core Data data model (schema) and its specific managed object classes with an OS X/iOS Round Trip.
- Chapter 12, "Adding Data to Apps with Bundles and Resources": This is one of the most general ways of managing data in apps. It doesn't use iCloud directly, but it may be an appropriate addition to an iCloud app to complement iCloud-synchronized data.

Part IV: Using iCloud Documents and Data

The final part of the book brings together the APIs and technologies in documents and file wrappers. You'll see how to implement them on OS X as well as on iOS. In addition, you'll see how to complete a Round Trip as the documents synchronize across iOS and OS X.

- Chapter 13, "Adding the iCloud Infrastructure": This chapter shows you the basic infrastructure to use with iCloud—the code to establish contact with iCloud, manage changes in iCloud availability, and make iCloud account changes. Note that this is code that will need to be implemented in any of the following chapters. In order to focus on the specific issues of the following chapters in this part of the book, it is not repeated in them.
- Chapter 14, "Working with File Wrappers in iCloud": File wrappers implement a structure akin to packages in the finder: a collection of files that appear to be a single file to the user. They are a very efficient structure to take advantage of iCloud synchronization.
- Chapter 15, "Working with iOS Documents": This chapter provides the iOS document model based on `UIDocument`. You'll see how to monitor changes in your iCloud documents in real time.
- Chapter 16, "Working with OS X Documents": On OS X, Cocoa takes care of the changes in iCloud documents for you, so you have less work to do than in Chapter 15. However, there is still work to be done, and this chapter shows you how to use `NSDocument` to accomplish what is necessary.
- Chapter 17, "Working with Core Data and iCloud": This chapter provides you with the code you'll need to manage Core Data-based apps with iCloud. It builds on Chapter 10.
- Chapter 18, "Completing the Round Trip": Finally, you'll see how to put together a Round Trip. Remember to add the code from Chapter 13 to both of your targets (OS X and iOS).

This page intentionally left blank

Acknowledgments

As always, Carole Jelen at Waterside Productions provided help and guidance in bringing this book to fruition. At Addison-Wesley, Trina MacDonald helped move this book along from idea to publication. Michael Thurston provided excellent editorial advice. The production manager, Julie Nahil, kept things moving along in the very complicated process of creating a technical book. Anna Popick, the freelance project manager, and Carol Lallier, freelance copy editor, contributed mightily to the book's development. The elegant cover design is by Chuti Prasertsith.

Notwithstanding the help of these and many other people, any errors are the author's.

This page intentionally left blank

About the Author

Jesse Feiler is a developer and author. He has been an Apple developer since before it became fashionable. His books include *Sams Teach Yourself Core Data for Mac and iOS in 24 Hours* (Sams Publishing, 2011), *Sams Teach Yourself Objective-C in 24 Hours* (Sams Publishing, 2012), *FileMaker 12 in Depth* (Que Publishing, 2012), and *iWork for Dummies* (Wiley, 2012).

Jesse has written about Objective-C and the Apple frameworks beginning with *Rhapsody Developer's Guide* (Academic Press, 1997) and *Mac OS X Developer's Guide* (Morgan Kaufmann, 2001). His books on Apple technologies such as Cyberdog, OpenDoc, ODF, Bento (in both incarnations), and Apple Guide occupy a special place on the shelf of developer books.

He is the author of Minutes Machine, the meeting management app for iPad, as well as the Saranac River Trail app for iPhone and iPad. They are available on the App Store; more details are available at champlainarts.com.

A native of Washington, DC, Jesse has lived in New York City and currently lives in Plattsburgh, New York, where he serves on the board of the Plattsburgh Public Library and as chair of the Saranac River Trail Advisory Committee.

He can be reached at http://northcountryconsulting.com.

This page intentionally left blank

We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let us know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that we cannot help you with technical problems related to the topic of this book and that due to the high volume of mail we receive, we might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address.

Email: trina.macdonald@pearson.com
Mail: Reader Feedback
Addison-Wesley Learning Series
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at **informit.com/register** for convenient access to any updates, downloads, or errata that might be available for this book.

This page intentionally left blank

Introduction

Cocoa and Cocoa Touch consist of frameworks that contain classes as well as protocols, defined constants, and some other supporting items including dynamic sharable libraries. The most basic frameworks are Foundation, UIKit (Cocoa Touch—iOS) and AppKit (Cocoa—OS X). More specialized frameworks, such as the Core Audio Kit Framework, are used as needed by developers.

iCloud is different. Don't search for an iCloud framework: there is none. Don't even search for an iCloud API. There are a couple iCloud-specific methods, but they are few and far between. In fact, they're very far between in the sense that they are scattered across various classes and frameworks. `URLForPublishingUbiquitousItemAtURL:expirationDate:error:` is part of the `NSFileManager` class (there are seven iCloud-related methods among the 52 methods in this class), while `NSPersistentStoreDidImportUbiquitousContentChangesNotification` is part of the `NSPersistentStoreCoordinator` class (it is one of two iCloud-related notifications in this class).

The implementation of iCloud in this way means that existing apps that don't use iCloud aren't affected. In addition, because iCloud spans multiple devices as well as both operating systems (OS X and iOS), it is hard to imagine how it could have been implemented in a single framework or API.

Along with these few additions to the Cocoa and Cocoa Touch APIs, the implementation of iCloud relies on long-time best practices, which now have been converted to essential practices. Design patterns such as key-value coding that date back to the very early versions of NeXTSTEP have been used for a quarter of a century now, and they are used in new ways in iCloud, although in most cases you don't have to do anything to take advantage of the iCloud functionality.

Core Data, which has long been the most powerful solution to managing an app's persistent data, is deeply integrated with iCloud. However, that integration is largely (but not totally) done behind the scenes. If you don't use iCloud, your existing Core Data code is just fine. Perhaps the most significant impact of iCloud on Core Data is that, in the past, there were two ways of creating a data store that could be distributed with an app. You could place a seed database in the app's bundle, or you could add seed

data programmatically to an empty data store that you create the first time the app launches (or whenever the seed data needs to be recreated). Both techniques have been used for years. The biggest impact that iCloud has on Core Data is that with iCloud, the second technique needs to be used; the first one will not work properly. This is scarcely a major change.

Perhaps the most visible impact of iCloud on developers is the enhancement of entitlements that control what an app can do in its runtime environment. Entitlements implement the new sandboxing rules that come into play with shared documents on iCloud. Explicit entitlements and sandboxing define the functions and capabilities of the operating system that an app will use along with the specific parts of disk storage where the app can write data. They increase the stability and security of both operating systems. They are required on iOS and are optional on OS X. On both operating systems, they are more aggressively implemented. Furthermore, from a developer's point of view, you'll probably be happy to hear that the developer-facing interface for entitlements in Xcode 5 is now vastly changed and dramatically simplified. (Sandboxing is related to iCloud, but they are two separate functionalities.)

The implementation of iCloud has proceeded over several years; in mid-2012, the release of OS X Mountain Lion (10.8) and iOS 6 brought together some of the pieces that had been released over the previous year. In the fall of 2013, OS X Mavericks (10.9) and iOS 7 refined iCloud and expanded its behind-the-scenes tools for developers. If you have not used any of the iWork apps (Numbers, Pages, and Keynote), try one of them on multiple iCloud-enabled devices. They provide the best demonstration of iCloud from the user's point of view.

Actually, that statement is wrong. They provide the best demonstration of iCloud from the user's point of view—until you write *your* iCloud-enabled app.

This page intentionally left blank

2

Setting Up iCloud for Development

In Chapter 1, "Exploring iCloud and Its User Interface," you saw how iCloud looks to users. For many users, it's just a logical way of working, and iCloud really isn't an issue that they think about. For other users who have become used to managing their own files and folders on their various devices and desktops, there can be a significant effort at familiarization—an *unlearning* process. As technology advances, these unlearning events happen from time to time, but in the long run, the old way of doing things is forgotten. You may have had to configure a dial-up modem with bit rates and parity settings, for example. Now, even dial-up modems are automated and managed with handshakes to adjust their own settings. Dial-up modems are (fortunately) becoming artifacts of the past. And reports are surfacing of mystified children who don't know what a computer mouse is in the world of touchscreens that they inhabit.

This chapter looks at iCloud from the developer's perspective. As noted in the Introduction, iCloud isn't a monolithic API or framework that you just plug into your code. It's a collection of additions and modifications to many parts of Cocoa and Cocoa Touch. In this chapter, you'll find a high-level view of those additions and modifications with particular emphasis on setting up iCloud in your app.

iCloud involves synchronizing data across a user's devices, and as soon as you start thinking about sharing data among various devices, you have to consider the security issues involved. Fortunately, the engineers at Apple have done this: iCloud takes advantage of the security mechanisms that are built into the App Store and the Mac App Store. App security has not changed dramatically over time; however, configuring security has been difficult for many developers. In part, this is because it is a relatively complex process that, for most developers, is done relatively infrequently. That combination is a classic recipe for difficulty.

With the introduction of Xcode 5 in 2013, the implementation and setup of app security has changed insofar as the developer interface is involved (the underlying security mechanism is not changed). The changes make it easier for you to set up your app's security, but for many developers, it is a new process. Once again, there is an

unlearning process involved when you use the new and simpler tools. Because iCloud requires security to be in place and because the way in which you implement it is changed, this chapter begins with an introduction to the new, improved, 2013 version of app security.

Managing App Security on iOS and OS X

The heart of the app security system is digitally signing your code with two digital signatures. Both of these signatures are generated by Apple, and each one references the other. (This is a common security mechanism that you can read about on Wikipedia in articles such as "Code Signing," which explains the process.) These signatures will not match if either the one identifying Apple or the one identifying the developer has been altered; in addition, part of the digital signature contains a checksum mechanism that causes the security system to fail if the code has been altered since it was signed.

Part of the complexity arises from the use of these digital signatures. The security for apps is built on a combination of developer.apple.com tools, Xcode tools, and Keychain Access tools. It is important to note that Apple IDs are used to identify people, and there are two categories of people (thus Apple IDs) that come into play: During development, you as a developer have an Apple ID. At runtime, the user's Apple ID comes into play with iCloud.

Identifying Yourself and Your App on developer.apple.com

Along with changes in Xcode 5, during 2013 developer.apple.com was revisited to consolidate the process of managing certificates, identifiers, devices, and provisioning profiles. These are the components of the security system for apps on iOS and OS X. Although the terminology hasn't changed, the layout of developer.apple.com has changed. Furthermore, with Xcode 5, it is easier to manage these security features, but the process is slightly changed.

Here is an overview of the process. It is required for you to set up your app to use iCloud as well as to ultimately distribute it.

After you register on developer.apple.com, go to Certificates, Identifiers & Profiles (currently at the right of developer.apple.com). It handles security for both your iOS and OS X apps. However, you can now do this through Xcode 5 and later: it's much easier there.

- You must identify yourself as a registered developer with a signing identity.
- You must identify your app with an App ID.
- You must identify the devices that you want to use for testing your app during development.
- You must create a provisioning profile that brings together your developer ID, your app ID, and the IDs for your test devices.

Managing Your Developer Signing Identity

A key part of the security mechanism is the certificate that you can download for each signing identity that you create. That is the link between developer.apple.com and Keychain Access on your Mac. The certificate is a portable and secure representation of your signing identity. In developer documentation, you may find the terms used interchangeably.

You can manage your signing identities in Xcode or on developer.apple.com. Xcode 5 introduced the ability to manage one or more Apple IDs for developer accounts, as you can see in Figure 2.1. A given Apple ID does not uniquely identify a specific developer account because a developer can be invited to join one or more development teams. You normally continue to use your developer Apple ID even though you may be working on several teams. Figure 2.1 shows the simplest scenario: a single developer Apple ID working on a single team. For iOS and Mac, one developer can have different roles.

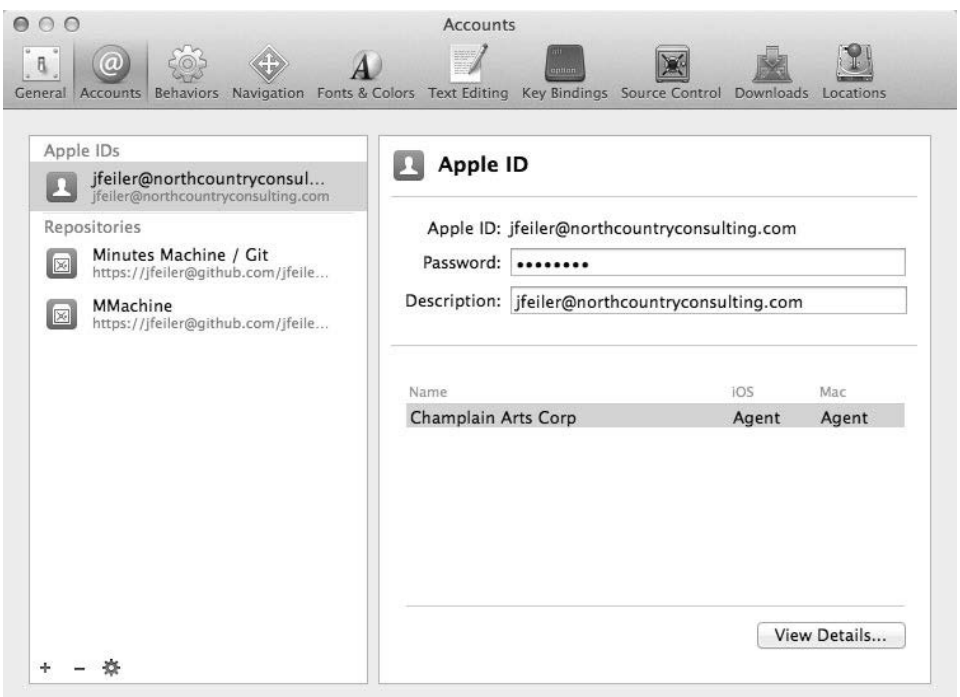

Figure 2.1 Manage your accounts on Xcode

Perhaps the most important point to take away is that you must use your own Apple ID to avoid compromising the security system for apps. Use the tools described on developer.apple.com to manage development teams so that developers can be assigned to the appropriate team without destroying the security structure.

Select the team you want to work with, and click View Details (shown at the lower right of Figure 2.1) to see its signing identities and provisioning profiles, as you see in Figure 2.2.

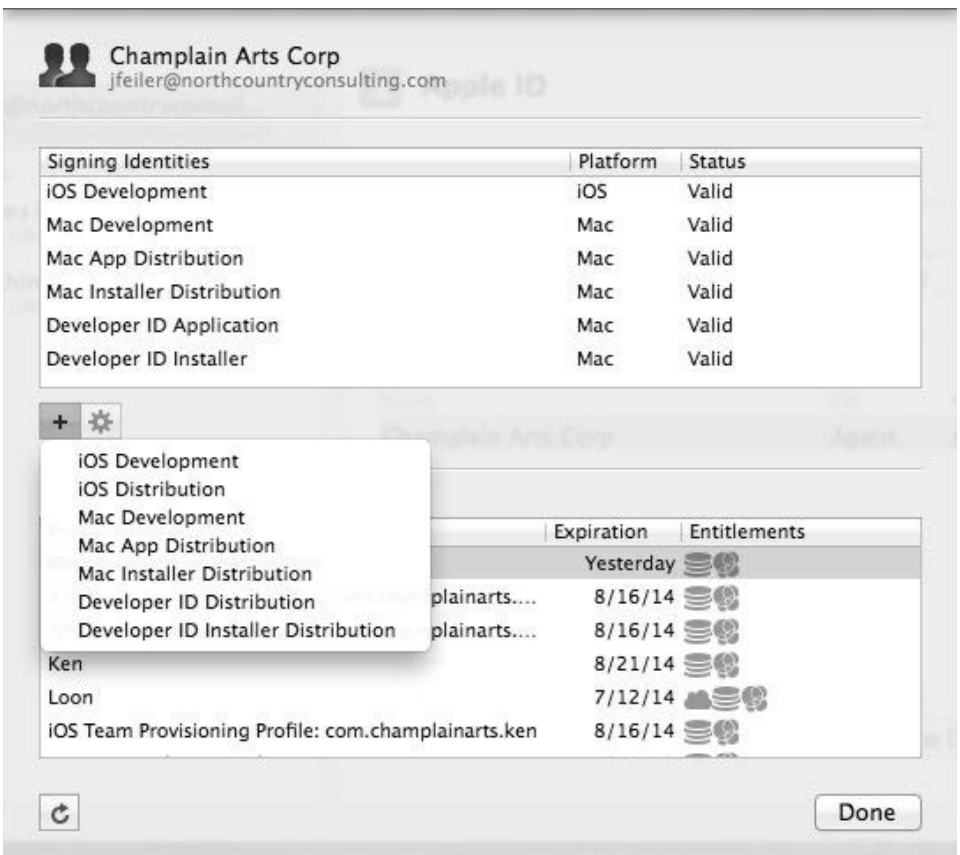

Figure 2.2 Manage certificates in Xcode account preferences

Figure 2.3 shows the list of Mac certificates for a developer on developer.apple.com. Note that each one has a name that you provide, a type that you choose, and an expiration date that is set and enforced by Apple.

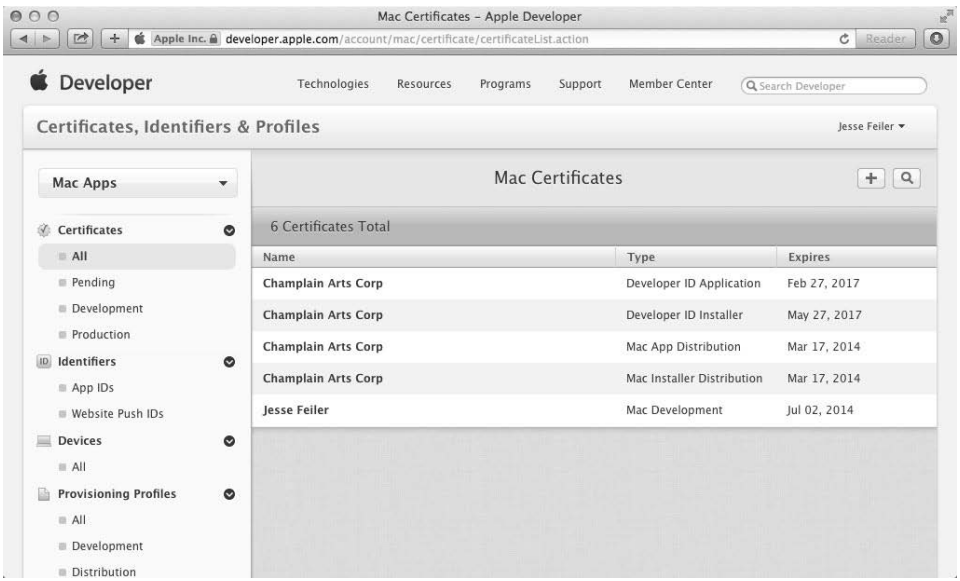

Figure 2.3 Manage certificates on developer.apple.com

Managing Your App ID

Unlike your developer signing identity, which can be edited on developer.apple.com through Xcode accounts, your App ID must be managed for the most part on developer.apple.com. You give the app a name, which can be changed later on if you want. (This is not the name the user sees.) What is important to note is that when you register your App ID, you can enable services that you want to use, such as Game Center, In-App Purchase, Maps, Push Notifications, and most important for this book, iCloud.

Although you cannot create an App ID through Xcode, when you turn on a capability such as iCloud, Xcode offers to update your App ID to add the iCloud capability automatically. (You'll see this demonstrated in Figure 2.9 later in this chapter.)

Managing Your Devices

You can register a number of devices that can be used for testing your apps. (As of this writing, the number is 100.) When you recruit people to test your app, ask them for the UDID (iOS) or UUID (OS X) of the device they want to test with. These people do not need to be registered developers, and sometimes it's a good idea to recruit one

or two testers who are "real people" as opposed to developers. The rules for managing devices are detailed on developer.apple.com. There are limits to how many times you can change the list: this limit prevents you from allowing your app to be installed on a large number of devices without going through the App Store. (Ad hoc distribution is a specific option you may want to explore in this case.) You have one list of devices for your developer account. The provisioning profiles associate them with App IDs.

Managing Provisioning Profiles

Now that you have your App ID and a list of devices, you can create a provisioning profile to combine the two. As you see at the bottom of Figure 2.2, the provisioning profiles are listed by name and expiration date along with the various entitlements associated with them when you look at them in Xcode accounts. When you look at them in developer.apple.com, you'll see that some are marked as being managed by Xcode. For the others, you can specify the devices and the services you want to enable on developer.apple.com.

Thus, at this point, you should have your developer signing identity; your app and its App ID; and your provisioning profile that brings together testing devices, your app ID, and the entitlements or services that it uses. You're ready to start thinking about runtime.

Identifying Your User and Your Ubiquity Container at Runtime

As you saw in Chapter 1, iCloud helps users organize their data by app rather than by file and folder. Users can still work with documents, but those documents aren't on a visible file system in most cases: they're in iCloud. But where is iCloud?

As with all cloud computing, the cloud is an artifact of the Internet and large server farms. If you follow the trail of bits, you see that these server farms synchronize and store data so that it is accessible on an as-needed basis by users. The physical location of the data doesn't matter, and in fact, the actual storage is so often duplicated across servers that there is frequently no single primary data store among the many stores that come into play.

A user's data is available (subject to security constraints) whenever a user accesses the cloud with the appropriate account information and password. That's not the model with iCloud. With iCloud, data is available with the presentation of two identifiers:

- **Apple ID:** This identifies the user.
- **A ubiquity container identifier:** A ubiquity container is the object that holds the iCloud data for the app. It typically is a bundle identifier such as com.champlainarts.colby. It is prefixed automatically by your developer Team ID.

When the user connects to iCloud (usually this happens through the iCloud System Preference panel whenever the user logs in), the Apple ID is made available to all apps that are entitled to use iCloud. The ubiquity container identifier is usually set in the

Capabilities tab of the target in Xcode. As you will see in Chapter 18, "Completing the Round Trip," a shared ubiquity container may have an identifier that does not correspond to an app bundle identifier. In the case of the Round Trip, the two apps are com.champlainarts.ColbyOSX and com.champlainarts.ColbyiOS. They share a ubiquity container called com.champlainarts.Colby. It is the last component of the ubiquity container identifier that shows up in the System Preferences iCloud pane shown previously at the left in Figure 1.5.

With these two pieces of information, you can connect to the appropriate iCloud ubiquity container. That is your first task when your app starts to run.

Looking Inside the iCloud Basics

Bundle identifiers and Apple IDs have been around for a long time, but now they have key roles to play in iCloud. Both of them are needed to gain access to a section of iCloud. This is the implementation of the app-based file structure described in Chapter 1.

You might expect to find standard log-in methods in the iCloud API that enable your app to present an Apple ID and a bundle ID to iCloud in order to gain access to the data. That's not how it happens. Remember that there is no explicit iCloud API; beyond that, the notion of logging in to iCloud for an app isn't what happens. (Users do log in to iCloud—often automatically with their settings in the iCloud pane of System Preferences.)

Your app interacts directly with a local copy of the iCloud data for the user and the app. This copy of the iCloud data for the user and app is stored locally in a ubiquity container. The ubiquity containers are stored on the local device, and their contents are synchronized by the local OS and its interaction with iCloud. Just as is the case with any other local data access, you can read and write as necessary, and you can expect (and even check on) the results of those read and write statements.

Because you are not reading and writing to the iCloud data directly in most cases, you can't expect the changes that you have made to the local ubiquity data to be propagated to iCloud immediately. If you want to get into naming things, iCloud is an asynchronous and declarative implementation of cloud technology.

The key components of iCloud are

- Apple ID
- Bundle identifier
- Entitlements and capabilities
- Ubiquity container

The following sections cover the basics of what you need to know about them.

Declarative Programming

As is the case with more and more software today, iCloud relies heavily on *declarative* programming techniques. Declarative programming is distinguished from other styles that specify what happens and, frequently, in what order. (Common names for that style are *imperative*, *functional*, and *procedural* programming.) Declarative programming simply describes what should be done without specifying a control flow.

You find examples of declarative programming throughout OS X and iOS with more examples showing up with each new iteration of the Cocoa frameworks. Blocks, for example, allow you to specify code that is executed for each element of an enumerator or on completion of some task, but you do not hang around waiting for that trigger to occur. You define the block and then send it off (often as a parameter of a method), and it is executed at the appropriate time. You are out of the traffic-cop business, and, not coincidentally, multithreading at the system level is much easier for the OS to manage in your absence.

If you're not familiar with declarative programming or are still not comfortable with it, explore the topic online (Wikipedia is a great place to start for this type of research).

Apple ID

We're now looking at iCloud runtime behavior. The Apple ID discussed here is the user's Apple ID.

An Apple ID uniquely identifies . . . something. It started in 2000 as an account name on Apple's early Internet service, iTools, which provided free email accounts as <accountname>@mac.com. Over time, <accountname>@mac.com became <accountname>@me.com (MobileMe) and then <accountname>@icloud.com. With the advent of the iTunes store, customers used an Apple ID for their purchases. The email account name served as the first Apple IDs, but, particularly after Apple began charging for email accounts, Apple IDs no longer consisted of me.com or mac.com addresses. Every Apple ID did have to have an email address associated with it (for verification if for no other reason) and, for purchases in iTunes Store, a credit card number.

The idea that an Apple ID uniquely identifies an individual person has long gone away. An Apple ID has a name, a password, an email address, an optional rescue email address (in case the primary address is unreachable), and, if used for purchases, it may have a credit card associated with it. Apple suggests that people not share Apple IDs, but we know that sometimes a family or even a small business will share one.

Apple suggests that people may like to have one Apple ID to identify themselves to iTunes and another to identify themselves for other purposes such as iCloud, FaceTime, and the like. Developers often have one or more Apple IDs for their personal life and another for their developer account. iBook authors need their own Apple ID, so a developer who is also an iBook author needs two right there.

There is a unique identifier underneath all the attributes, so email address, name, password, and credit card can all be changed without creating a new Apple ID. Every iOS device requires that the user has an Apple ID in order to gain access to downloads of the operating system as well as any purchased apps or music.

On OS X, although the installation process encourages it, you do not need an Apple ID. If you want to use iCloud, you do need an Apple ID. Apple has recognized the proliferation of multi-Apple ID individuals in OS X Mountain Lion (10.8) and later versions of OS X. Figure 2.4 shows part of the Users & Groups pane in System Preferences.

Figure 2.4 You can have multiple Apple IDs on OS X.

If you click Change, you see a list of the Apple IDs you have associated with this account. You can add or delete some and create a new one, as shown in Figure 2.5.

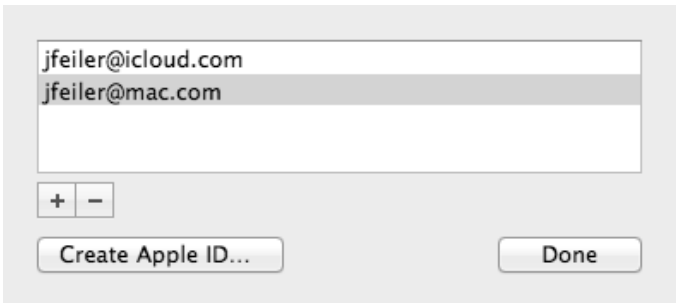

Figure 2.5 Switching Apple IDs on OS X

Most of the time, people don't pay attention to their Apple ID when they set up a device beyond checking that their email works (if it uses the Apple ID). However, for ongoing support of your iCloud app, remember (and let your tech support people remember) that the Apple ID is a critical part of iCloud access. If someone in an office uses an OS X account for business and another for personal matters, the iCloud documents created under those two OS X accounts may be using different Apple IDs. A Pages document under one account will not be shared with the other, although you can do so with sharing commands implemented in Pages and other apps.

The Apple ID that a user has used to sign into iCloud is available to the operating system at runtime, and that is how the Apple ID part of the iCloud authentication takes place: you don't do anything.

Bundle Identifier

The bundle identifier is set in your app's target settings in Xcode (in the General tab of the target). As you step through the process of creating a new project, you are asked for information, including the product name and the company identifier. You provide the product name, and the company identifier is editable (it actually is sticky—you start with the last company identifier you used).

> **Note**
>
> The management of bundle identifiers, product names, and targets, as described in this section, has been a matter of concern for a number of developers over the years. You can find many references on the Internet to what is going on. Unfortunately, some of them (particularly those from several years ago) are misleading. The information in this section is current as of Xcode 5, which is the version released with iOS 7 and OS X Mavericks (10.9).

The bundle identifier that Xcode starts with is the combination of the company identifier (which is usually your reverse domain name) and the product name, as in `com.yourcompany.yourproductname`.

If you look at the Info tab of your project, as shown in Figure 2.6, you'll see that the bundle identifier is set to your product name. The product name is also used as the target name, so you begin with identical values for your target and the last component of your bundle identifier. You can change your target name in the left side of the project editor: just double-click and type in a new name. You'll see that the last component of the bundle name also changes, because, as you see in Figure 2.6, it is picking up the product name.

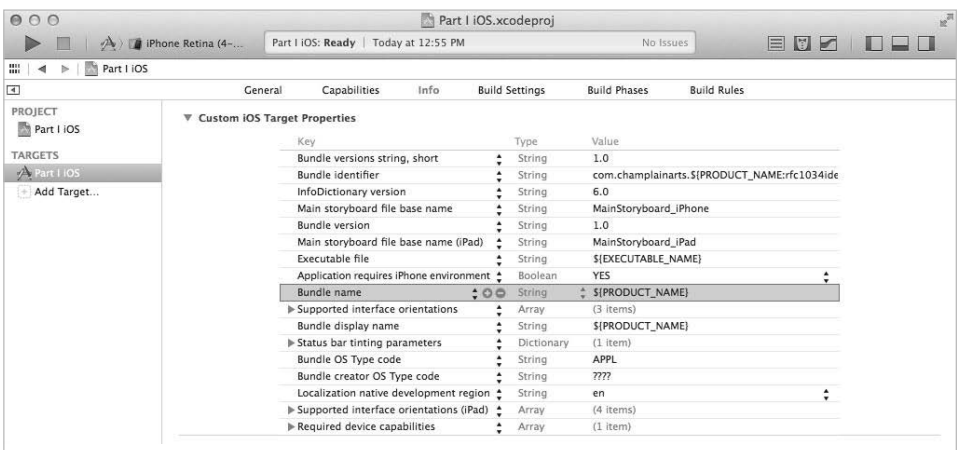

Figure 2.6 Editing the bundle identifier in Info

However, you can edit the bundle identifier itself in the General tab. As initially set up, it is set to `com.yourcompany.${PRODUCT_NAME:rfc1034identifier}`. If you trace through the various settings, you'll see that product name (in the Packaging section of Build Settings) is set to `$(TARGET_NAME)`. This means that if you change the target name, the product name will change, and because it's used as the last component in the bundle identifier, that, too, will change. Anywhere along the line, you can double-click to edit the setting. If you change Product Name to be MyProject instead of `$(TARGET_NAME)`, you will change the product name, and indirectly, the last component of the bundle identifier. Generally, the best place to edit a bundle identifier is in the General tab of the project itself rather than in the Info tab. That is because the Info tab sets up the naming structure with placeholders such as `$(TARGET_NAME)` and the General tab lets you type in the actual name that you want to use, which overrides the placeholders.

Most of the time, the default settings are fine, and you don't have to worry about them. However, they come into play with iCloud when you need a ubiquity container that is shared among several apps. (Perhaps most commonly, one is a Mac app and the other is an iOS app.)

Entitlements and Capabilities

Entitlements specify what your app can do. The Capabilities tab shown in Figure 2.7 lets you configure the capabilities and the related entitlements and other settings. As you can see, there's a simple switch for each capability—iCloud, Game Center, Passbook, In-App Purchase, and more (still more are likely to come in the future).

> **Note**
>
> The Capabilities tab is new in Xcode 5. It replaces previous entitlements configurations that were different for iOS and OS X. The interface shown here is for iOS apps, but it is almost identical for OS X apps.

If a given capability is off, turning it on will also open the disclosure triangle to show you what additional steps you and/or Xcode must take, as you see in Figure 2.7.

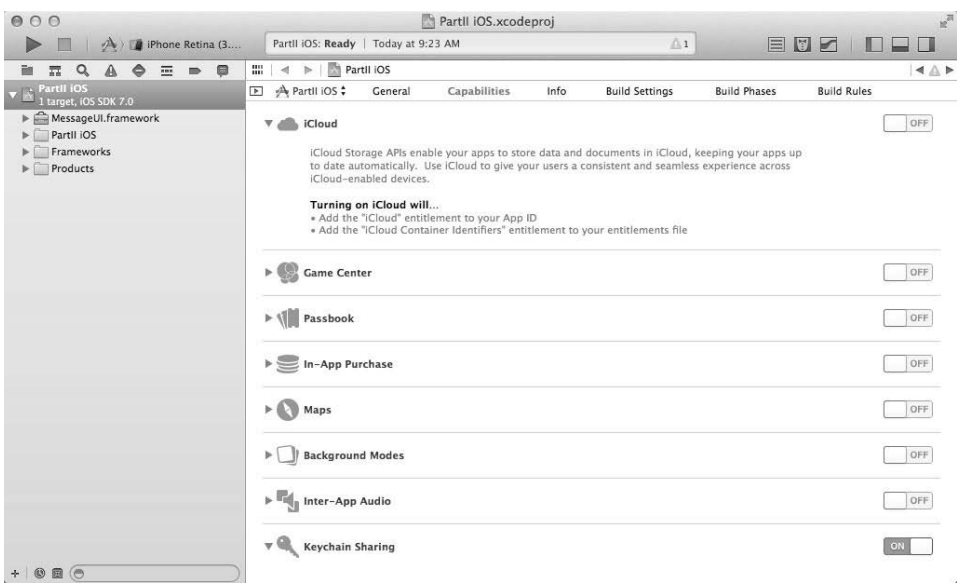

Figure 2.7 Turn capabilities on and off.

When you turn a capability on, you'll be asked to choose a development team to use in provisioning, as you see in Figure 2.8.

Figure 2.8 Choose a development team.

The steps that need to be taken, as shown in Figure 2.7, are checked off or, if a problem occurred, you are usually given an opportunity to have Xcode fix it, as you see in Figure 2.9.

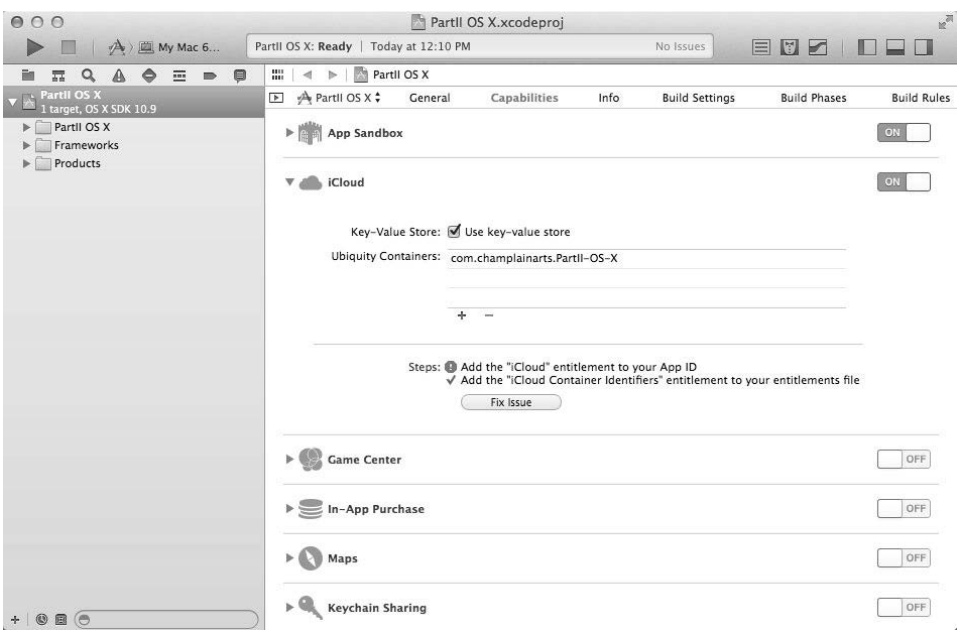

Figure 2.9 Managing Capabilities

Beginning in Xcode 5, this process replaces the manual configuration that you had to do in the past on developer.apple.com. You can still do that, and that is still the best place to actually see the details of your identities, provisioning profiles, and app IDs, but for many if not most of your transactions, Xcode will take care of those tasks. Also note that Xcode sets up the appropriate entries in your project's plist.

As you can see in Figure 2.9, when you enable iCloud, you'll be able to choose the entitlements file, but Xcode will begin by naming one for you. If you want to use a key-value store, you can enable it here: that is the topic of Chapter 8, "Using Key-Value Coding (KVC)." For documents (that is, data other than KVC data), you use a ubiquity container. You may have more than one, but the first one is always assumed to be the main one. If you are using KVC without documents, you don't need a ubiquity container.

Ubiquity Container

As you can see in Figure 2.9, you can specify ubiquity containers for your app. The first one you create has a default name set by Xcode, and it has a special role to play. (You can change the default name if you want, and in some cases, you must, as you'll see in the next paragraph.) The first ubiquity container is the *primary* ubiquity container. On OS X, its contents are displayed in the open and save dialogs available in NSDocument. (On iOS, you create your own interface to display documents in iCloud if you use them.)

The default name for the primary ubiquity container is the bundle identifier. In cases where you want to share a ubiquity container among several apps (such as an OS X version and an iOS version), change one of the ubiquity container names to the other one so it is shared. As you will see in Chapter 18, "Completing the Round Trip," the shared ubiquity container may have any name you want. In Chapter 18, the two apps have bundle identifiers of com.champlainarts.ColbyiOS and com.champlainarts.ColbyOSX. The shared ubiquity container is com.champlainarts.colby.

Using iCloud in Your App

At this point, you're ready to use iCloud in your app. You will see concrete examples of how to do so starting in Part III, "Using the Technologies." There is one step that you can take now to confirm that your app has been properly set up and that the entitlements and provisioning are correct.

Create a new app or use an existing app that you want to enable for iCloud (starting from a new app is a simpler way in the long run until you're more comfortable with iCloud). Set up the entitlements and provisioning as described in the previous sections. Add a single line of code to test if iCloud is available:

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

On OS X, it should go in `applicationDidFinishLaunching:`, and on iOS, it should go in `application:didFinishLaunchingWithOptions:`. In both cases, it normally goes after your other initializations. (Note that this method was added in iOS 6 and OS X Mountain Lion (10.8). You can find older and more complex ways of performing this task on the web.)

The iCloud token that is returned is an opaque object identifying the iCloud account (that means that you can't see inside it). There are two possibilities when you ask for the token:

- If it is nil, the user is not signed into an iCloud account.
- Although you can't see the account details, you can check if a token is the same as another token using `isEqual:`. This lets you check to see if the user has changed iCloud accounts.

Note that if a user has been signed into an iCloud account and turns on Airplane mode or turns off networking on a Mac, the token is still returned. You can access the local copy of your ubiquity container's data. When Airplane mode is turned on again, iCloud will take care of syncing the two stores and will let you know if there is anything for you to do. Because the operating systems manage these disruptions in connectivity, resist the temptation to store extra copies of data locally in the app's sandbox.

Apple recommends as a best practice that you use either iCloud storage or sandbox storage. Mixing the two provides a suboptimal user experience. Along those lines, ask users if they want to use iCloud the first time they run your app. Unless they reinstall the app, don't ask them again.

The iWork apps are a good example of how to manage documents in iCloud. Over the last few years, they have moved to an explicit Export command, which, among other things, can let you export the contents of an iCloud document to another format and to a non-iCloud location.

Chapter Summary

In this chapter, you've seen the basics of how iCloud works. iCloud for document data relies on a ubiquity container, which is identified by a user's Apple ID and your app's bundle ID and is enabled by entitlements. You can share ubiquity containers across several apps by using a single app's bundle ID for all of them.

A high-level overview of the provisioning process has shown you where you enable iCloud for your app. Provisioning is done by registered developers on developer.apple.com and in Xcode accounts. Provisioning profiles as well as identity certificates are then downloaded. You install provisioning profiles in Xcode accounts, while certificates are installed automatically in Key Chain.

Exercises

1. If you have any doubts about the wisdom of Apple's advice to either use iCloud for all storage or local (sandbox) storage for all storage, try to come up with a user interface of your own to manage them.
2. Set up entitlements for an iCloud-enabled app as described in this chapter. Start by following the steps exactly—either those in this chapter or those on developer.apple.com. Don't take any shortcuts until your first provisioning profile is running properly. Then you can experiment.
3. TextEdit supports iCloud documents; it is installed as part of the OS X installation. Experiment with it and particularly note how iCloud has been integrated into the File Save dialog. You can access this dialog from your own code when you instantiate NSDocument objects.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z