24

Introduction to iCloud Storage

iCloud Storage is a set of classes and services that enable you to share data between instances of your application running across different devices. In this lesson, you learn to use the iCloud Storage APIs in your apps.

BASIC CONCEPTS

Apple's iCloud is a service that allows applications to synchronize data across devices. Your data is stored across a set of servers maintained by Apple and is made available to copies of your app across all iCloud-compatible devices. Changes made to this data by one instance of your application are automatically propagated to other instances.

From a developer's perspective, you need to use Apple's iCloud Storage APIs to interact with the iCloud service. These APIs enable you to store both documents and small amounts of key-value data.

> **NOTE** *This lesson does not cover key-value data storage. For more information on storing key-value data with iCloud, refer to the Designing for Key-Value Data in iCloud section of the iCloud Design Guide, available at:*
>
> `https://developer.apple.com/library/ios/documentation/General/Conceptual/iCloudDesignGuide/Chapters/DesigningForKey-ValueDataIniCloud.html#//apple_ref/doc/uid/TP40012094-CH7-SW1.`

iCloud applications cannot be tested on the iOS Simulator, and to make the most of this lesson you should ideally have two iOS devices to test on. iCloud Storage APIs are available to both iOS and MacOS X developers.

Your iOS applications always execute in a restricted environment on the device known as the *application sandbox*. Some of these restrictions affect where and how your application can store data.

Each application is given a directory on the device's file system. The contents of this directory are private to the application and cannot be read by other applications on the device.

Each application's directory has four locations into which you can store data:

- `Preferences`
- `Documents`
- `Caches`
- `tmp`

The first of these, `Preferences`, is not intended for direct file manipulation; however, the other three are. The most commonly used directories are the `Documents` and the `tmp` directories.

The `Documents` directory is the main location for storing application data. The contents of this directory can also be manipulated within iTunes. The `Caches` directory is used to store temporary files that need to persist between application launches. The `tmp` directory is used to store temporary files that do not need to persist between application launches.

Applications are responsible for cleaning up the contents of these directories because storage space on a device is limited. The contents of the `Caches` and `tmp` directories are not backed up by iTunes.

iCloud Storage conceptually extends this model and allows your applications to upload your data from its private directory to Apple's servers. This data then filters down to other iCloud-compatible devices on which copies of your application are running. Your application also receives notifications when a document has been created or updated by another copy of the application.

This synchronization is achieved by a background process (also known as a daemon) that runs on all iCloud-compatible devices. Figure 24-1 illustrates the iCloud architecture.

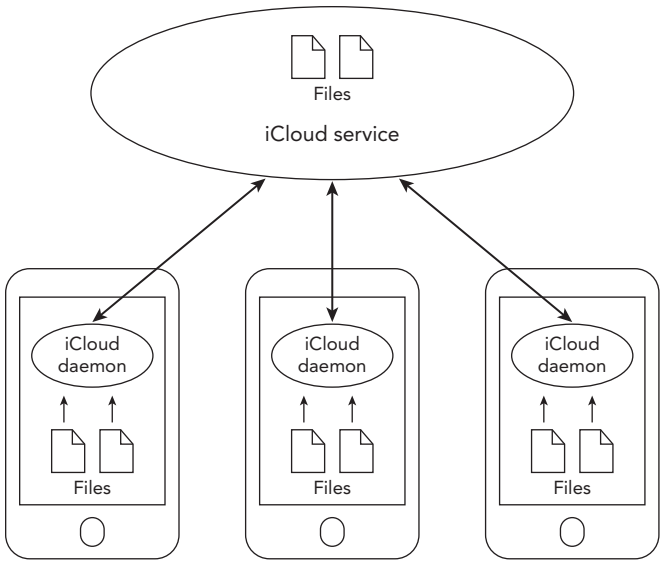

FIGURE 24-1

PREPARING TO USE THE ICLOUD STORAGE APIS

To use the iCloud Storage APIs in an application, you need to perform three steps:

1. Create an iCloud-enabled App ID.
2. Create an appropriate provisioning profile.
3. Enable appropriate entitlements in your Xcode project.

Creating an iCloud-Enabled App ID

To create an appropriate App ID, log in to your iOS developer account at `https://developer.apple.com/ios`. Click the Member Center link on the right side to navigate to the member center. Within the member center click, the Certificates, Identifiers & Profiles link (see Figure 24-2).

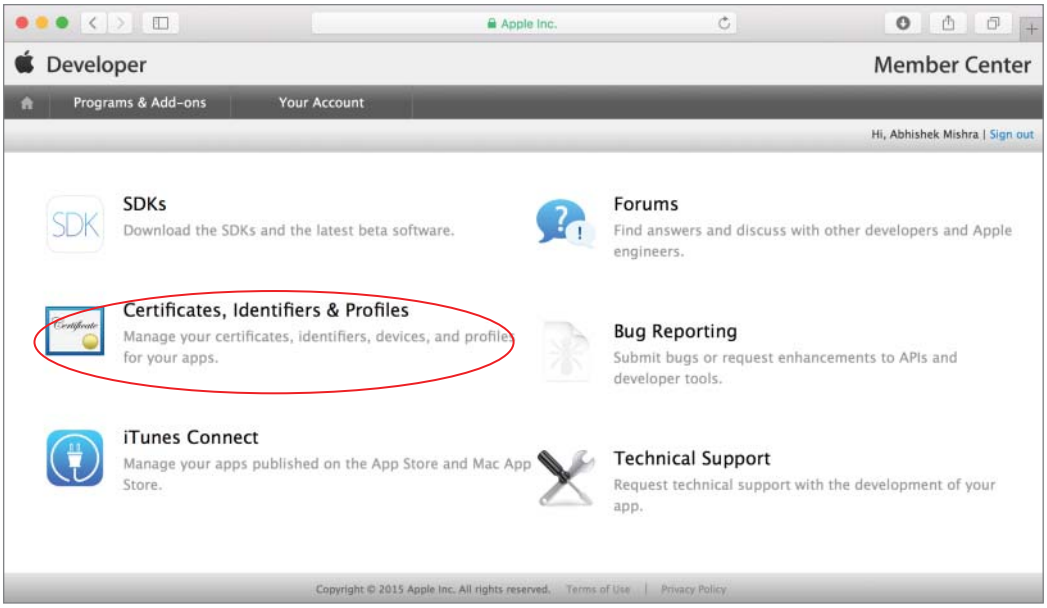

FIGURE 24-2

Next, click the Identifiers link in the iOS Apps category on the left side of the page (see Figure 24-3).

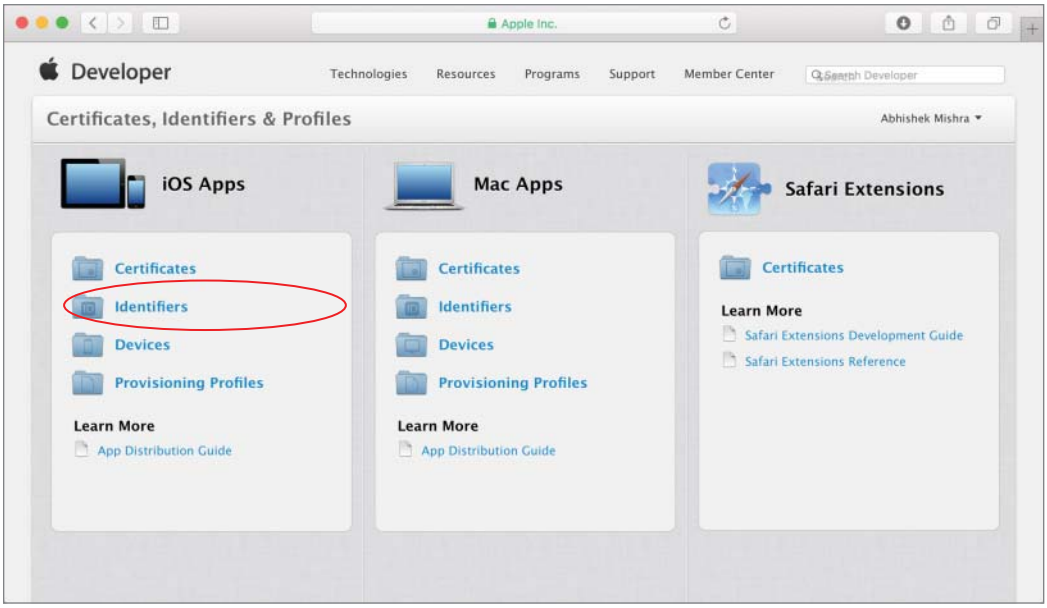

FIGURE 24-3

To create a new App ID, click the New App ID button on the top-right side (see Figure 24-4).

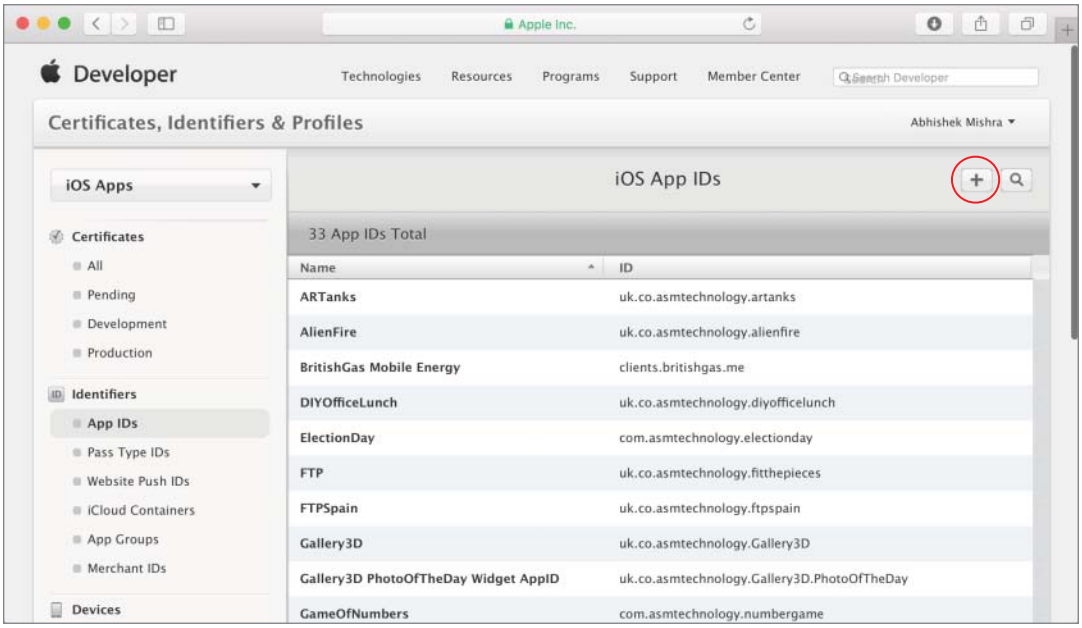

FIGURE 24-4

Provide a descriptive name of the new App ID in the Name field and select Team ID in the App ID prefix drop-down. Select the Explicit App ID radio button under the App ID suffix section and provide a unique identifier in the Bundle ID field that ends in the name of the Xcode project you are going to create (or have created).

Typically, you create this identifier by combining the reverse-domain name of your website and the name of your Xcode project. For example, the project created in this lesson is called `SwiftCloudTest` and the bundle identifier specified is `com.wileybook.CloudTest`. Your browser window should resemble Figure 24-5.

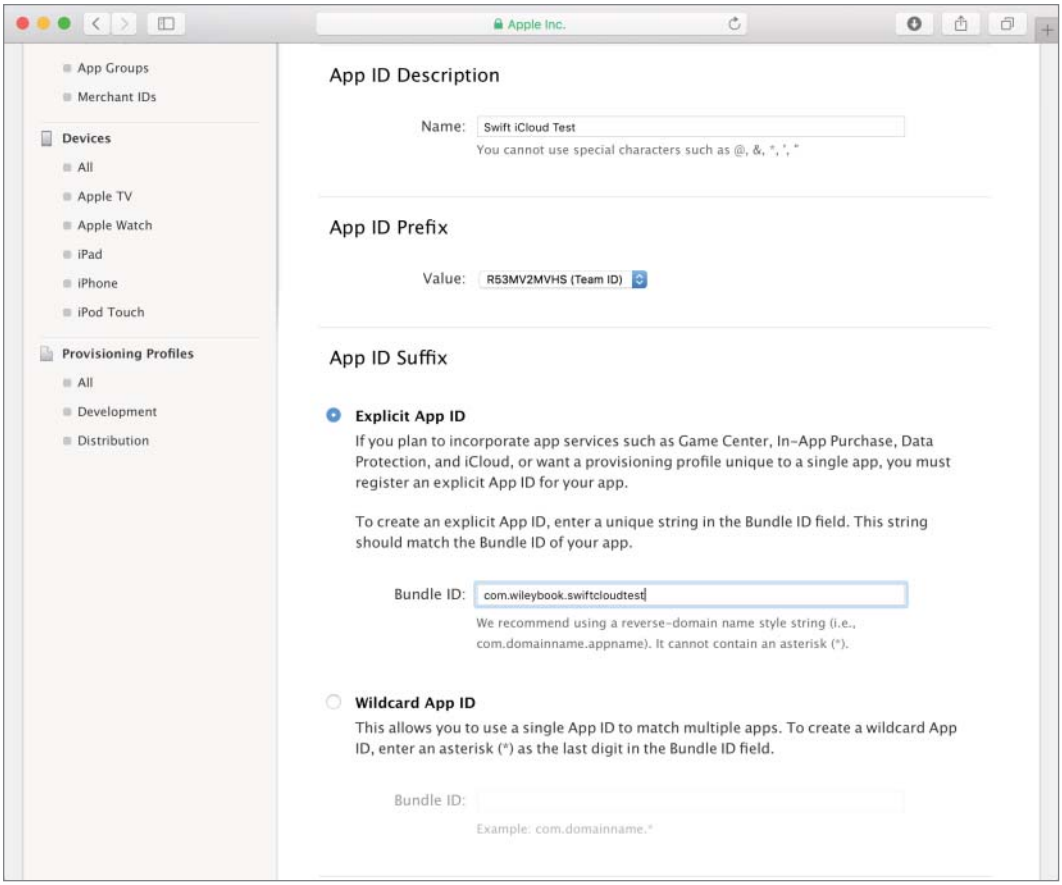

FIGURE 24-5

Scroll down to reveal the App Services section and ensure the iCloud checkbox is selected and the Compatible with Xcode 5 option is selected (see Figure 24-6).

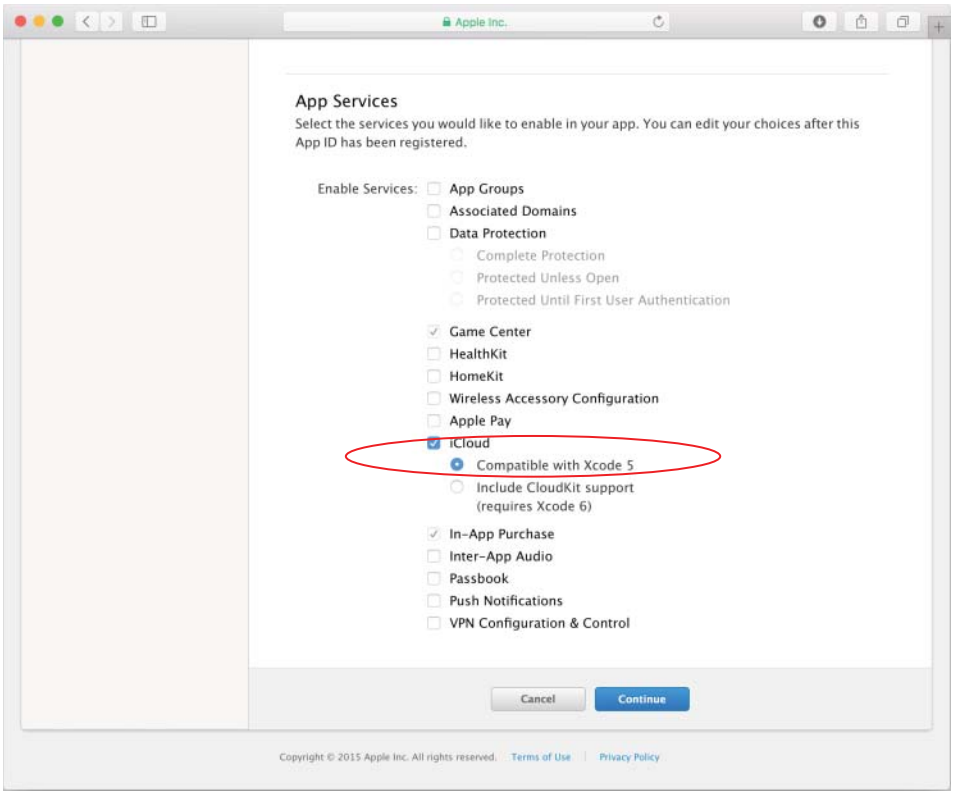

FIGURE 24-6

Click the Continue button to proceed. You will be presented with a summary of the App ID information (see Figure 24-7). Click on Submit to finish creating the App ID.

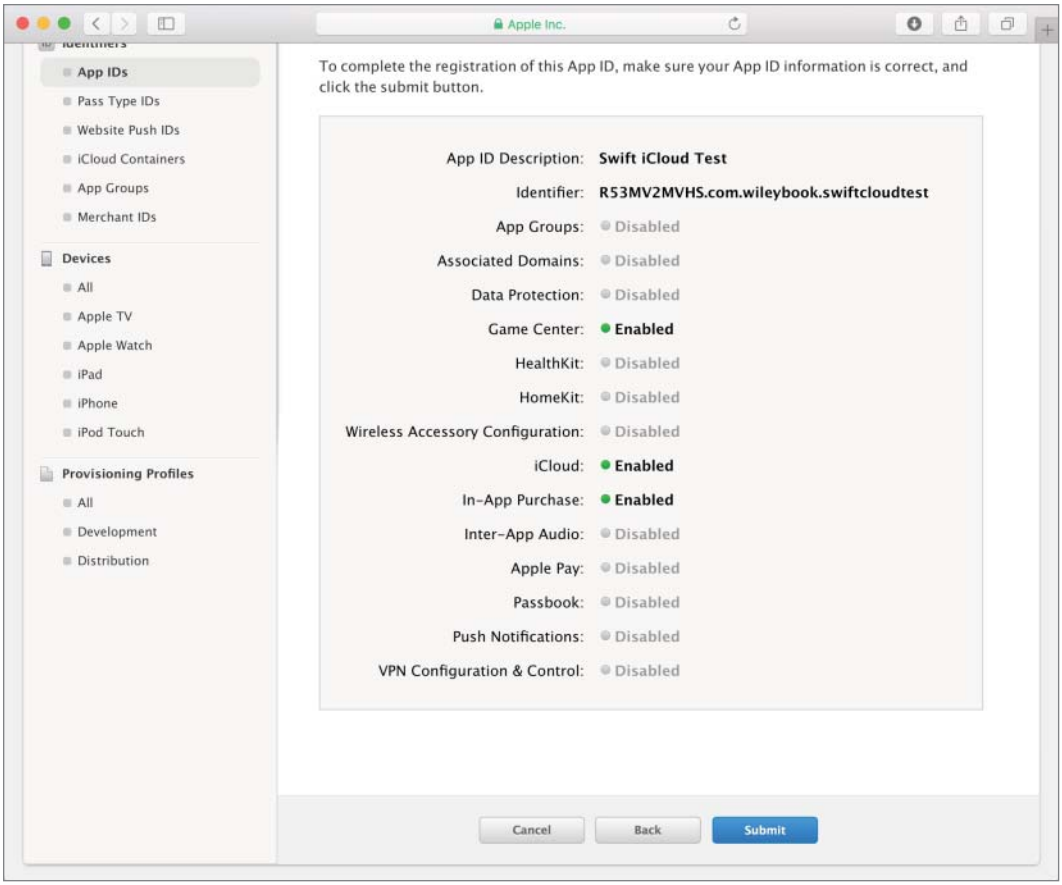

FIGURE 24-7

Creating an Appropriate Provisioning Profile

To create a provisioning profile for an iCloud-enabled App ID, click the All link (under the Provisioning category) in the menu on the left side of the iOS Provisioning Portal window (see Figure 24-8).

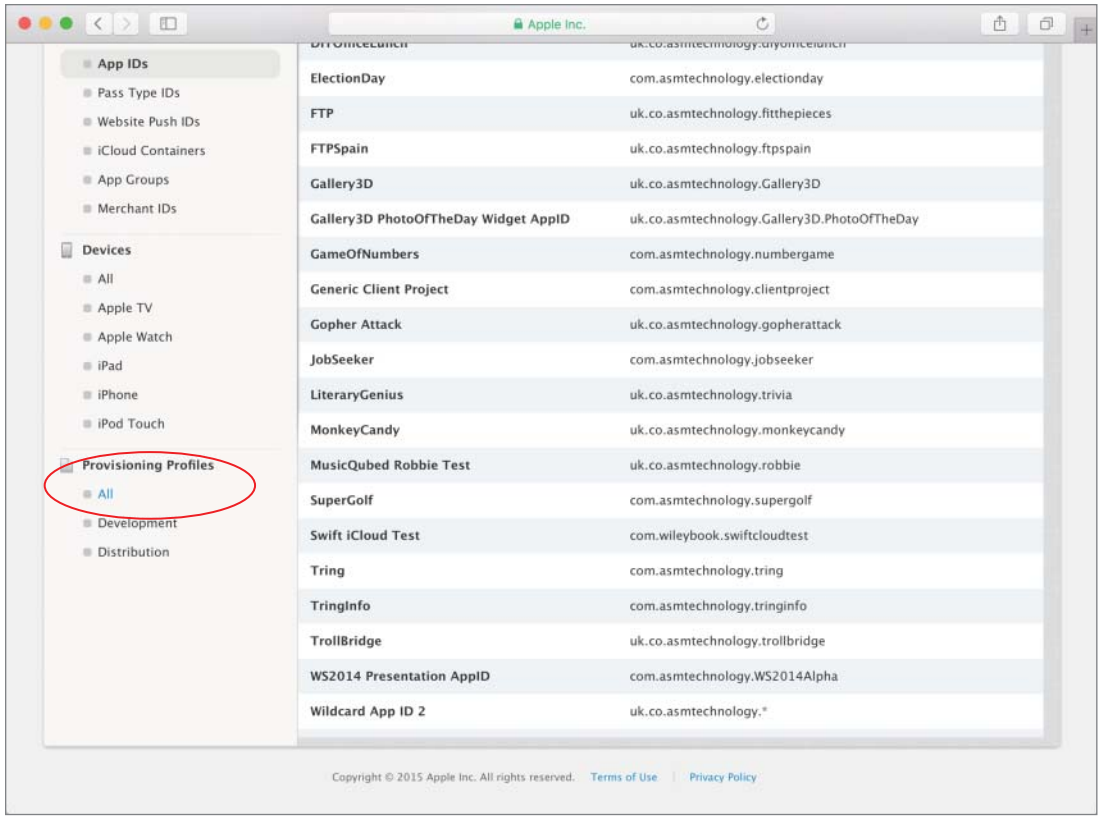

FIGURE 24-8

Click the New Profile button on the top-right side (see Figure 24-9).

FIGURE 24-9

You will be asked to choose between a development or distribution provisioning profile. A distribution provisioning profile is used to submit applications to iTunes Connect. For the moment, select the iOS App Development option and click Continue (see Figure 24-10).

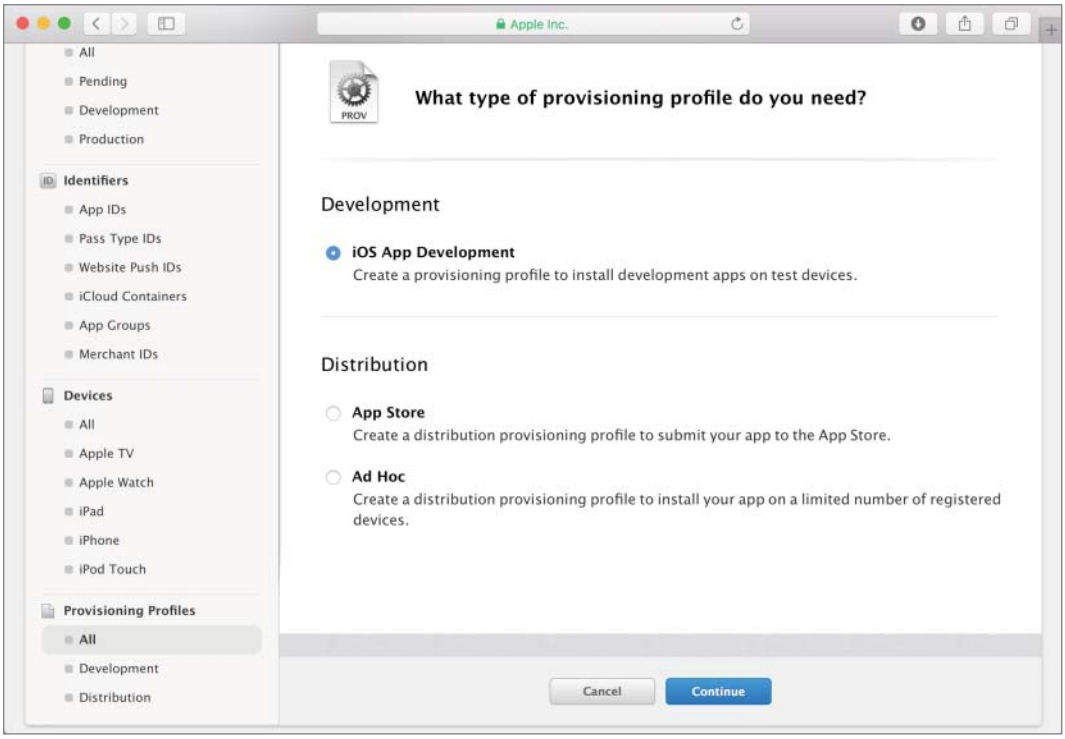

FIGURE 24-10

A development provisioning profile ties together three pieces of information:

- A single App ID
- One or more public keys
- A list of test device IDs

The next step requires you to select an App ID that will be associated with this provisioning profile. Select the iCloud-enabled App ID you have created (see Figure 24-11) and click Continue.

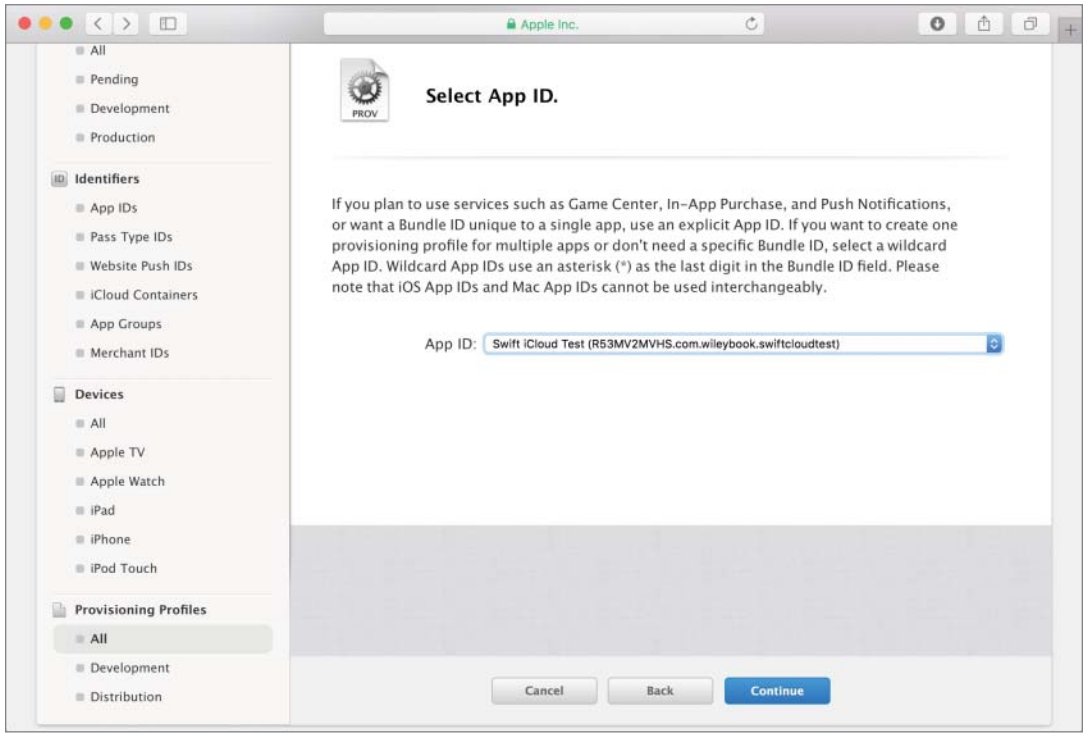

FIGURE 24-11

Select one or more development certificates that will be included in the profile. You must make sure to sign the app in Xcode using one of the certificates you select here. Select a suitable certificate and click Continue (see Figure 24-12).

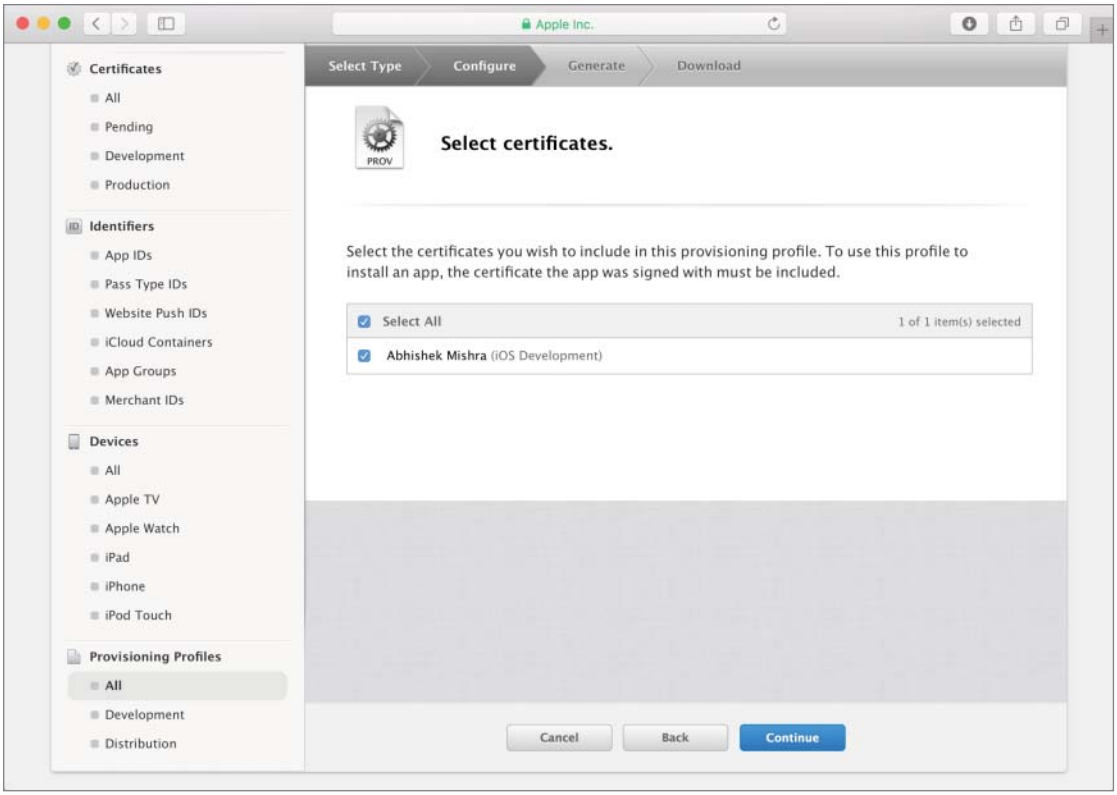

FIGURE 24-12

Next, you must select one or more devices that will be included in this provisioning profile. The corresponding identifiers for these devices must be registered with your development account. Your app will only be testable on these devices (see Figure 24-13).

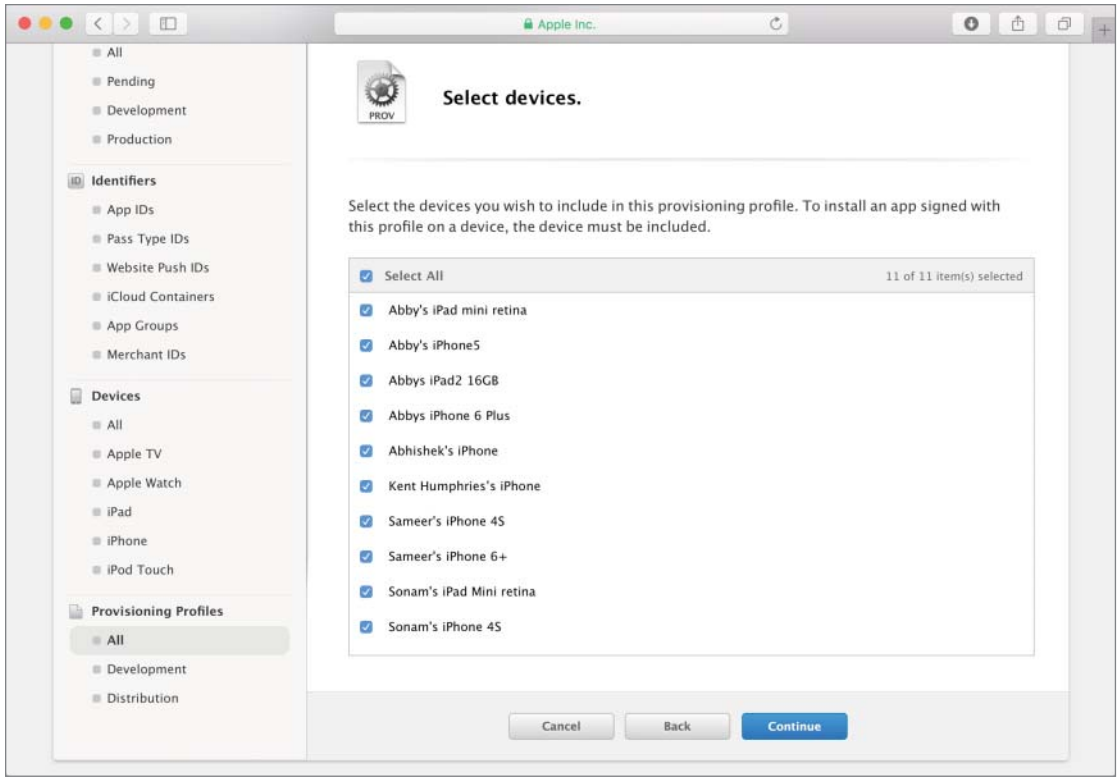

FIGURE 24-13

The final step involves providing a suitable name for the profile and clicking the Generate button. When the profile is created, you will be provided an option to download it onto your computer (see Figure 24-14).

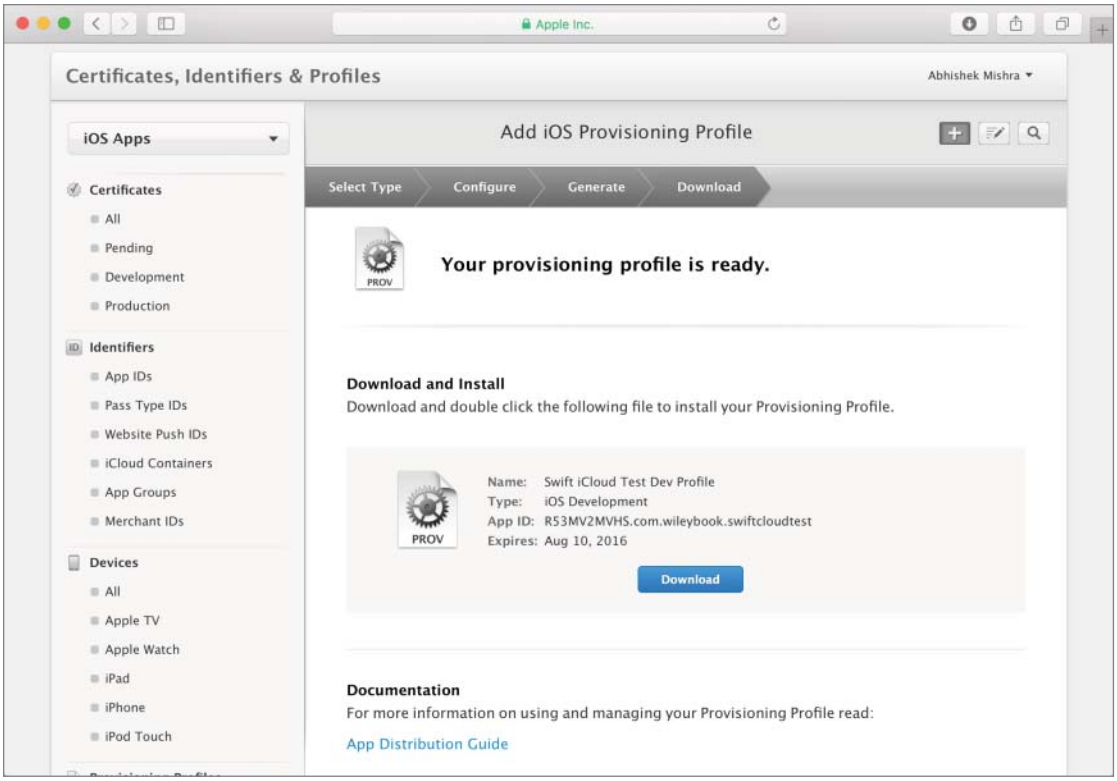

FIGURE 24-14

If you were to now click the All link under the Provisioning section on the left side menu, you should see an entry for the new profile in the list of available profiles. You can also download a provisioning profile from this list.

Once the profile has been downloaded, simply locate it in the Downloads folder on your Mac and double-click it to install it in Xcode.

Enabling Appropriate Entitlements in Your Xcode Project

Create a new project in Xcode using one of the standard iOS application templates. In the Project Options dialog box, make sure you provide the correct value for the Product Name and Organization Identifier fields so as to create the same App ID that was registered on the iOS Provisioning Portal. If, for instance, the App ID you registered was `com.wileybook.swiftcloudtest`, use `swiftcloudtest` for the Product Name field and `com.wileybook` for the Company Identifier field.

Applications that use iCloud must be signed with iCloud-specific entitlements. These entitlements ensure that only your applications can access the documents that they create. To enable entitlements, select the project's root node in the project navigator and the appropriate build target. Ensure the

Capabilities tab is selected. Locate the iCloud node and enable it. You may be asked to provide your iOS developer accounts credentials when you enable the iCloud entitlement. Because this lesson is about iCloud document storage, ensure the iCloud Documents checkbox is checked (see Figure 24-15).

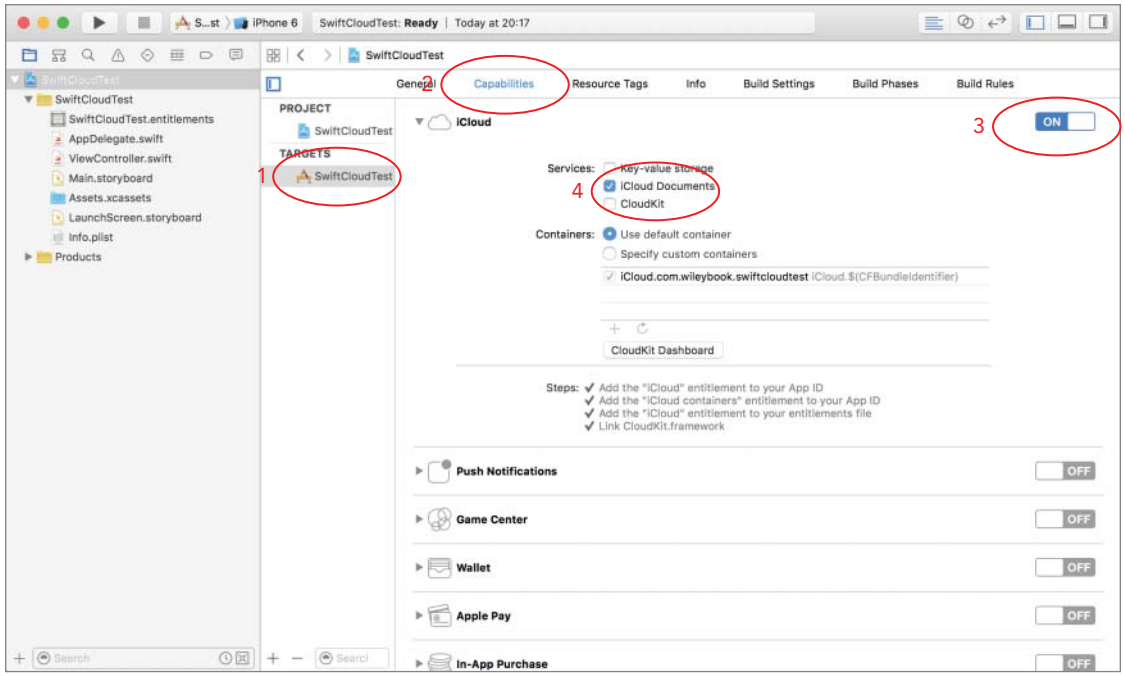

FIGURE 24-15

CHECKING FOR SERVICE AVAILABILITY

If your application intends to make use of the iCloud Storage APIs, you must ensure that the service is available to the application. This may not necessarily be the case if, for example, the user has not set up iCloud on the device.

To check for service availability, use the `URLForUbiquityContainerIdentifier()` method of the `NSFileManager` class. This method requires one `String` parameter that specifies a container identifier that your application uses.

If this method succeeds, the return value is an `NSURL` instance that identifies the container directory. If the method fails, the return value is `nil`.

If your application uses only one container identifier, or you want to use the main container identifier for the application, pass `nil` for the parameter. If your application accesses multiple containers, you must call this method for each container identifier to ensure you have access to each container. The following code snippet shows how to use this method for the main container identifier:

```
let folderURL =
NSFileManager.defaultManager().URLForUbiquityContainerIdentifier(nil)

if let unwrappedFolderURL = folderURL {
    // cloud access is available
}
else {
    // cloud access is not available.
}
```

USING ICLOUD DOCUMENT STORAGE

Any file stored by your application on iCloud must be managed by a file presenter object. A file presenter is an object that implements the `NSFilePresenter` protocol. Essentially, a file presenter acts as an agent for a file. Before an external source can change the file, the file presenter for the file is notified. When your app wants to change the file, it must lock the file by making its changes through a file coordinator object. A file coordinator object is an instance of the `NSFileCoordinator` class.

The simplest way to incorporate file presenters and coordinators in your application is to have your data classes (also known as model classes) subclass `UIDocument`. The `UIDocument` class implements the methods of the `NSFilePresenter` protocol and handles all of the file-related management. At the most basic level, you will need to override two `UIDocument` methods:

```
public func loadFromContents(contents: AnyObject,
            ofType typeName: String?) throws

public func contentsForType(typeName: String) throws -> AnyObject
```

The `loadFromContents(contents, ofType)` method is overridden by your `UIDocument` subclass and is called when the application needs to read data into its data model.

The first parameter of this method, `contents`, encapsulates the document data to be read. In the case of flat files, `contents` is an instance of an `NSData` object. It can also be an `NSFileWrapper` instance if the data being read corresponds to a file package. The `typeName` parameter indicates the file type of the document.

If you cannot load the document for some reason, you should throw an exception encapsulating the reason for failure.

The `contentsForType()` method is also overridden by your `UIDocument` subclass and is called when the application saves data to a file. This method must return an `NSData` instance that will be written to the file. If you cannot return an `NSData` instance for some reason, you throw an exception that encapsulates the reason for failure.

The following code presents a simple `UIDocument` subclass called `SwiftCloudTestDocument`. The example assumes that the application where this class is used has a rather simple data model consisting of a single `String` instance.

```
import UIKit

enum DocumentReadError: ErrorType {
```

```
    case InvalidInput
}

enum DocumentWriteError: ErrorType {
    case NoContentToSave
}

class SwiftCloudTestDocument: UIDocument {

    var documentContents:String?

    override init(fileURL url: NSURL) {
        super.init(fileURL: url)
    }

    override func loadFromContents(contents: AnyObject,
                  ofType typeName: String?) throws {

        if let castedContents = contents as? NSData {
            documentContents = NSString(data: castedContents,
            encoding: NSUTF8StringEncoding) as? String

        }
        else {
            documentContents = nil
            throw DocumentReadError.InvalidInput
        }
    }

    override func contentsForType(typeName: String) throws -> AnyObject {

        if documentContents == nil {
            throw DocumentWriteError.NoContentToSave
        }

        return documentContents!.dataUsingEncoding(NSUTF8StringEncoding)!
    }
}
```

Creating a New iCloud Document

To create a new document, initialize an instance of your `UIDocument` subclass by using the `init(fileURL url: NSURL)` initializer and then call `saveToURL(url, saveOperation, completionHandler)` on the instance.

The initializer requires a single `NSURL` parameter that identifies the location where document data is to be written. This URL is usually composed by appending a filename in the `Documents` subdirectory to the path to an iCloud container. For instance, to create a new document on iCloud called `phoneNumber.txt`, you could use the following snippet:

```
let containerURL =
NSFileManager.defaultManager().URLForUbiquityContainerIdentifier(nil)
```

```
let documentDirectoryURL = containerURL!.URLByAppendingPathComponent("Documents")

let documentURL =
documentDirectoryURL.URLByAppendingPathComponent("phoneNumber.txt")

let cloudDocument:SwiftCloudTestDocument =
    SwiftCloudTestDocument(fileURL: documentURL)

cloudDocument.saveToURL(cloudDocument.fileURL,
forSaveOperation: UIDocumentSaveOperation.ForCreating) {
   (Bool success) -> Void in
    if (success) {
        // document was created successfully.
    }
}
```

The `saveToURL(url, saveOperation, completionHandler)` method is described later in this lesson.

Opening an Existing Document

To open an existing document, allocate and initialize an instance of your `UIDocument` subclass and call `openWithCompletionHandler()` on the instance. For example, you could open a file called `phoneNumbers.txt` from iCloud using the following snippet:

```
let containerURL =
NSFileManager.defaultManager().URLForUbiquityContainerIdentifier(nil)

let documentDirectoryURL = containerURL!.URLByAppendingPathComponent("Documents")

let documentURL =
documentDirectoryURL.URLByAppendingPathComponent("phoneNumber.txt")

let cloudDocument:SwiftCloudTestDocument =
SwiftCloudTestDocument(fileURL: documentURL)

cloudDocument.openWithCompletionHandler {
        (BOOL success) -> Void in
        if (success)
        {
            // cloud document opened successfully!
        }
}
```

Saving a Document

Once you have an instance of a `UIDocument` subclass, saving it to iCloud is simply a matter of calling the `saveToURL(url, saveOperation, completionHandler)` method on it. The first parameter to this method is an `NSURL` instance that contains the target URL. You can compose this URL in the same manner as when you instantiated your `UIDocument` subclass.

If, however, you want to retrieve the URL corresponding to an existing `UIDocument` subclass, simply use the `fileURL` property of the subclass. Thus, if `cloudDocument` is an instance of a `UIDocument` subclass, you can retrieve the URL used when it was instantiated using the following code:

```
Let documentURL = cloudDocument.fileURL
```

The second parameter is a constant that is used to indicate whether the document contents are being saved for the first time, or overwritten. It can be either of:

- `UIDocumentSaveOperation.ForCreating`
- `UIDocumentSaveOperation.ForOverwriting`

The third parameter is a block completion handler.

> **NOTE** *For more information on the* `UIDocument` *class, refer to the UIDocument Class reference, available at:*
>
> `https://developer.apple.com/library/prerelease/ios/documentation/UIKit/Reference/UIDocument_Class/index.html`.

Searching for Documents on iCloud

Often, you will need to search iCloud container directories for documents. To do this, you need to create a search query using an `NSMetadataQuery` instance, set up an appropriate search filter, and execute the query.

Queries have two phases: an initial search phase and a second live-update phase. During the live-update phase, updated results are typically available once every second. The following code snippet builds a search query:

```
let searchQuery:NSMetadataQuery = NSMetadataQuery()
searchQuery.searchScopes = [NSMetadataQueryUbiquitousDocumentsScope];
```

The `searchScopes` property allows you to specify an array of directory strings over which the search should execute. To specify the iCloud container folder as the search target, you provide an `Array` instance with a single object:

```
NSMetadataQueryUbiquitousDocumentsScope
```

Before you can execute the query, you need to specify a search filter. Search filters are also known as predicates and are instances of the `NSPredicate` class. The following code snippet creates an `NSPredicate` instance that filters out a file with a specific name:

```
let documentFileName = "cloudDocument.txt"
let predicate = NSPredicate(format: "%K == %@",
argumentArray: [NSMetadataItemFSNameKey, documentFileName])
```

To apply the predicate to the search query, use the `predicate` property on the `NSMetadataQuery` instance:

```
searchQuery.predicate = predicate
```

Search queries execute asynchronously. When the query has finished gathering results, your application will receive the `NSMetadataQueryDidFinishGatheringNotification` notification message. Use the following code snippet to set up a method in your code called `queryDidFinish()` to be called when this notification is received:

```
NSNotificationCenter.defaultCenter().addObserver(self,
selector: "queryDidFinish:",
name: NSMetadataQueryDidFinishGatheringNotification ,
object: searchQuery)
```

Finally, to start the query, call the `startQuery` method of the `NSMetadataQuery` instance:

```
searchQuery.startQuery()
```

When you receive the notification message, you can find out the number of results returned by the search by querying the `resultCount` property of the `NSMetadataQuery` instance:

```
let numResults = searchQuery.resultCount
```

To retrieve an `NSURL` instance for each result returned by the search query, you can use a simple `for` loop:

```
for (var resultIndex = 0; resultIndex < numResults; resultIndex++)
{
    let item:NSMetadataItem? = searchQuery.results[resultIndex] as?
                               NSMetadataItem

    if let unwrappedItem = item {
        let url = unwrappedItem.valueForAttribute(NSMetadataItemURLKey)
    }
}
```

If you do not want the search query to continue returning results, use the following code snippet to stop it:

```
searchQuery.disableUpdates()
searchQuery.stopQuery()
```

The Try It section for this lesson contains a simple project that uses an `NSMetadataQuery` instance to find a document on iCloud and then proceeds to open it.

> **NOTE** *For more information on the* `NSMetadataQuery` *class, refer to the NSMetadataQuery Class Reference, available at:*
>
> `https://developer.apple.com/library/prerelease/ios/documentation/Cocoa/Reference/Foundation/Classes/NSMetadataQuery_Class/.`
>
> *For more information on the* `NSPredicate` *class, refer to the NSPredicate Class Reference available at:*
>
> `https://developer.apple.com/library/prerelease/mac/documentation/Cocoa/Reference/Foundation/Classes/NSPredicate_Class/index.html.`

TRY IT

In this Try It, you build a new Xcode project based on the Single View Application template called `SwiftCloudTest`. In this application, you create a simple text document called `cloudDocument.txt` and store it on iCloud. This document can then be edited across multiple copies of the application running on different iOS devices.

Lesson Requirements

- Create a new Universal application project based on the Single View Application template.
- Register the App ID with the iOS Provisioning Portal.
- Create a development provisioning profile.
- Download and install the development provisioning profile.
- Create a simple user interface that consists of a `UIButton` instance, a `UILabel` instance, and a `UITextView` instance.
- Create a data class that subclasses `UIDocument`.
- Check iCloud service availability in the `viewDidLoad` method of the view controller class.
- Load an existing document stored on iCloud. If the document does not exist, create a new one.
- Implement code to save the document on iCloud when a button is tapped.

> **REFERENCE** *The code for this Try It is available at* `www.wrox.com/go/swiftios`.

Hints

- To make best use of this application, you will need at least two iOS devices set up to use the same iCloud account.
- You must ensure iCloud has been set up on each test device.
- Testing your apps on iOS devices is covered in Appendix A.

Step-by-Step

- Create a Single View Application in Xcode called `SwiftCloudTest`.
  1. Launch Xcode and create a new application by selecting File ➪ New ➪ Project.
  2. Select the Single View Application template from the list of iOS project templates.
  3. In the project options screen, use the following values:
     - **Product Name:** SwiftCloudTest
     - **Organization Name:** your company
     - **Organization Identifier:** com.yourcompany
     - **Language:** Swift
     - **Devices:** Universal
     - **Use Core Data:** Unchecked
     - **Include UI Tests:** Unchecked
     - **Include Unit Tests:** Unchecked
  4. Save the project onto your hard disk.
- Register an App ID with the iOS Provisioning Portal.
  1. Log in to the iOS Provisioning Portal, and register a new App ID with the following details:
     - **Description:** SwiftCloudTest AppID
     - **Bundle Seed ID:** Use Team ID
     - **Bundle Identifier:** com.wileybook.SwiftCloudTest
  2. Enable the App ID to use with iCloud. This process is covered in the section titled "Creating an iCloud-Enabled App ID" earlier in this lesson.
- Create a development provisioning profile using the App ID created in the previous step.
  1. The process of creating the provisioning profile is covered in the section titled "Creating an Appropriate Provisioning Profile" earlier in this lesson. Follow those instructions to create a development provisioning profile called Swift Cloud Test Development Profile.
  2. Download and install the provisioning profile by double-clicking on the profile after it has been downloaded to your computer.

- Enable iCloud-specific entitlements for the application target.
  1. Select the project's root node in the project navigator and select the appropriate build target. Ensure the Capabilities tab is selected. Scroll down to the iCloud option and enable it.
  2. Once the iCloud entitlement has been enabled, ensure the iCloud Documents checkbox has been checked
- Create a `UIDocument` subclass.
  1. Right-click your project's root node in the project navigator and select New File from the context menu.
  2. Select the Swift file template and click Next.
  3. Name the class `SwiftCloudTestDocument` and click Create.
  4. Replace the contents of the `SwiftCloudTestDocument.swift` file with the following:

```
import UIKit

enum DocumentReadError: ErrorType {
    case InvalidInput
}

enum DocumentWriteError: ErrorType {
    case NoContentToSave
}

class SwiftCloudTestDocument: UIDocument {

    var documentContents:String?

    override init(fileURL url: NSURL) {
        super.init(fileURL: url)
    }

    override func loadFromContents(contents: AnyObject,
                  ofType typeName: String?) throws {

        if let castedContents = contents as? NSData {
            documentContents = NSString(data: castedContents,
            encoding: NSUTF8StringEncoding) as? String

        NSNotificationCenter.defaultCenter().
        postNotificationName("refreshDocumentPreview",
        object: self)
        }
        else {
            documentContents = nil
            throw DocumentReadError.InvalidInput
        }
    }

    override func contentsForType(typeName: String)
```

```
                    throws -> AnyObject {

            if documentContents == nil {
                throw DocumentWriteError.NoContentToSave
            }

            return documentContents!.dataUsingEncoding(NSUTF8StringEncoding)!
        }
    }
```

Recall that the `loadFromContents()` method is called when a document must be loaded from a file. In case of iCloud documents, this method is also called automatically when the contents of the file have changed. This will typically happen when the file was edited by another copy of the application.

In the preceding implementation, in addition to loading the contents of the file into member variables of the `SwiftCloudTestDocument` class, you also send out an application-wide notification called `refreshDocumentPreview`.

The view controller class listens for these notifications, and treats the arrival of one as a cue to update the user interface.

- Create a simple user interface with Interface Builder.

1. Open the storyboard file and use the Object library to drag and drop a label, button, and text view onto the default scene.
2. Select the label and display the Pin constraints popup. Ensure the Constrain to margins options is unchecked and Update Frames is set to None. Create the following layout constraints:
    - **Left:** 20
    - **Right:** 20
    - **Top:** 20
    - **Height:** 21
3. Select the button and display the Pin constraints popup. Ensure the Constrain to margins options is unchecked and Update Frames is set to None. Create the following layout constraints:
    - **Left:** 20
    - **Right:** 20
    - **Top:** 20
    - **Height:** 30
4. Select the text view and the Pin constraints popup. Ensure the Constrain to margins options is unchecked and Update Frames is set to All Frames in Container. Create the following layout constraints:
    - **Left:** 20
    - **Right:** 20

- **Top:** 20
- **Bottom:** 20

5. Use the Attribute inspector to set the text property of the label to iCloud Service Status.
6. Use the Attribute inspector to set the Alignment property of the label to center.
7. Double-click the button in the scene and change its title to Save Document.
8. Change the background color of the button to a shade of gray.
9. Use the assistant editor to create an outlet called `serviceStatus` in the `ViewController` class and connect it to the `UILabel` instance in the default scene.
10. Use the assistant editor to create an outlet called `documentContentView` in the `ViewController` class and connect it to the `UITextView` instance in the default scene.
11. Use the assistant editor to create an action method called `onSaveDocument` in the `ViewController` class and connect it to the `Touch Up Inside` event of the `UIButton` instance in the default scene.

Your storyboard should resemble Figure 24-16.

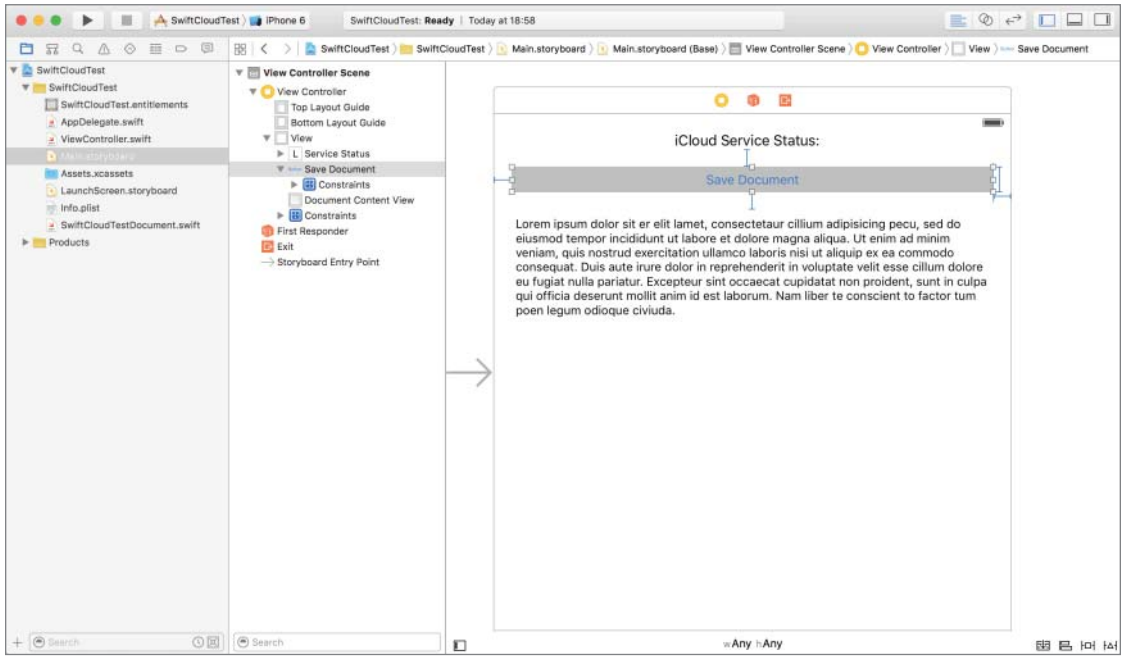

FIGURE 24-16

- Edit the `ViewController.swift` file.

1. Add the following member variable declarations to the class:

```
var cloudServicesAreAvailable:Bool?
var cloudDocument:SwiftCloudTestDocument?
var searchQuery:NSMetadataQuery?
```

2. Update the implementation of the `viewDidLoad` method to resemble the following:

```
override func viewDidLoad() {

    super.viewDidLoad()

    documentContentView.text = ""

    // register this class as an observer for the 'refreshDocumentPreview'
    // notification, this notification is sent by the document class when
    // the contents of the document have ben updated.
    NSNotificationCenter.defaultCenter().addObserver(self,
    selector: "refreshDocumentPreview:",
    name: "refreshDocumentPreview" ,
    object: nil)

    // check if cloud services are available.
    let containerURL =
    NSFileManager.defaultManager().URLForUbiquityContainerIdentifier(nil)

    if containerURL != nil  {
        self.cloudServicesAreAvailable = true
        serviceStatus.text = "Cloud Service Status: Available"

        // load existing document, or create a new document
        loadDocument()
    }
    else {
        self.cloudServicesAreAvailable = false
        serviceStatus.text = "Cloud Service Status: Not Available"

        let alert = UIAlertController(title: "Error",
        message: "iCloud has not been setup on this device!",
        preferredStyle: UIAlertControllerStyle.Alert)

        alert.addAction(UIAlertAction(title: "Ok",
        style: UIAlertActionStyle.Default,
        handler: nil))

        self.presentViewController(alert,
        animated: true,
        completion: nil)
    }
}
```

In this method, you check if the iCloud service is available, and if it is, then proceed to load a specific document from iCloud.

3. Implement the `deinit` method in your `ViewController` class as follows:

```
deinit {
```

```
    if cloudDocument != nil {
        cloudDocument?.closeWithCompletionHandler(nil)
    }

    NSNotificationCenter.defaultCenter().removeObserver(self)
}
```

4. Add a new method called `loadDocument` method as follows:

```
func loadDocument(){
    // search for cloudDocument.txt
    searchQuery = NSMetadataQuery()
    searchQuery!.searchScopes = [NSMetadataQueryUbiquitousDocumentsScope];

    let documentFileName = "cloudDocument.txt"
    let predicate = NSPredicate(format: "%K == %@",
    argumentArray: [NSMetadataItemFSNameKey, documentFileName])

    searchQuery!.predicate = predicate

    NSNotificationCenter.defaultCenter().addObserver(self,
    selector: "queryDidFinish:",
    name: NSMetadataQueryDidFinishGatheringNotification ,
    object: searchQuery)

    UIApplication.sharedApplication().networkActivityIndicatorVisible = true

    searchQuery!.startQuery()
}
```

These statements instantiate an `NSMetadataQuery` object to search the Documents directory in the application's iCloud container for a file called `cloudDocument.txt`. When the query is complete, the `queryDidFinish()` method of the view controller class will be called.

5. Implement the `queryDidFinish()` method as follows:

```
func queryDidFinish(notifcation: NSNotification) {

    UIApplication.sharedApplication().networkActivityIndicatorVisible = false

    // stop the query to prevent it from running constantly
    searchQuery!.disableUpdates()
    searchQuery!.stopQuery()

    NSNotificationCenter.defaultCenter().removeObserver(self,
    name: NSMetadataQueryDidFinishGatheringNotification,
    object: nil)

    // this application expects this query to return a single
    // result. If no documents were found, then create a new
    // document and inform the user.
    if searchQuery!.resultCount == 0
    {
        let alert = UIAlertController(title: "",
```

```
        message: "iCloud document not found., creating new document!",
        preferredStyle: UIAlertControllerStyle.Alert)

        alert.addAction(UIAlertAction(title: "Ok",
        style: UIAlertActionStyle.Default,
        handler: nil))

        self.presentViewController(alert,
animated: true,
completion: nil)

        createDocument()
        return
    }

    // instantiate a SwiftCloudTestDocument instance and
    // open the cloud document
    if cloudDocument == nil
    {
        let item:NSMetadataItem? = searchQuery!.results[0] as?
            NSMetadataItem

        if let unwrappedItem = item {
            let url = unwrappedItem.valueForAttribute(NSMetadataItemURLKey)
                      as! NSURL
            cloudDocument = SwiftCloudTestDocument(fileURL: url)
        }
    }

    cloudDocument!.openWithCompletionHandler {
        (BOOL success) -> Void in
        if (success) {
            let alert = UIAlertController(title: "",
            message: "iCloud document loaded!",
            preferredStyle: UIAlertControllerStyle.Alert)

            alert.addAction(UIAlertAction(title: "Ok",
            style: UIAlertActionStyle.Default,
            handler: nil))

            self.presentViewController(alert,
            animated: true,
            completion: nil)
        }
        else {
            let alert = UIAlertController(title: "",
            message: "Could not load iCloud document!",
            preferredStyle: UIAlertControllerStyle.Alert)

            alert.addAction(UIAlertAction(title: "Ok",
            style: UIAlertActionStyle.Default,
            handler: nil))

            self.presentViewController(alert,
```

```
            animated: true,
            completion: nil)
        }
    }
}
```

The preceding implementation first stops the query from running constantly. If the query did not return any results, it calls the `createDocument` method of the view controller class to create a new document on iCloud; otherwise, it loads the existing document from iCloud.

6. Implement the `onSaveDocument()` method as follows:

```
@IBAction func onSaveDocument(sender: AnyObject) {

    if cloudDocument == nil {
        return
    }

    documentContentView.resignFirstResponder()

    cloudDocument!.documentContents = documentContentView.text

    cloudDocument!.saveToURL(cloudDocument!.fileURL,
    forSaveOperation: UIDocumentSaveOperation.ForCreating) {
        (Bool success) -> Void in
        if (success) {
            self.cloudDocument!.openWithCompletionHandler(nil)
        }
    }
}
```

This method dismisses the keypad, if it is visible, and saves the `SwiftCloudTestDocument` object to the iCloud document.

7. Implement the `createDocument()` method as follows:

```
func createDocument(){

    if self.cloudDocument == nil {
        let containerURL =
 NSFileManager.defaultManager().URLForUbiquityContainerIdentifier(nil)

        let documentDirectoryURL =
containerURL!.URLByAppendingPathComponent("Documents")

        let documentURL =
documentDirectoryURL.URLByAppendingPathComponent("cloudDocument.txt")

        cloudDocument = SwiftCloudTestDocument(fileURL: documentURL)
    }

    cloudDocument!.documentContents = documentContentView.text
    cloudDocument!.saveToURL(cloudDocument!.fileURL,
        forSaveOperation: UIDocumentSaveOperation.ForCreating) {
        (Bool success) -> Void in
```

```
            if (success) {
                self.cloudDocument!.openWithCompletionHandler(nil)
            }
        }
    }
```

This method is used to create an empty file called `cloudDocument.txt` on iCloud, and is used when the `loadDocument` method could not find a document to load.

8. Implement the `refreshDocumentPreview()` method as follows:

```
func refreshDocumentPreview(notifcation: NSNotification) {
        documentContentView.text = cloudDocument!.documentContents;
}
```

This method is received when the `CloudTestDocument` object loads data from the iCloud document `cloudDocument.txt`. Here, you simply refresh the user interface.

➤ Test your app on an iOS device.

1. Connect your iOS device to your Mac.
2. Select your device from the Target/Device selector in the Xcode toolbar.
3. Ensure the correct value has been selected for the Code Signing Entity build settings of the application target (see Figure 24-17).

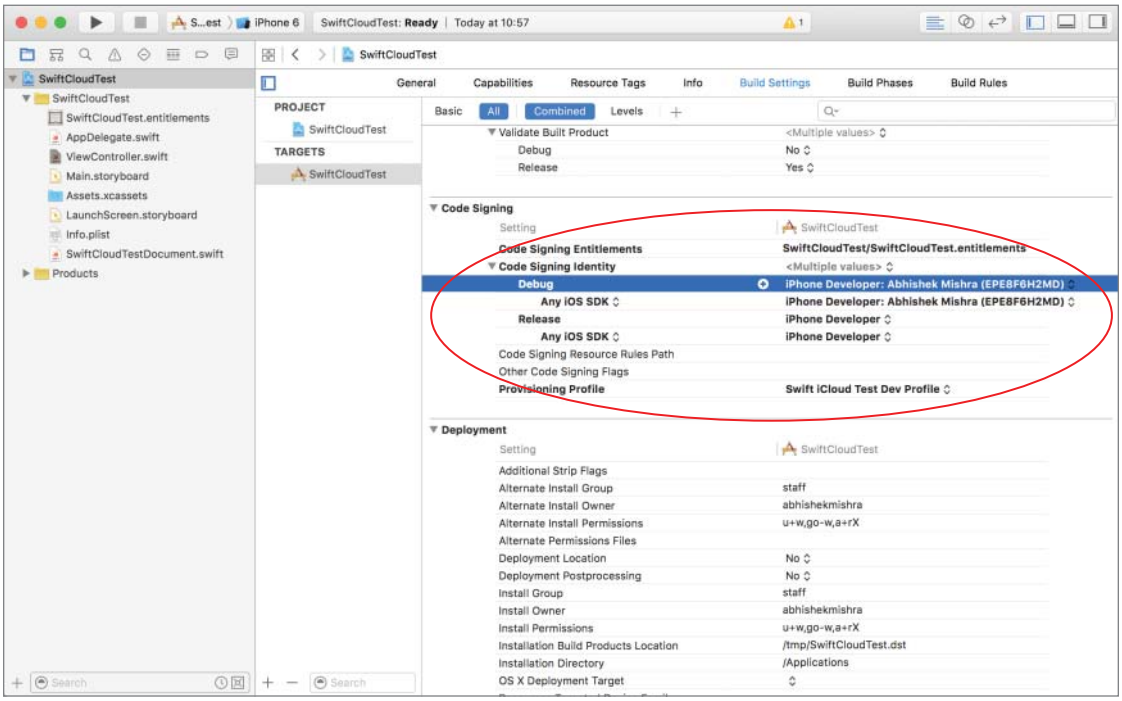

FIGURE 24-17

4. Click the Run button in the Xcode toolbar. Alternatively, you can use the Project ➪ Run menu item.
5. When you run the application for the first time, you will see a message similar to Figure 25-16, telling you that a new iCloud document is going to be created for you.
6. Type some text into the text view and tap the Save Document button.
7. If you now run this application on a different device, you will get a message telling you that an existing iCloud document has been opened.

REFERENCE *To see some of the examples from this lesson, watch the Lesson 24 video online at* `www.wrox.com/go/swiftiosvid`.