Introduction to CloudKit

Storing and Retrieving data from iCloud using CloudKit

Xamarin Inc.

# Brief

iCloud APIs enable iOS 8 applications to store data in iCloud, with support for automatic syncing across a user’s account. Using CloudKit gives users a consistent and seamless experience across iCloud-enabled devices. This article covers enabling CloudKit in an iOS 8 application using the Convenience API.

**Sample Code:**

[TODO: sample code download link]

**Related Articles:**

[TODO: related Xamarin articles list]

**Related Apple Documentation:**

[iCloud for Developers](https://developer.apple.com/icloud/index.html)  
[iCloud Design Guide](https://developer.apple.com/library/ios/icloud_design_guide)  
[CloudKit Framework Reference](https://developer.apple.com/library/prerelease/ios/documentation/CloudKit/Reference/CloudKit_Framework_Reference/index.html)

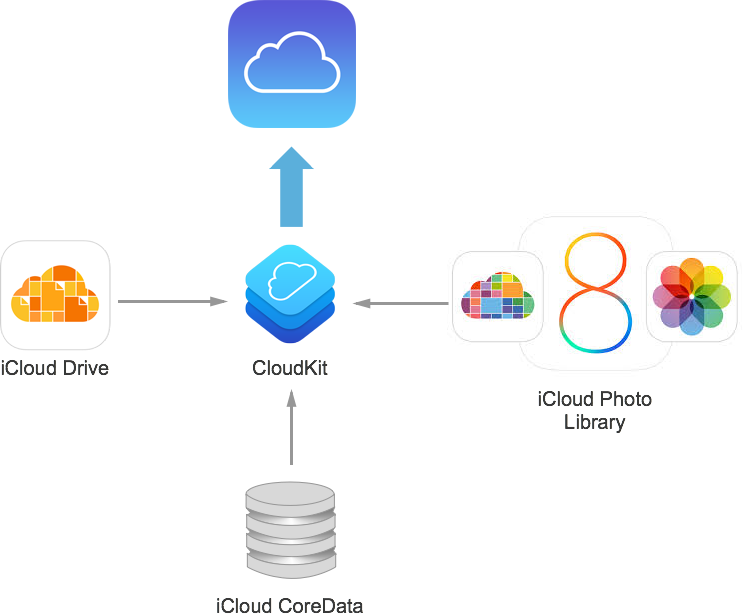
## Overview

Leverage the full power of iCloud and build applications with the new CloudKit Framework. Now the developer can easily and securely store and efficiently retrieve application data like structured data in a database or assets right from iCloud. CloudKit also enables users to anonymously sign in to applications with their iCloud Apple IDs without sharing their personal information.

With CloudKit, the developer can focus on their client-side application development and let iCloud eliminate the need to write server-side application logic. CloudKit provides authentication, private and public databases, and structured data and asset storage services.

### What is CloudKit

CloudKit is a way to give the developer access to the iCloud Servers. It provides the foundation for both iCloud Drive and iCloud Photo Library. CloudKit is supported on both Mac OS X and Apple iOS Devices.



CloudKit uses the iCloud Account infrastructure. If there is a user logged into an iCloud Account on the device, CloudKit will use their ID to identify the user. If no account is available, then limited read-only access will be provided.

CloudKit supports both the concept of public and private databases. Public databases provide a “soup” of all the data that the user has access to. Private databases are meant to store the private data bounds to a specific user.

CloudKit supports both structured and bulk data. It is capable of handling large file transfers seamlessly. CloudKit takes care of efficiently transferring large files to and from the iCloud Servers in the background, freeing the developer to focus on other tasks.

It is important to note that CloudKit is a Transport Technology, it doesn’t provide any persistence, it only enables an application to send and receive information from the servers efficiently.

As of this writing, Apple is initially providing CloudKit for free with a high limit on both bandwidth and storage capacity. For larger projects or applications with a large user base, Apple has hinted that an affordable pricing scale will be provided.

## Requirements

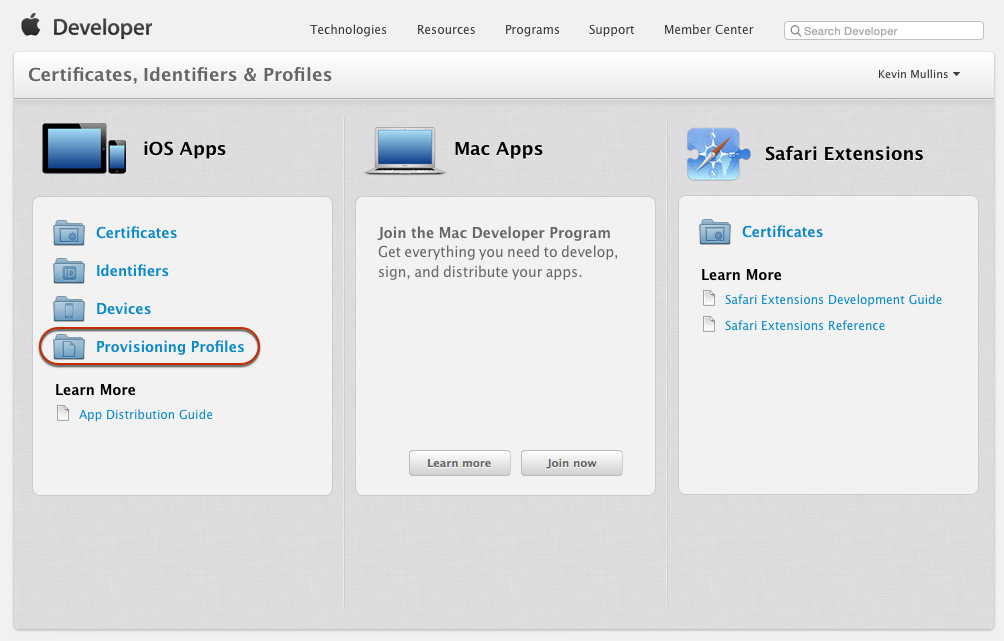
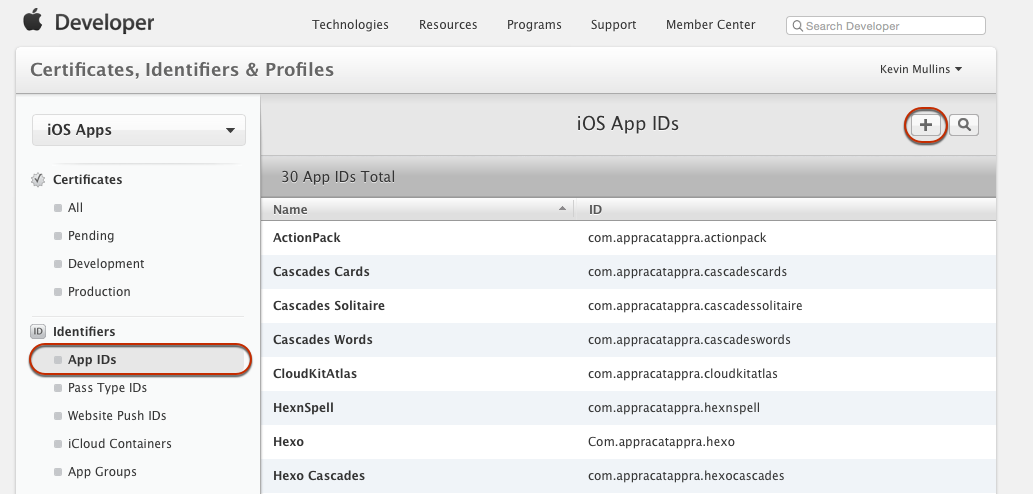
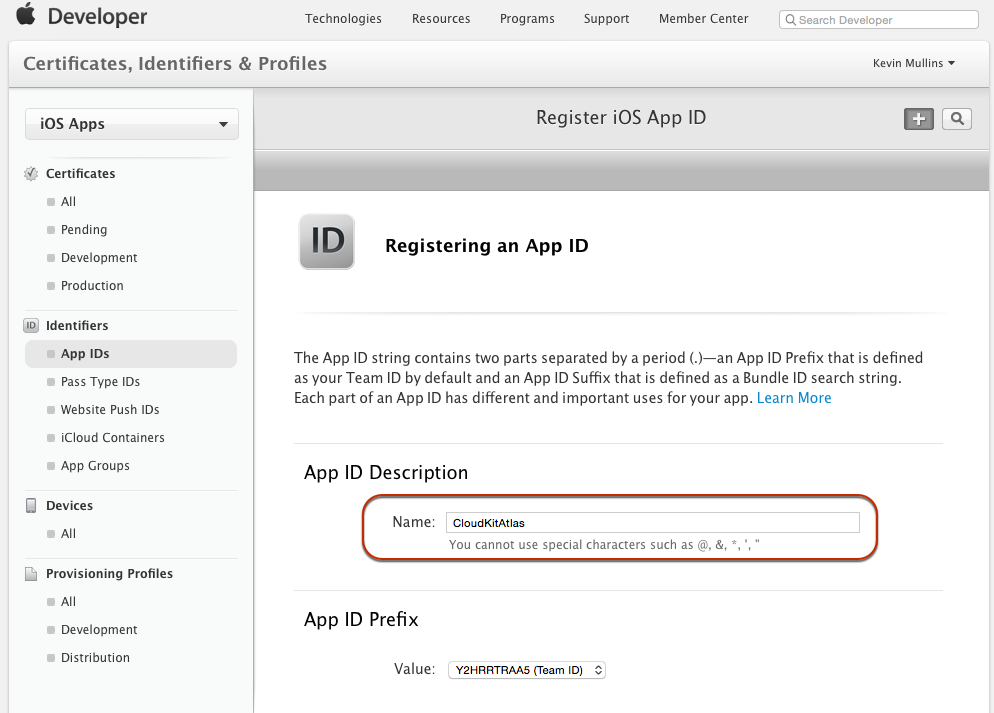
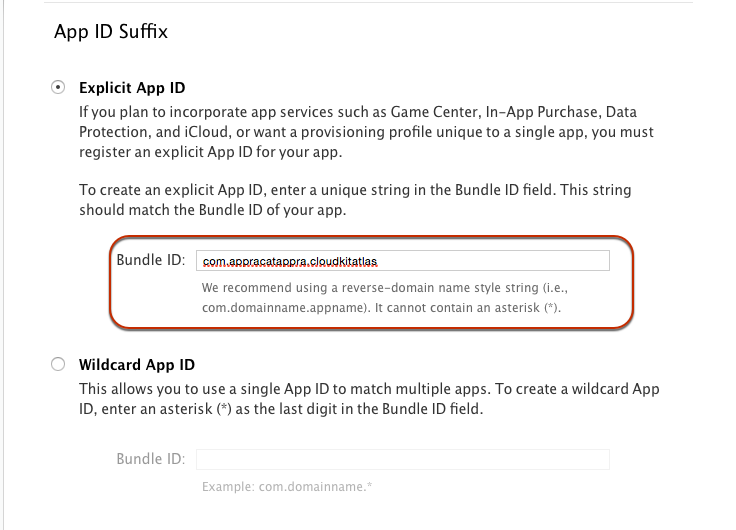
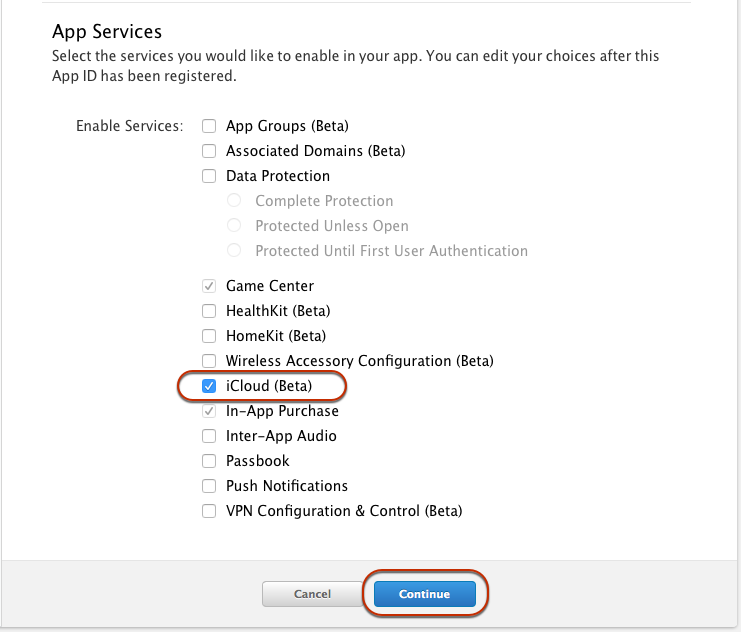
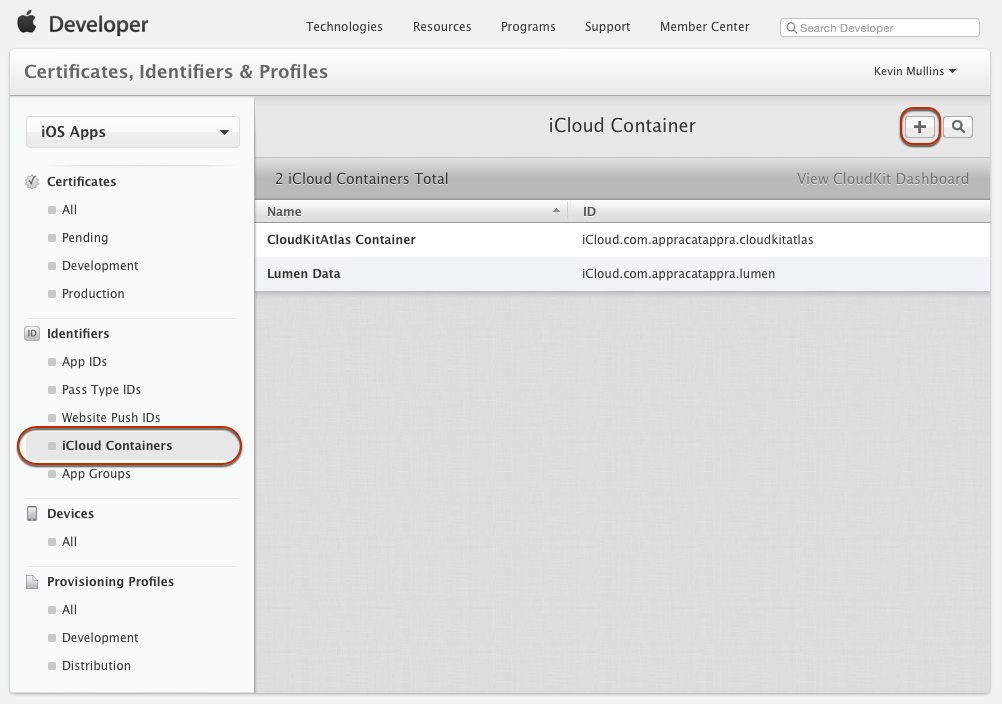
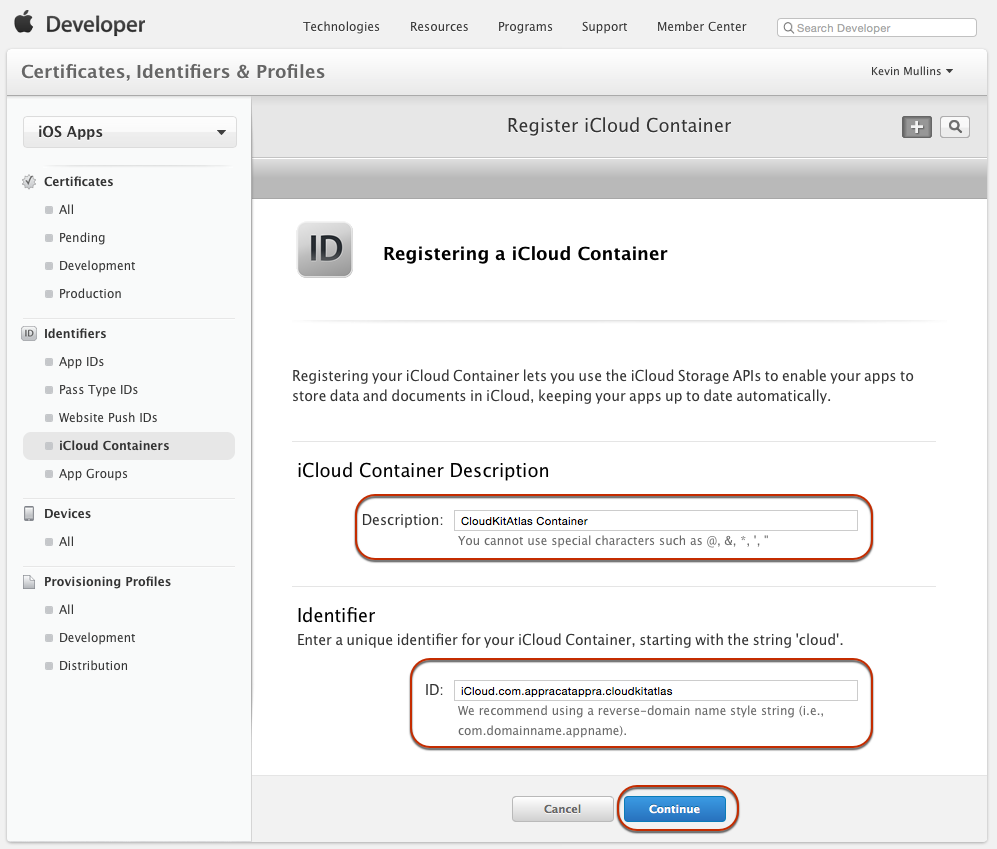
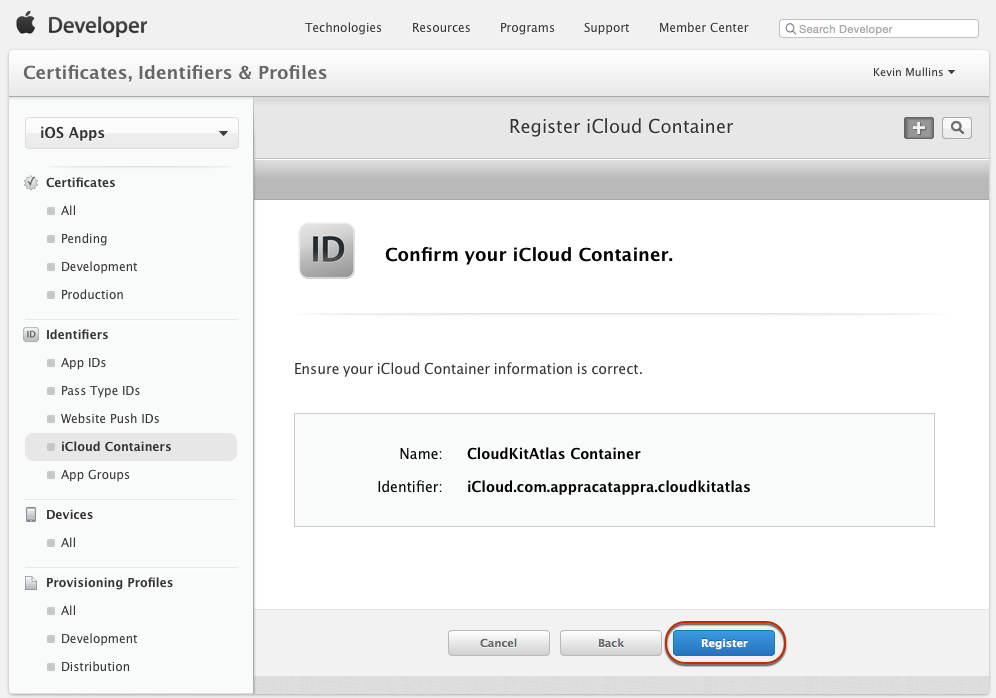
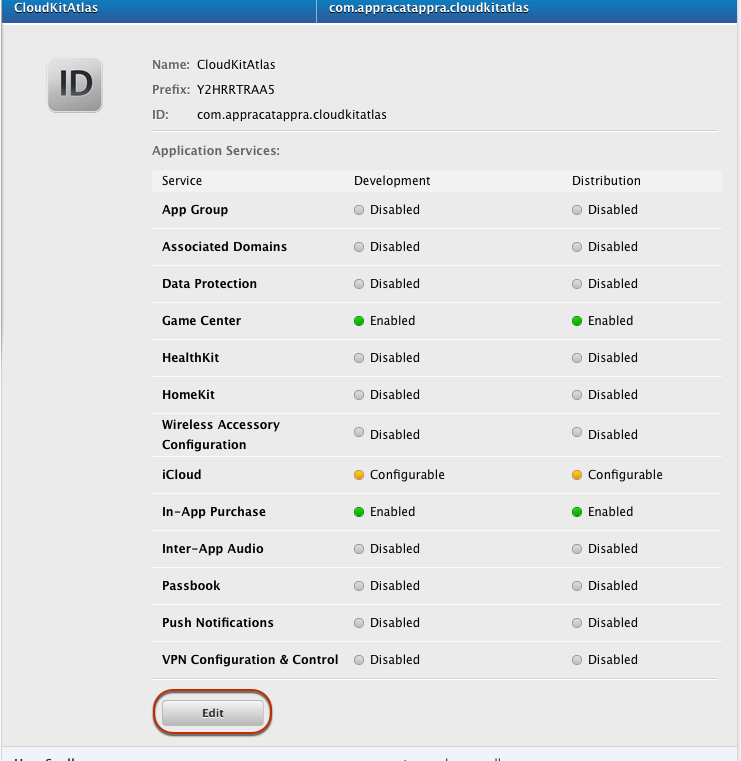
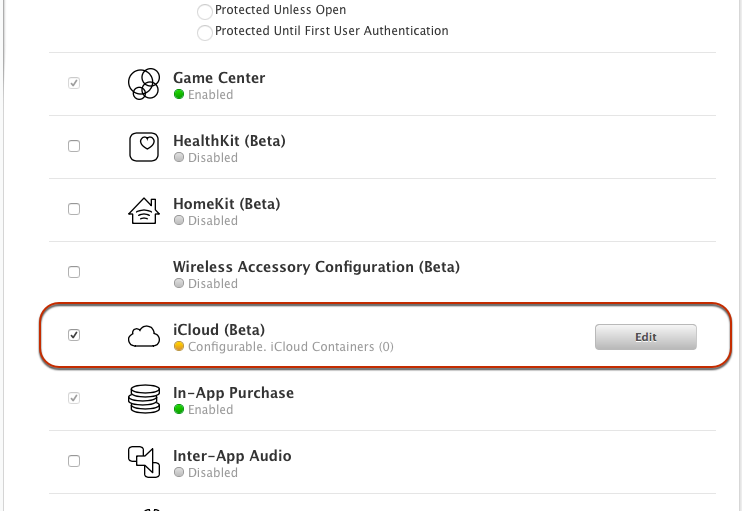
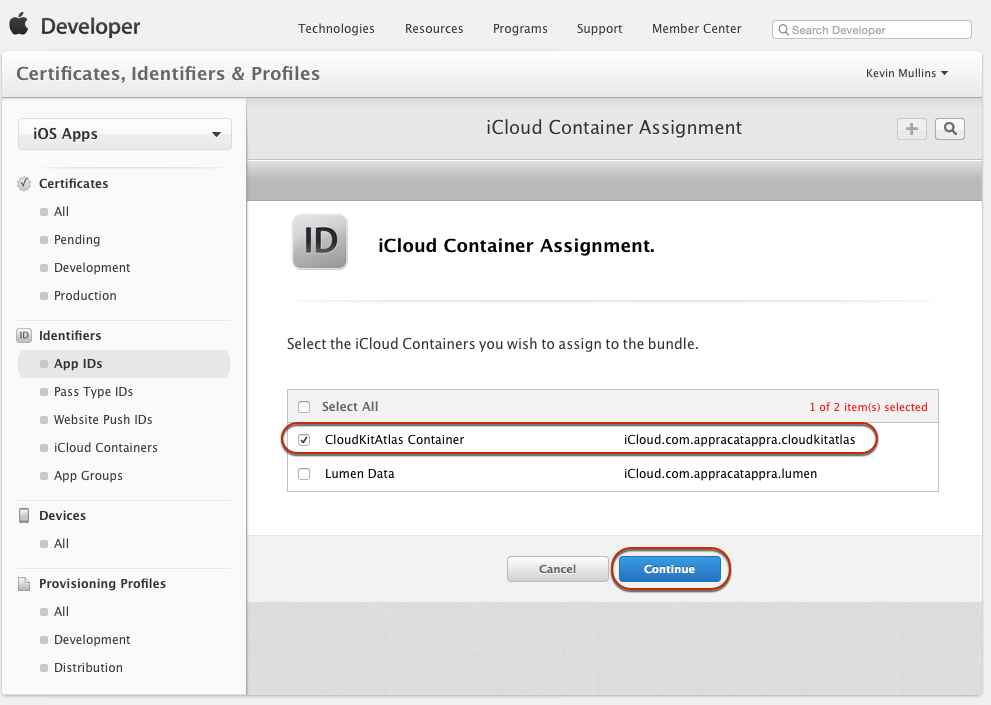
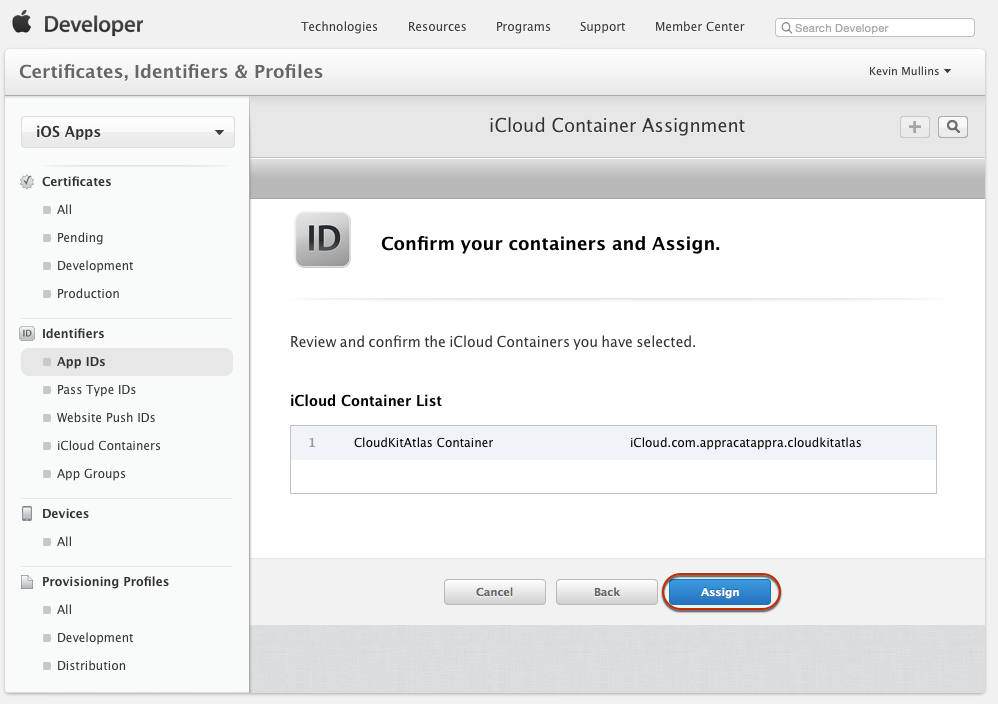
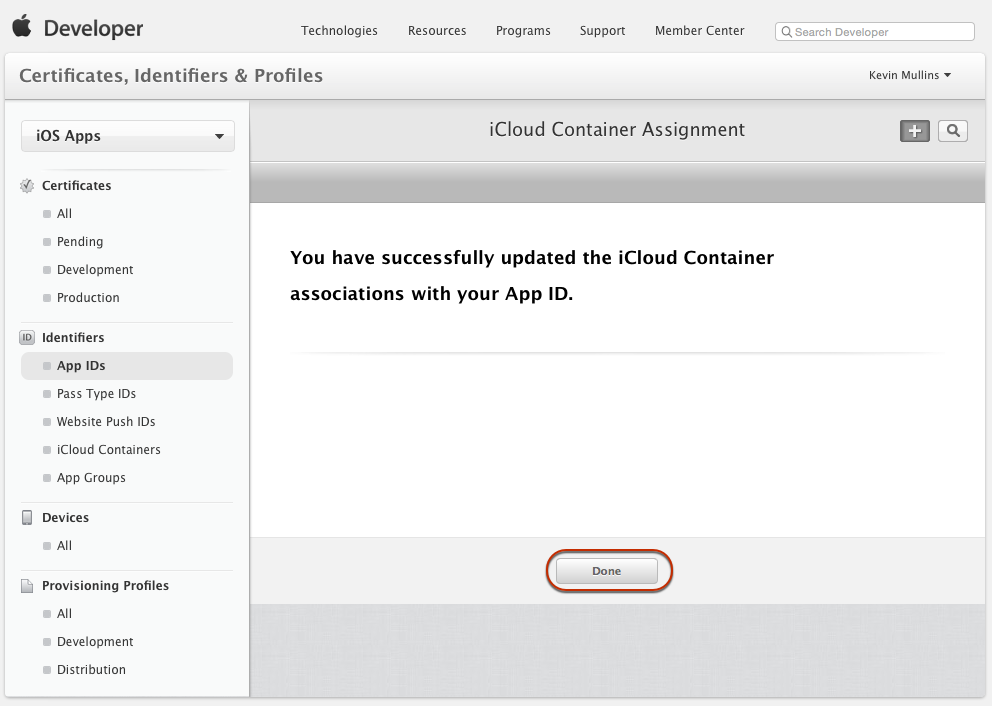
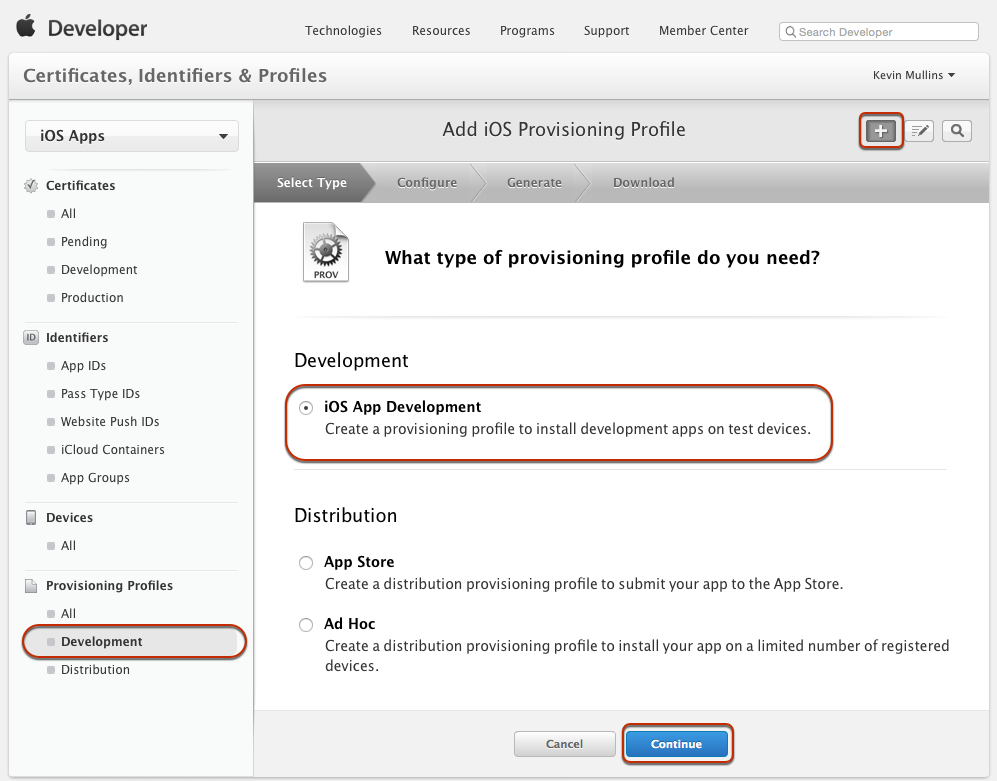
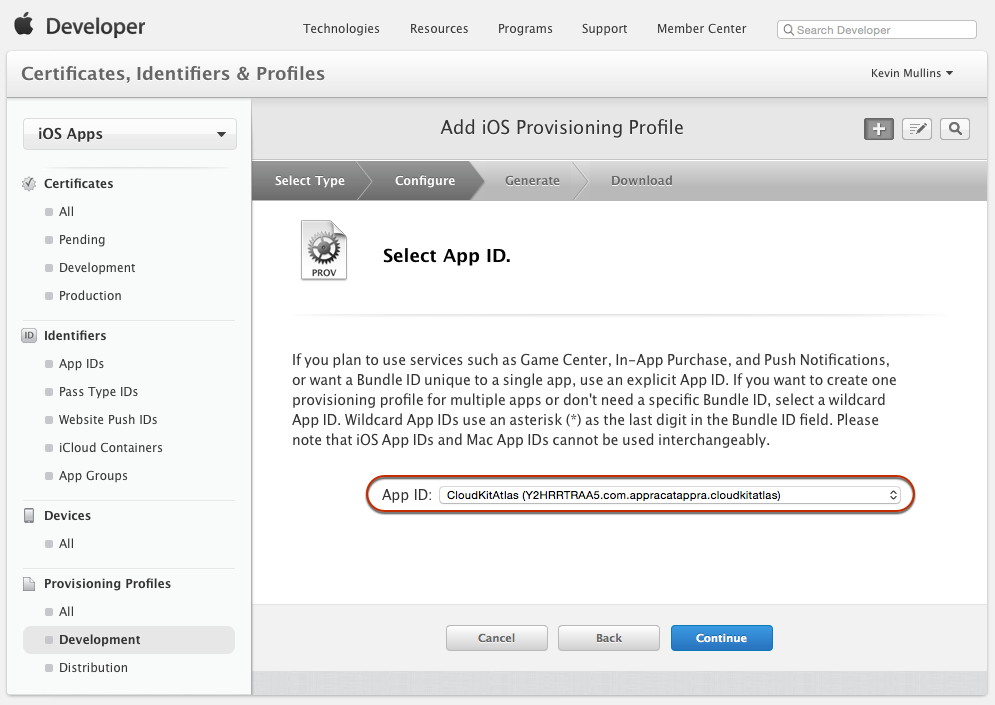
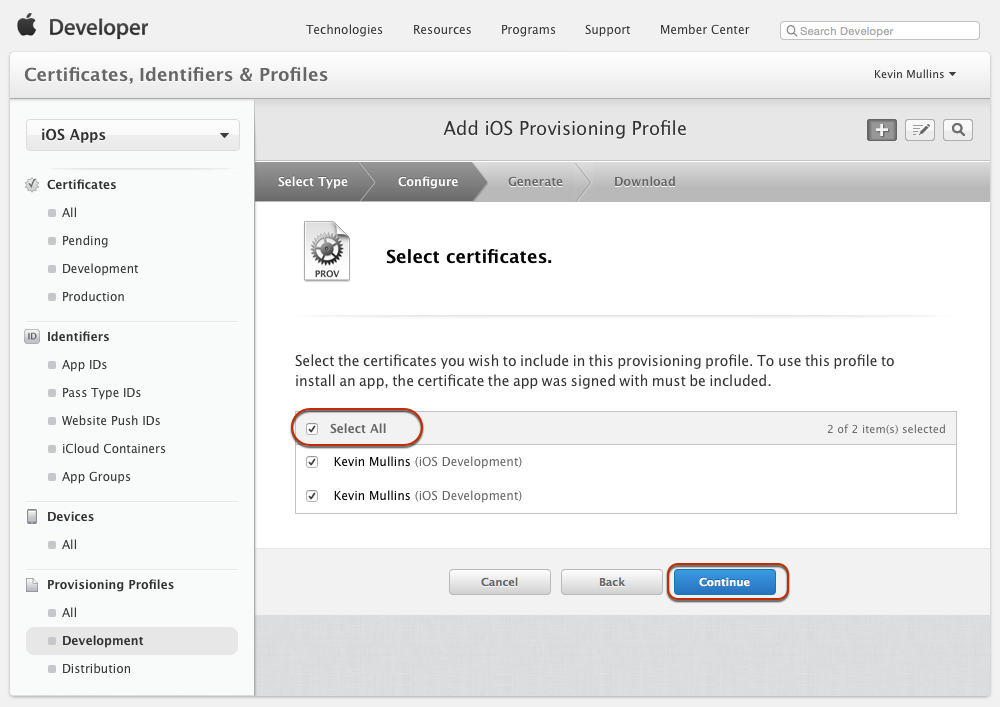
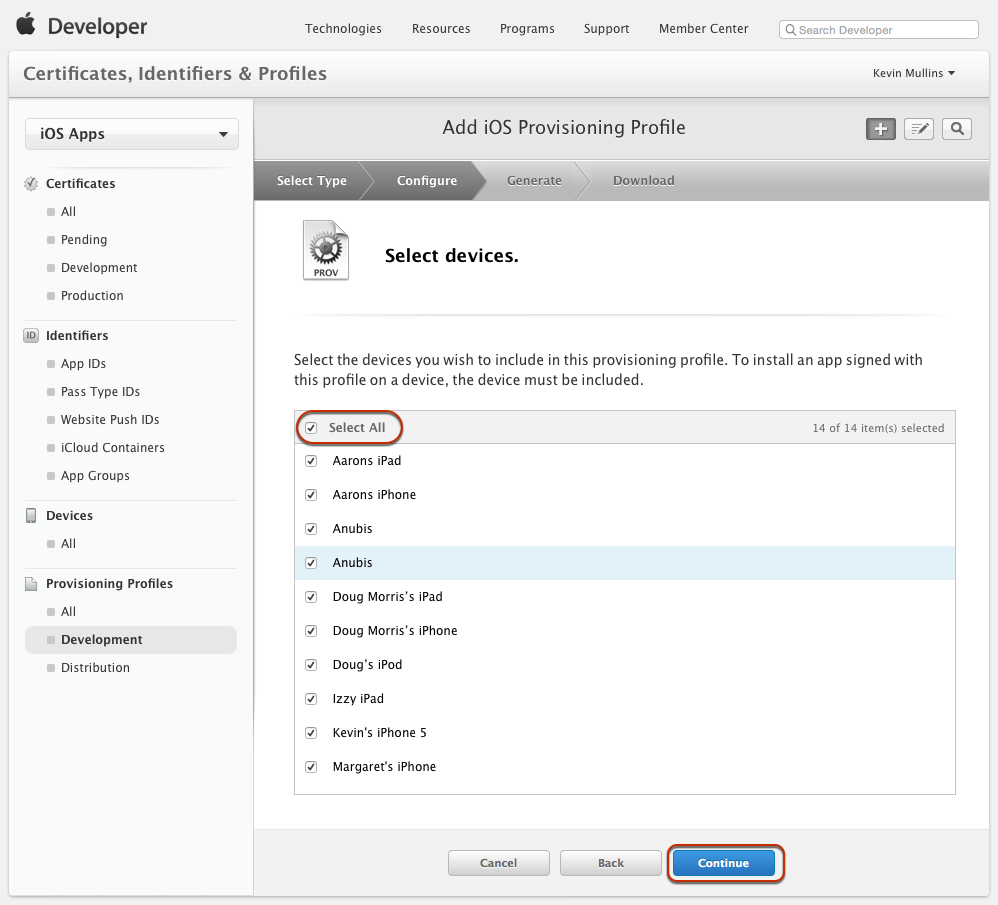
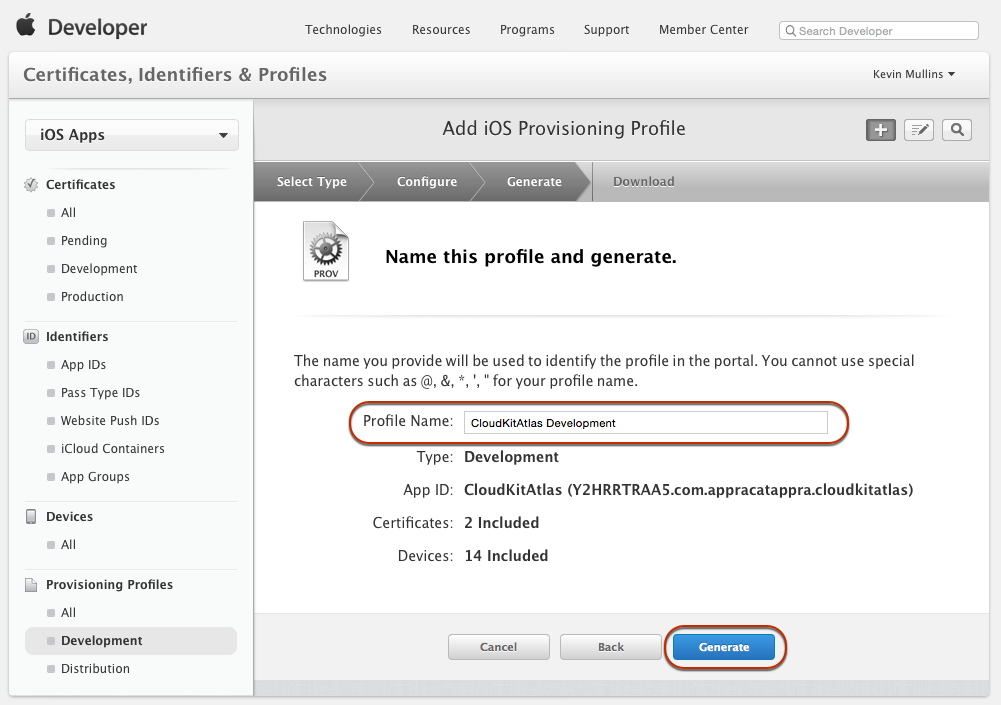
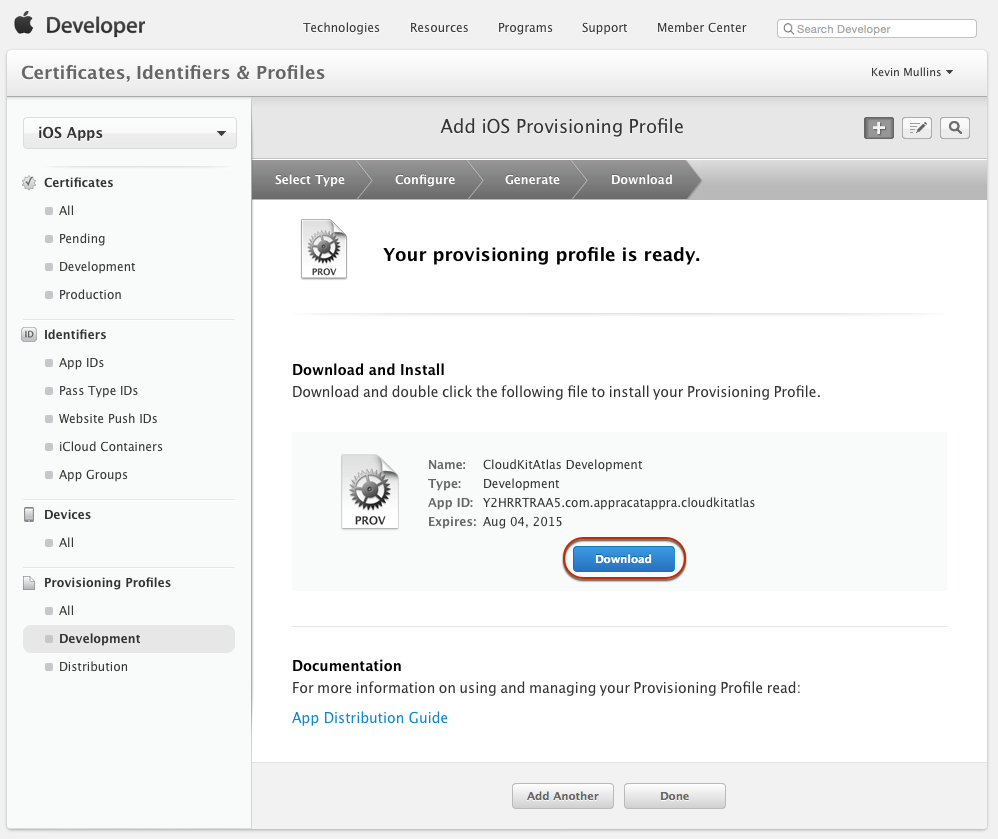
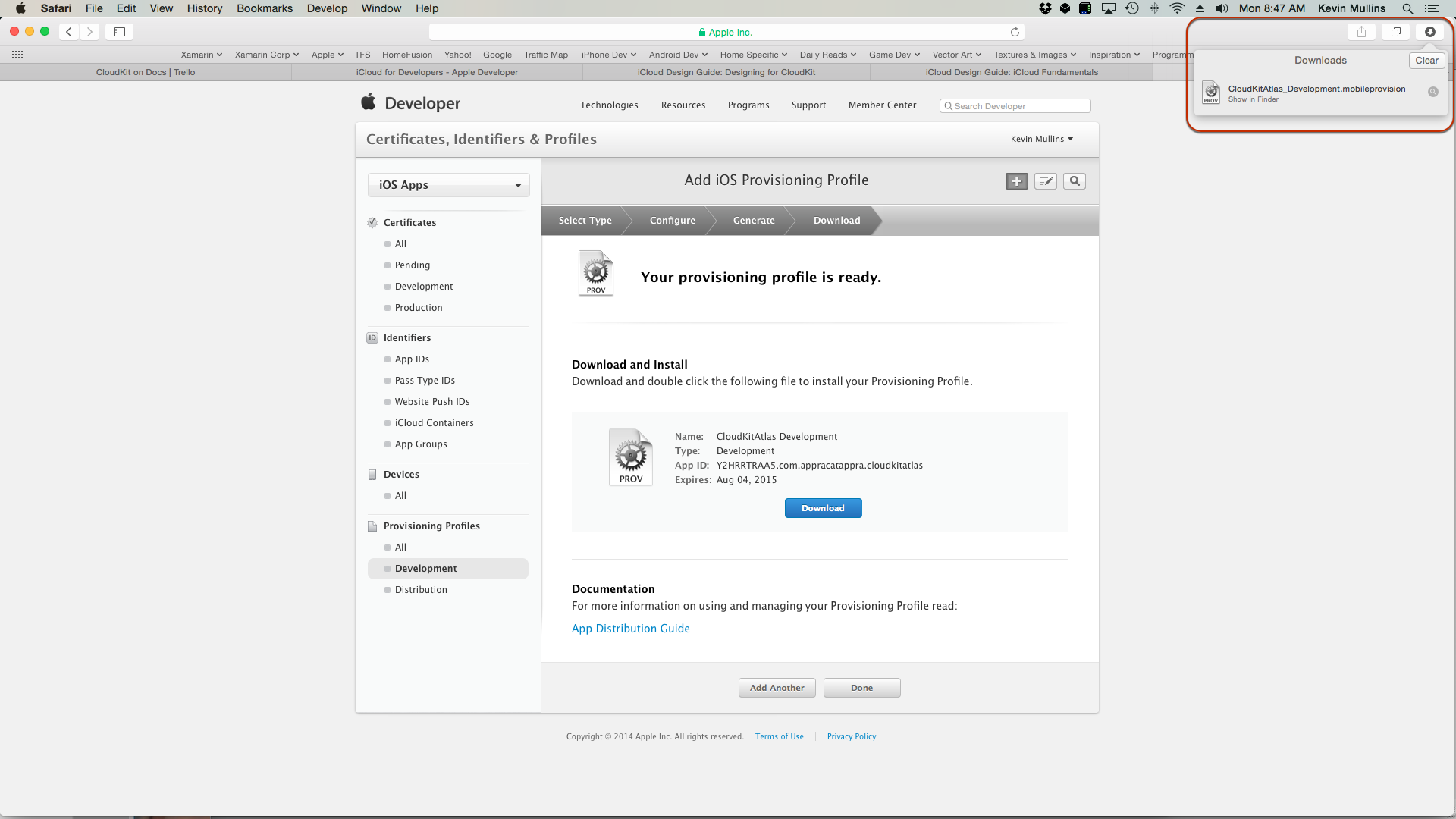
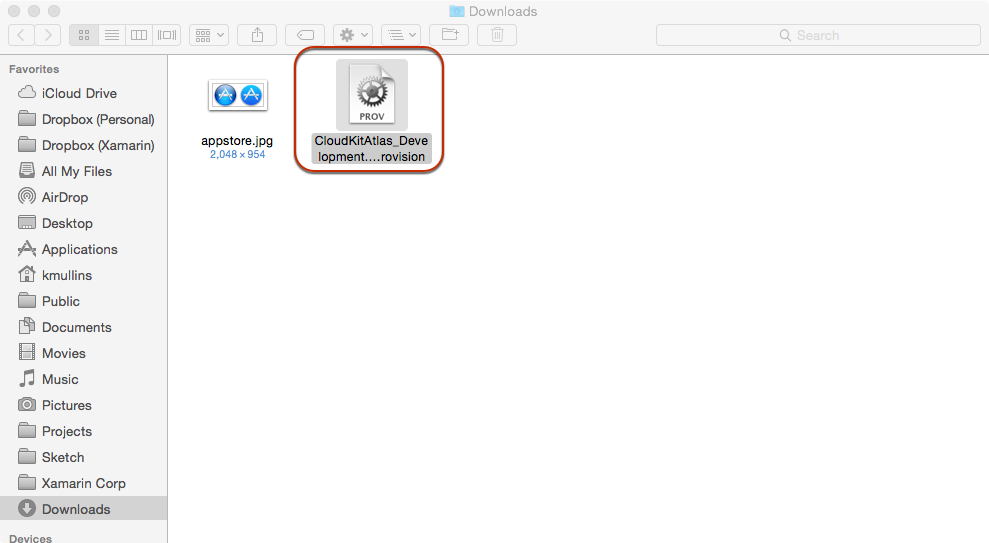
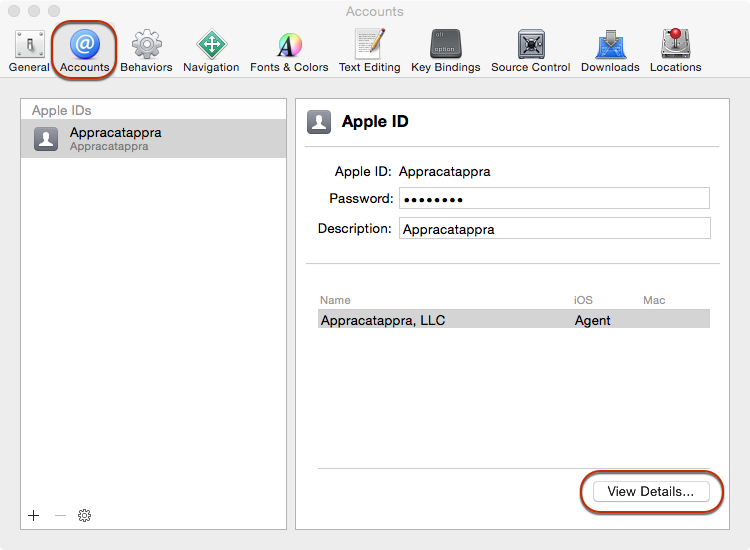
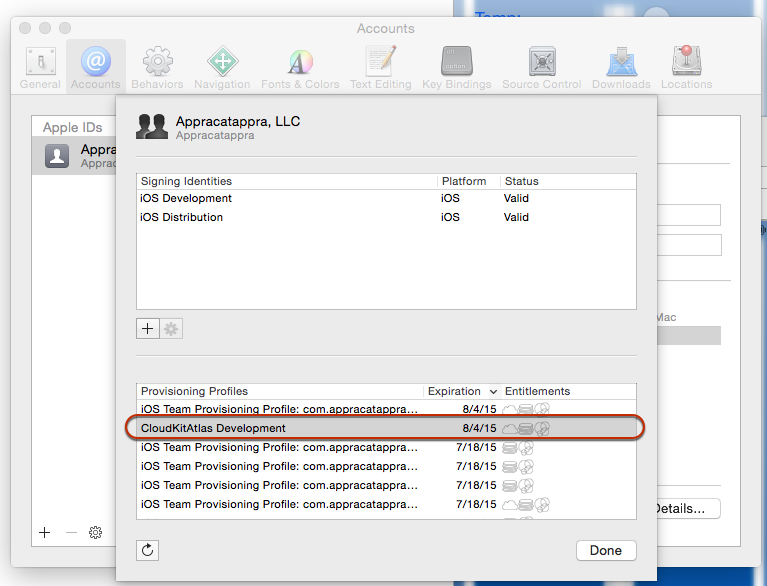
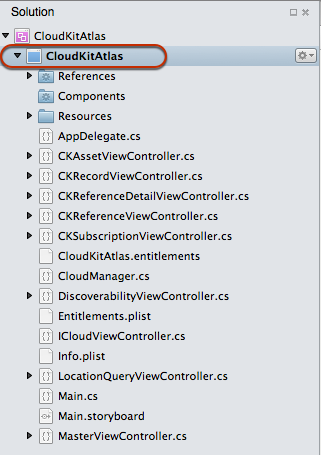
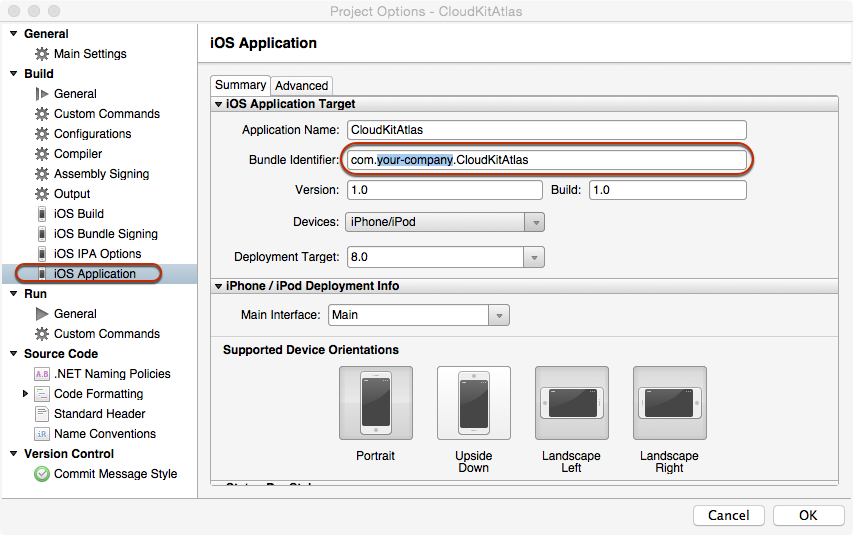
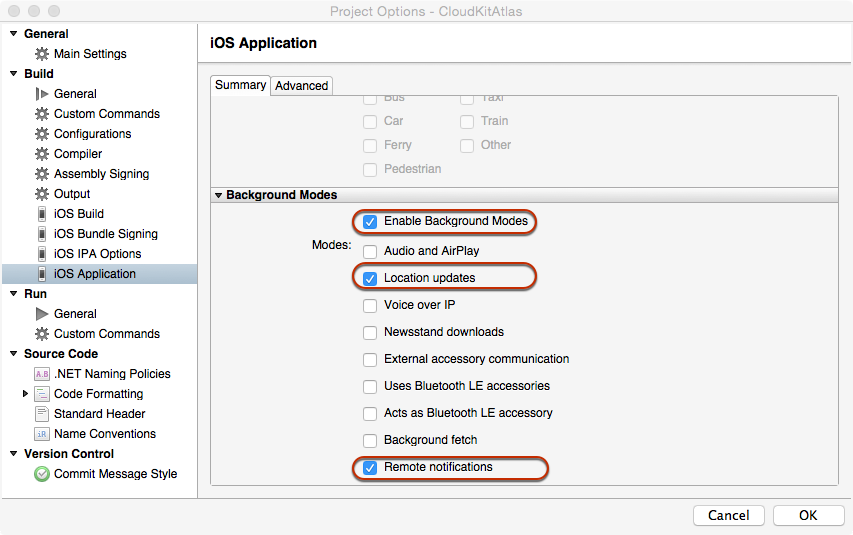
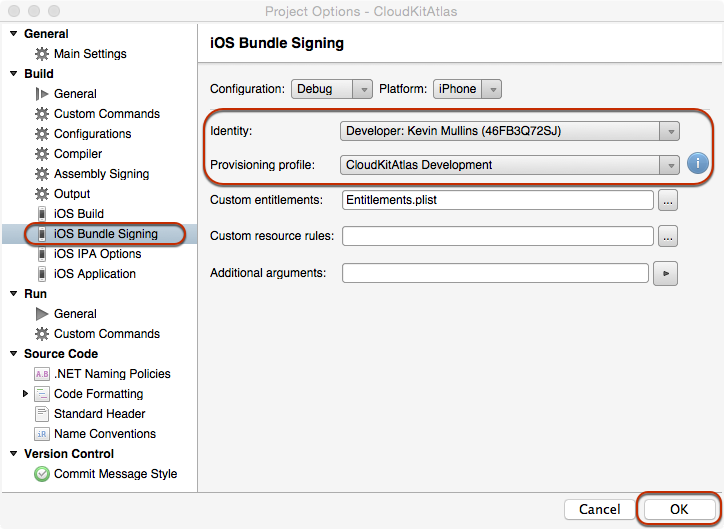
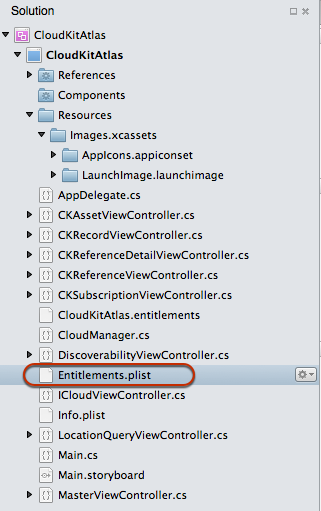
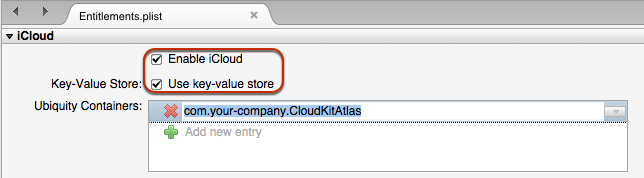
The following is required to complete the steps presented in this article:

* **Xcode 6 and iOS 8** – Apple’s Xcode 6 and iOS 8 APIs need to be installed and configured on the developer’s computer.
* **Xamarin Studio** – The latest version of Xamarin Studio should be installed and configured on the user device.
* **iOS 8 Device** – An iOS device running the latest version of iOS 8 for testing.

## Enabling CloudKit in a Xamarin Application

Before a Xamarin application can utilize the CloudKit Framework, the application must be correctly provisioned, in both the Apple Developer Portal and in Xamarin Studio.

Do the following to enable CloudKit support:

1. In a web browser, navigate to <http://developer.apple.com>.
2. Log into the iOS Development account.
3. Click on **Certificates, Identifiers and Profiles**.
4. Select **Provisioning Profiles**:  
   
5. Select **App IDs**, then click the **+** button:  
   
6. Enter a **Name** for the new Profile:  
   
7. Enter a **Bundle ID** following Apple’s naming recommendation:  
   
8. Scroll down to the **App Services** section, select **iCloud** and click the **Continue** button:  
   
9. Verify all of the settings, then **Submit** the App ID.
10. Select **iCloud Containers** and click the **+** button:  
    
11. Enter a **Name** for the Container, an **ID** using Apple’s naming suggestion and click **Continue**:  
    
12. Click the **Register** button:  
    
13. Return to **App IDs** and select the App ID created above, click the **Edit** button:  
    
14. Select **iCloud** and click the **Edit** Button:  
    
15. Select the **iCloud Container** created above, the click the **Continue** button:  
    
16. Click the **Assign** button:  
    
17. Click the **Done** button:  
    
18. Select **Provisioning Profiles** > **Development**, click the **+** button, select **iOS App Development**, then click **Continue**:  
    
19. Select the **App ID** created above, then click **Continue**:  
    
20. Click **Select All**, then click **Continue**:  
    
21. Click **Select All** again, then click **Continue**:  
    
22. Enter a **Profile Name** using Apple’s naming suggestions, then click **Continue**:  
    
23. Download the new Profile:  
    
24. Reveal the downloaded Provisioning Profile in the file system:  
    
25. Double click the **Provisioning Profile** to open it in Xcode 6:  
    
26. From the Xcode menu select **Preferences…**
27. Select **Accounts**, then click the **View Details…** button:   
    
28. Ensure that the **Provisioning Profile** created above has been installed in Xcode:  
    
29. Open the project to add CloudKit support to in Xamarin Studio.
30. In the **Solution Explorer**, select the Project:  
    
31. Right-click the project and select **Options**.
32. In the **Options Dialog Box** select **iOS Application**, ensure that the Bundle Identifier matches the one that was defined in App ID created above in iTunes Connect for the application:  
    
33. Scroll down to the bottom of the iOS Application Options and select **Enabled Background Modes**, **Location Updates** and **Remote Notifications**:  
    
34. Select **iOS Bundle Signing**, select the developer **Identity** and **Provisioning Profile** created above:  
    
35. Click the **OK** button to save the changes and close the dialog box.
36. Select Entitlements.plist from the **Solution Explorer** and double-click it to open it in the editor:  
    
37. Check both **Enable iCloud** and **Use key-value** store:  
    
38. Click the **+** button and add a **Ubiquity Container**. Enter the Bundle Identifier for the application. Example:  
    com.your-company.CloudKitAtlas
39. Save the changes to the file.
40. In the Solution Explorer, select Entitlements.plist, right-click and select Open With.
41. Select a plain text editor to edit the plist directly and make it look like the following:  
    <?xml version="1.0" encoding="UTF-8"?>  
    <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">  
    <plist version="1.0">  
    <dict>  
    <key>com.apple.developer.ubiquity-container-identifiers</key>  
    <array>  
    <string>$(AppIdentifierPrefix)com.appracatappra.cloudkitatlas</string>  
    </array>  
    <key>com.apple.developer.ubiquity-kvstore-identifier</key>  
    <string>$(AppIdentifierPrefix)$(CFBundleIdentifier)</string>  
    <key>com.apple.developer.icloud-services</key>  
    <array>  
    <string>CloudKit</string>  
    </array>  
    <key>com.apple.developer.icloud-container-identifiers</key>  
    <array>  
    <string>iCloud.$(CFBundleIdentifier)</string>  
    </array>  
    </dict>  
    </plist>
42. Save the changes to the file and return to Xamarin Studio.

With these settings in place, the application is now ready to access the CloudKit Framework APIs.

## The CloudKit API Overview

Before implementing CloudKit in a Xamarin iOS application, this article is going to cover the fundamentals of the Cloudkit Framework. Which will include the following topics:

* **Containers** – Isolated silos of iCloud communications.
* **Databases** – Public and private are available to the application.
* **Records** – The mechanism in which structured data is moved to and from CloudKit.
* **Record Zones** – Are groups of Records.
* **Record Identifiers** – Are fully normalized and represent the specific location of the record.
* **Reference** – Provide parent-child relationships between related Records within a given Database.
* **Assets** – Allow for file of large, unstructured data to be uploaded to iCloud and associated with a given Record.

Next we’ll look at each topic in detail, as it relates to an iOS application using CloudKit.

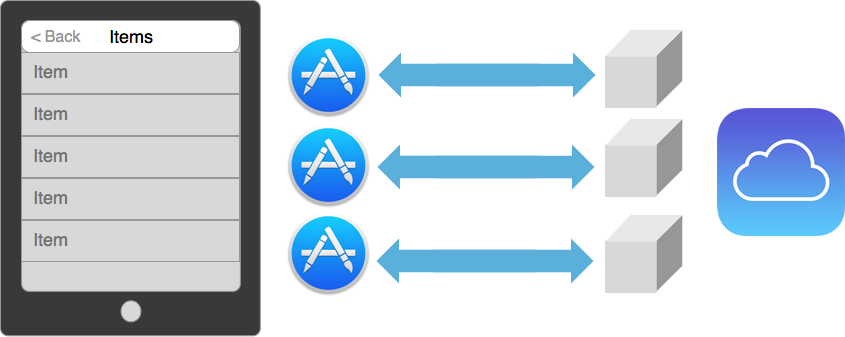
### Containers

A given application running on an iOS device is always running along side other applications and services on that device. On the client device, the application is going to be siloed or sandboxed is some ways. In some cases this is a literal sandbox and in other the application is simply running in it’s own memory space.

The concept of taking a client application and running it separated from other clients is very powerful and provides the following advantages:

* **It helps with security** - One application cannot interfere with other client apps or the OS itself.
* **It helps with stability** – If the client application crashes it cannot take out other apps of the OS.
* **It helps with privacy** – Each client application has limited access to the personal information stored within the device.

CloudKit was designed to provide the same above listed advantages, and apply them to working with cloud-based information.



Just like the application being one-of-many running on the device, so is the application’s communications with iCloud one-of-many. Each of these different communication silos are called Containers.

Containers are exposed in the CloudKit Framework via the CKContainer class. By default, one application talks to one container and this container segregates the data for that application. This means that several applications can be storing information to the same iCloud account, yet this information will never be intermingled.

The containerization of iCloud data also allows CloudKit to encapsulate user information. In this way, the application will have some limited access to the iCloud account and the user information stored within, all while still protecting the privacy and security of the user.

Containers are fully managed by the developer of the application via the WWDR portal. The namespace of the container is global across all Apple developers, so the container not must not only be unique to a given developer’s applications, but to all Apple developers and applications.

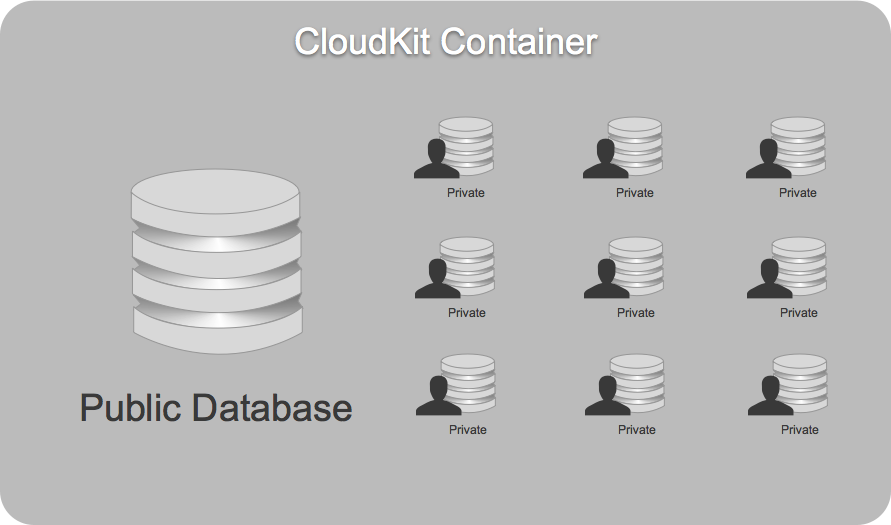
Apple suggests using reverse DNS notation when creating the namespace for application Containers. Example:

iCloud.com.company-name-.application-name

While Containers are, by default, bound one-to-one to a given application, they can be shared across applications. So multiple applications can coordinate on a single Container. A single application can also talk to multiple Containers.

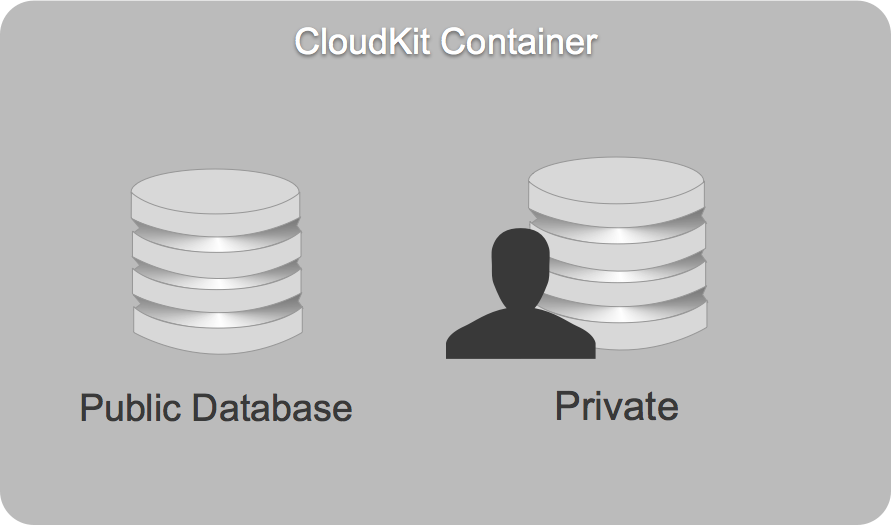
### Databases

One of the primary functions of CloudKit is to take an application’s data model and replication that information up to the iCloud servers. Some information is intended to be used by the user that created it, other information is public data that could be created by a user for public use (like a restaurant review) or it could be information that the developer has published for the application. In either case, the audience is not just a single user, but a community of people.



Inside of a Container, first and foremost is the public database. This is where all of the public information lives and co-mingles. Additionally, there are several individual private databases for each user of the application.

When running on an iOS device, the application will only have access to the information for the currently logged-on iCloud user. So the application’s view of the container will be as follows:



It can only see the public database and the private database associated with the currently logged-on iCloud user.

Databases are exposed in the CloudKit Framework via the CKDatabase class. Every application has access to two databases: the public database and the private one.

The Container is the initial entry point into CloudKit. The following code can be used to access the public and private database from the application’s default Container:

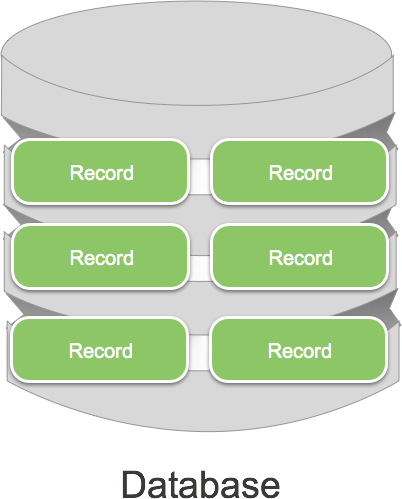
using MonoTouch.CloudKit;  
...  
  
public CKDatabase PublicDatabase { get; set; }  
public CKDatabase PrivateDatabase { get; set; }  
...  
  
// Get the default public and private databases for  
// the application  
PublicDatabase = CKContainer.DefaultContainer.PublicCloudDatabase;   
PrivateDatabase = CKContainer.DefaultContainer.PrivateCloudDatabase;

Here are the following differences between the two database types:

|  |  |  |
| --- | --- | --- |
|  | Public Database | Private Database |
| Data Type | Shared Data | Current User’s Data |
| User Account | Required for writing | Required |
| Quota | Accounted for in the Developer’s Quota | Accounted for in the User’s Quota |
| Default Permissions | World Readable | User Readable |
| Editing Permissions | iCloud Dashboard Roles via a record-class level | N/A |

### Records

Containers hold databases, and inside databases are records. Records are the mechanism in which structured data is moved to and from CloudKit:



Records are exposed in the CloudKit Framework via the CKRecord class, which wraps key-value pairs. An instance of an object in an application is equivalent to a CKRecord in CloudKit. In addition, each CKRecord posses a record type, which is equivalent to the class of an object.

Records have a just-in-time schema, so the data is described to CloudKit before being handed off for processing. From that point, CloudKit will interpret the information and handle the logistics of storing and retrieving the record.

The CKRecord class also supports a wide range of metadata. For example, a record contains information about when it was created and the user that created it. A record also contains information about when it was last modified and the user that modified it.

Records contain the notion of a Change Tag. This is a previous version of a revision of a given record. The Change Tag is used as a light-weight way of determining if the client and the server have the same version of a given record.

As stated above, CKRecords wrap key-value pairs and as such, the following types of data can be stored in a record:

* NSString
* NSNumber
* NSData
* NSDate
* CLLocation
* CKReferences
* CKAssets

In addition to the single value types, a record can contain a homogenous array of any of the above listed types.

The following code can be used to create a new record and store it in a database:

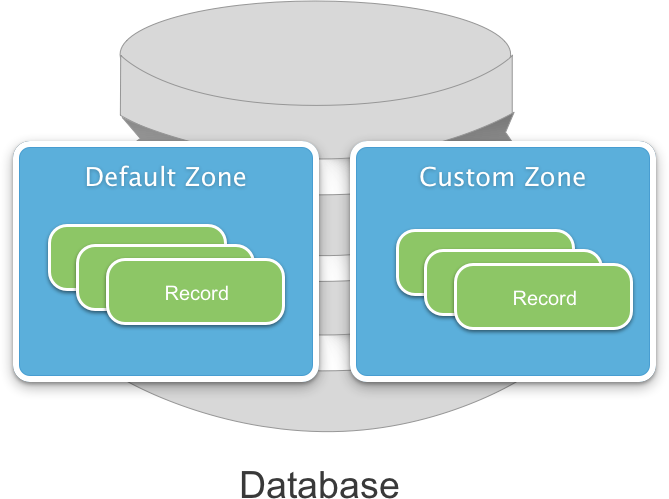
using MonoTouch.CloudKit;  
...  
  
private const string ReferenceItemRecordName = "ReferenceItems";  
...  
  
var newRecord = new CKRecord (ReferenceItemRecordName);  
newRecord ["name"] = (NSString)nameTextField.Text;  
await CloudManager.SaveAsync (newRecord);

### Record Zones

Records don’t exist by themselves within a given database, groups of records exist together inside a Record Zone. Record Zones can be thought of like Tables in a traditional relational databases:



There can be multiple records within a given Record Zone and multiple Record Zones within a given database. Every database contains a Default Record Zone:



This is where are records are stored by default. In addition, Custom Record Zones can be created. Record Zones represent the base granularity at which Atomic Commits and Change Tracking is done.

### Record Identifiers

Record Identifiers are represented as a tuple, containing both a client provided Record Name and the zone in which the record exists. Record Identifiers have the following characteristics:

* They are created by the client application.
* They are fully normalized and represent the specific location of the record.
* By assigning the Unique ID of a record in a foreign database to the record name, they can be used to bridge local databases that are not stored within CloudKit.

When the developer creates a new record they can choose to pass in a Record Identifier. If a Record Identifier is not specified, a UUID will automatically be created and assigned to the record.

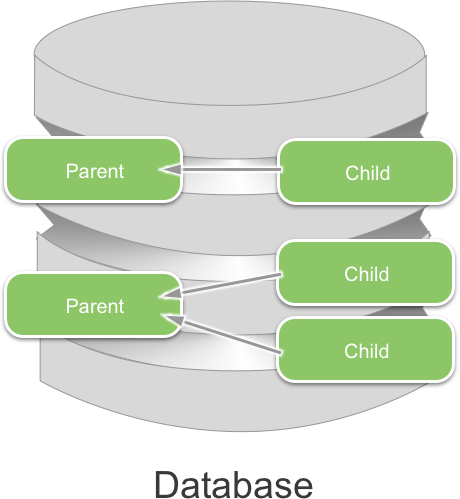
When the developer creates a new Record Identifier, they can choose to specify the Record Zone that the record will belong to. If none is specified, the Default Record Zone will be used.

Record Identifiers are exposed in the CloudKit Framework via the CKRecordID class. The following code can be used to create a new Record Identifier:

var recordID =  new CKRecordID("My Record");

### References

References provide relationships between related Records within a given Database:



In the above example, Parents own Children so that the Child is a child record of the parent record. The Relationship goes from the child record to the parent record and is referred to as a Back Reference.

References are exposed in the CloudKit Framework via the CKReference class. They are a way of letting the iCloud server understand the relationship between records.

References provide the mechanism behind Cascading Deletes. If a parent record is deleted from the database, any child records (as specified in a Relationship) will automatically be deleted from the database as well.

Note: Dangling Pointers are a possibility when using CloudKit. For example, by the time the application has fetched a list of record pointers, selected a record and then asked for the record, the record might no longer exist in the database. An application *must* be coded to handle this situation gracefully.

While not required, Back References are preferred when working with the CloudKit Framework. Apple has fine-tuned the system to make this the most efficient type of reference.

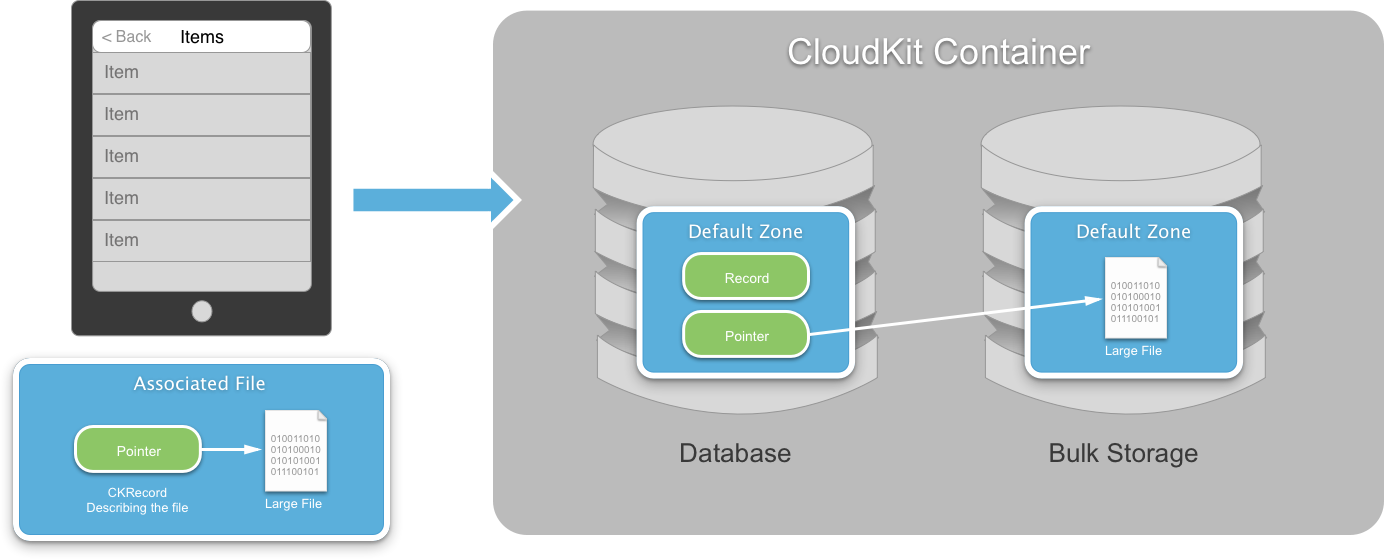
When creating a Reference the developer can either provide a record that is already in memory or create a reference between a Record Identifier. If using a Record Identifier and specified reference didn’t exist in the database, a Dangling Pointer would be created.

The following is an example of creating a reference against a known Record:

var reference = new CKReference(newRecord,new CKReferenceAction());

### Assets

Assets allow for file of large, unstructured data to be uploaded to iCloud and associated with a given Record:



On the client a CKRecord is created that describes the file that is going to be uploaded onto the iCloud server. A CKAsset is created to contain the file and is linked to the record describing it.

When the file is uploaded to the server, the record is placed in the database and the file is copied into a special Bulk Storage database. A link is created between the record Pointer and the uploaded file.

Assets are exposed in the CloudKit Framework via the CKAsset class and are used to store large, unstructured data. Because the developer never wants to have large, unstructured data in memory, Assets are implemented using files on disk.

Assets are owned by records, which allows the assets to be retrieved from iCloud using the record as a pointer. In this way the server can Garbage Collect Assets when the record that owns the Asset is deleted.

Because CKAssets are intended handle large data files, Apple designed CloudKit to efficiently upload and download the Assets.

The following code can be used to create an Asset and associate it with the Record:

var fileUrl = new NSUrl("LargeFile.mov");  
var asset = new CKAsset(fileUrl);  
newRecord ["name"] = asset;

We have now covered all of the fundamental objects within Cloudkit. Containers are associated to applications and contain Database. Databases contain Records that are grouped into Record Zones and pointed to by Record Identifiers. Parent-child relationships are defined between Records using References. Finally, large files can be uploaded and associated to Records using Assets.

## CloudKit Convenience API

Apple offers two difference API sets for working with CloudKit:

* **Operational API** – Offers every single feature of CloudKit. For more complex applications, this API provides fine-grained control over CloudKit.
* **Convenience API** – Offers a common, pre-configured subset of CloudKit features. It provides a convenient, easy access solution for including CloudKit functionality in an iOS application.

The Convenience API is usually the best choice for most IOS applications and Apple suggests starting with it first. The rest of this section will cover the following Convenience API topics:

* Saving a Record.
* Fetching a Record.
* Updating a Record.

### Common Setup Code

Before getting started with the CloudKit Convenience API, there is some standard setup code required. Start by modifying the application’s AppDelegate.cs file and make it look like the following:

using System;  
using System.Collections.Generic;  
using System.Linq;  
using MonoTouch.Foundation;  
using MonoTouch.UIKit;  
using MonoTouch.CloudKit;  
  
namespace CloudKitAtlas  
{  
    [Register ("AppDelegate")]  
    public partial class AppDelegate : UIApplicationDelegate  
    {  
        #region Computed Properties  
        public override UIWindow Window { get; set;}  
        public CKDatabase PublicDatabase { get; set; }  
        public CKDatabase PrivateDatabase { get; set; }  
        #endregion  
  
        #region Override Methods  
        public override bool FinishedLaunching (UIApplication application, NSDictionary launchOptions)  
        {  
            application.RegisterForRemoteNotifications ();  
  
            // Get the default public and private databases for  
            // the application  
            PublicDatabase = CKContainer.DefaultContainer.PublicCloudDatabase;   
            PrivateDatabase = CKContainer.DefaultContainer.PrivateCloudDatabase;  
  
            return true;  
        }  
  
        public override void RegisteredForRemoteNotifications (UIApplication application, NSData deviceToken)  
        {  
            Console.WriteLine ("Registered for Push notifications with token: {0}", deviceToken);  
        }  
  
        public override void FailedToRegisterForRemoteNotifications (UIApplication application, NSError error)  
        {  
            Console.WriteLine ("Push subscription failed");  
        }  
  
        public override void ReceivedRemoteNotification (UIApplication application, NSDictionary userInfo)  
        {  
            Console.WriteLine ("Push received");  
        }  
        #endregion  
    }  
}

The above code exposes the public and private CloudKit databases as shortcuts to make them easier to work with in the rest of the application.

Next, add the following code to any view or view container that will be using CloudKit:

using MonoTouch.CloudKit;  
…

#region Computed Properties  
/// <summary>  
/// Returns the delegate of the current running application  
/// </summary>  
/// <value>The this app.</value>  
public AppDelegate ThisApp {  
    get { return (AppDelegate)UIApplication.SharedApplication.Delegate; }  
}  
#endregion

This adds a shortcut to get to the AppDelegate and access the public and private database shortcuts created above.

With this code in place, let’s take a look at implementing the CloudKit Convenience API in a Xamarin iOS 8 application.

### Saving a Record

Using the pattern presented above when discussing Records, the following code will create a new record and use the Convenience API to save it to the public database:

private const string ReferenceItemRecordName = "ReferenceItems";

…  
  
// Create a new record  
var newRecord = new CKRecord (ReferenceItemRecordName);  
newRecord ["name"] = (NSString)nameTextField.Text;  
  
// Save it to the database  
ThisApp.PublicDatabase.SaveRecord(newRecord, (record,err) => {  
    // Was there an error?  
    if (err!=null) {  
        ...  
    }  
});

Three things to note about the above code:

1. By calling the SaveRecord method of the PublicDatabase, the developer doesn’t have to specify how the data is sent, what Zone it’s being written to, etc. The Convenience API is taking care of all of those details itself.
2. The call is asynchronous and provides a callback routine when the call completes either with success or failure. If the call fails, an error message will be provided.
3. CloudKit does not provide local storage/persistence; it is a transfer medium only. So when a request is made to save a Record, it is immediately sent to the iCloud servers.

**Note**: Because of the “lossy” nature of mobile network communications, where connections are constantly being dropped or interrupted, one of the first considerations the developer must make when working with CloudKit is error handling.

### Fetching a Record

With a Record created and successfully stored on the iCloud server, use the following code to retrieve the record:

// Create a record ID and fetch the record from the database  
var recordID = new CKRecordID("MyRecordName");  
ThisApp.PublicDatabase.FetchRecord(recordID,(record,err) =>{  
    // Was there an error?  
    if (err!=null) {  
  
    }  
});

Just like saving the Record, the above code is asynchronous, simple and requires great error handling.

### Updating a Record

After a Record has been fetched from the iCloud servers, the following code can be used to modify the Record and save the changes back to the database:

// Create a record ID and fetch the record from the database  
var recordID = new CKRecordID("MyRecordName");  
ThisApp.PublicDatabase.FetchRecord(recordID,(record,err) =>{  
    // Was there an error?  
    if (err!=null) {  
  
    } else {  
        // Modify the record  
        record["name"] = (NSString)"New Name";  
  
        // Save changes to database  
        ThisApp.PublicDatabase.SaveRecord(record, (r,e) => {  
            // Was there an error?  
            if (e!=null) {  
  
            }  
        });  
    }  
});

The FetchRecord method of the PublicDatabase returns a CKRecord if the call was successful. The application then modifies the Record and calls SaveRecord again to write the changes back to the database.

This section has shown the typical cycle that an application will use when working with the CloudKit Convenience API. The application will save Records to iCloud, retrieve those records from iCloud, modify the Records and save those changes back to iCloud.

## Designing for Scalability

So far this article has looked at storing and retrieving an application’s entire object model from the iCloud servers, every time it is going to be worked with. While this approach works well with a small amount of data and a very small user base, it does not scale well when the amount of information and/or user base increases.

### Big Data, Tiny Device

The more popular an application becomes, the more data in the database and the less feasible it is to have a cache of that entire data on the device. The following techniques can be used to solve this issue:

* **Keep the large data in the Cloud** – CloudKit was designed to handle large data efficiently.
* **Client should only view a slice of that data –** Bring down the bare minimum of data needed to handle any task at a given time.
* **Client views can change –** Because each user has different preferences, the slice of data being displayed can change from user to user and the user’s individual view of any given slice can be different.
* **Client use queries to focus the viewpoint –** Queries allow the user to view a small subset of a larger dataset that exists within the Cloud.

### Queries

As stated above, queries allow the developer to select a small subset of the larger dataset that exists in the Cloud. Queries are exposed in the CloudKit Framework via the CKQuery class.

A query combines three different things: a Record Type (RecordType), a Predicate (NSPredicate) and optionally a Sort Descriptor (NSSortDescriptors). CloudKit supports most of NSPredicate.

#### Supported Predicates

CloudKit supports the following types of NSPredicates when working with Queries:

* NSPredicate.FromFormat(string.Format("name = {0}", recordName))   
  This will match Records where the name is equal to a value stored in a variable.
* NSPredicate.FromFormat(string.Format("{0} = {1}", key, value))   
  This will allow matches to be based on a dynamic key value, so that the key doesn’t have to be know at compile time.
* NSPredicate.FromFormat(string.Format("start > {0}", (NSDate)date))   
  This will match Records where the Record’s value is greater than the given value.
* var location = new CLLocation(37.783,-122.404);  
  var predicate = NSPredicate.FromFormat(string.Format("distanceToLocation:fromLocation(Location,{0}) < 100", location));  
  This will match Records where Record’s location is within 100 meters of the given location.
* NSPredicate.FromFormat(string.Format("ALL tokenize({0}, ‘Cdl’) IN allTokens", “after session”))   
  CloudKit supports a tokenized search. The above call will create two tokens, one for after and another for session. It will return a Record that contains those two tokens.
* NSPredicate.FromFormat(string.Format("start > {0} AND name = {1}", (NSDate)date, recordName))   
  CloudKit supports compound predicates joined using the AND operator.

#### Creating Queries

The following code can be used to create a CKQuery in a Xamarin iOS 8 application:

var recordName = "MyRec";  
var predicate = NSPredicate.FromFormat(string.Format("name = {0}",recordName));  
var query = new CKQuery("CloudRecords",predicate);

First, it creates a Predicate to select only records that match a given name. Then it creates a query that will select Records of the given Record Type that match the Predicate.

#### Performing a Query

Once a Query has been created, use the following code to perform the query and process the returned records:

var recordName = "MyRec";  
var predicate = NSPredicate.FromFormat(string.Format("name = {0}",recordName));  
var query = new CKQuery("CloudRecords",predicate);  
  
ThisApp.PublicDatabase.PerformQuery(query,null,(NSArray results, NSError err) => {  
    // Was there an error?  
    if (err!=null) {  
  
    } else {  
        // Process the returned records  
        for(int i=0;i<results.Count;++i){  
            var record = (CKRecord)results[i];  
        }  
    }  
});

The above code takes the query created above and executes it against the Public Database. Since no Record Zone is specified, all zones are searched. If no errors occurred, an array of CKRecords will be returned matching the parameters of the query.

The way to think about Queries is that they are polls, and are great at slicing through large datasets. Queries, however, are not well suited for large, mostly static datasets because of the following reasons:

* They are bad for the device battery life.
* They are bad for network traffic.
* They are bad for user experience because the information they see is limited by how often the application is polling the database. Users today expect push notifications when something changes.

### Subscriptions

When dealing with large, mostly static datasets, the query should not be performed on the client device, it should run on the server on the client’s behalf. The query should run in the background and should be executed after every single record save, whether by the current device or another device touching the same database.

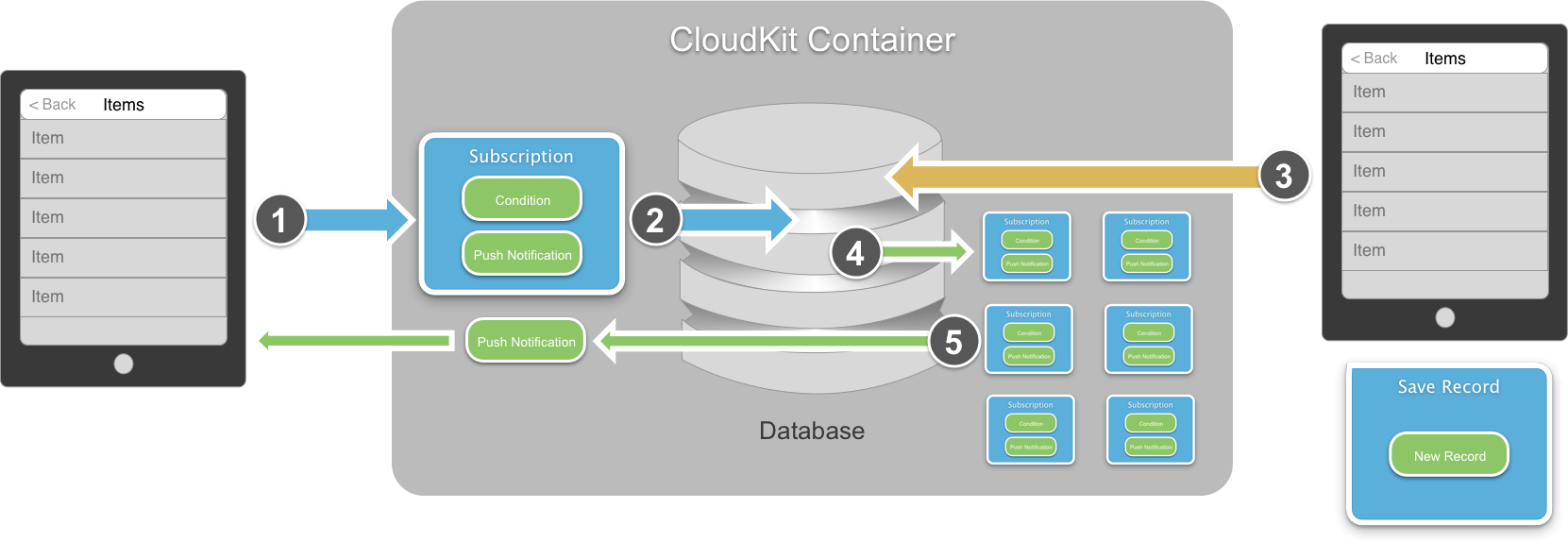
Finally, a push notification should be sent to every device attached to the database when the server-side query is run.

Subscriptions are exposed in the CloudKit Framework via the CKSubscription class. They combine a Record Type (RecordType), a Predicate (NSPredicate) and an Apple Push Notification (Push).

**Note**: CloudKit pushes are slightly augmented as they contain a payload containing CloudKit specific information such as what caused the push to happen.

#### How Subscriptions Work

Before implementing Subscription in C# code, let’s take a quick overview of how subscriptions work:



The above graph shows the typical subscription process as follows:

1. The client device creates a new Subscription containing the set of conditions that will trigger the subscription and a Push Notification that will be sent when the trigger occurs.
2. The Subscription is sent to the Database where it is added to the collection of existing subscriptions.
3. A second device creates a new Record and saves that record to the Database.
4. The Database searches through its list of Subscriptions to see if the new Record matches any of their conditions.
5. If a match is found, the Push Notification is sent to the device that registered the Subscription with information about the Record that caused it to be triggered.

With this knowledge in place, let’s look at creating Subscriptions in a Xamarin iOS 8 application.

#### Creating Subscriptions

The following code can be used to create a Subscription:

// Create a new subscription  
DateTime date;  
var predicate = NSPredicate.FromFormat(string.Format("start > {0}",(NSDate)date));  
var subscription = new CKSubscription("RecordType",predicate,CKSubscriptionOptions.FiresOnRecordCreation);  
  
// Describe the type of notification  
var notificationInfo = new CKNotificationInfo();  
notificationInfo.AlertLocalizationKey = "LOCAL\_NOTIFICATION\_KEY";  
notificationInfo.SoundName = "ping.aiff";  
notificationInfo.ShouldBadge = true;  
  
// Attach the notification info to the subscription  
subscription.NotificationInfo = notificationInfo;

First, it creates a Predicate that provides the condition to trigger the Subscription. Next, it creates the subscription against a specific Record Type and sets the option of when the trigger is tested. Finally it defines the type of notification that will occur when the Subscription is triggered and attaches it the Subscription.

#### Saving Subscriptions

With the Subscription created, the following code will save it to the Database:

// Save the subscription to the database  
ThisApp.PublicDatabase.SaveSubscription(subscription,(s,err) => {  
    // Was there an error?  
    if (err !=null) {  
  
    }  
});

Using the Convenience API the call is asynchronous, simple and provides easy error handling.

#### Handling Push Notifications

If the developer has previously used Apple Push Notifications (APS), then the process of dealing with Notifications generated by CloudKit should be familiar.

In the AppDelegate.cs, override the ReceivedRemoteNotification class as follows:  
public override void ReceivedRemoteNotification (UIApplication application, NSDictionary userInfo)  
{  
    // Parse the notification into a CloudKit Notification  
    var notification = CKNotification.FromRemoteNotificationDictionary (userInfo);  
  
    // Get the body of the message  
    var alertBody = notification.AlertBody;  
  
    // Was this a query?  
    if (notification.NotificationType == CKNotificationType.Query) {  
        // Yes, convert to a query notification and get the record ID  
        var query = notification as CKQueryNotification;  
        var recordID = query.RecordId;  
    }  
}

The above code asks CloudKit to parse the userInfo into a CloudKit Notification. Next, information is extracted about the alert. Finally, the type of notification is tested and the notification is handled accordingly.

This section has shown how to answer the Big Data, Tiny Device problem presented above by using Queries and Subscriptions. The application will leave its large data in the cloud and these technologies to provide views into this dataset.

## CloudKit User Accounts

As noted at the start of this article, CloudKit is built on top of the existing iCloud infrastructure. The following section will cover, in detail, how accounts are exposed to a developer using the CloudKit API.

### Authentication

When dealing with user accounts, the first consideration is authentication. CloudKit support authentication via the currently logged-in iCloud user on the device. Authentication takes place behind the scenes and is handled by iOS. In this way, the developer never has to worry about the details of implementing authentication, they only test to see if a user is logged on.

### User Account Information

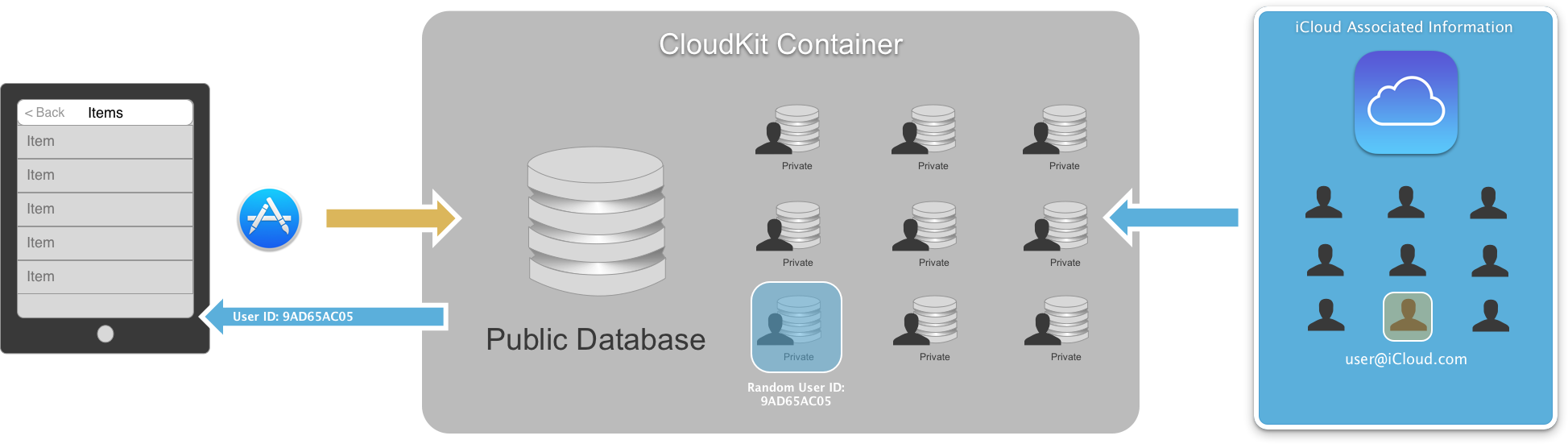
CloudKit provides the following information to the developer:

* **Identity** – a way of uniquely identifying the user.
* **Metadata** – The ability to save and retrieve information about users.
* **Privacy** – All information is handled in a privacy conscious manor. Nothing is exposed unless the user has agreed to it.
* **Discovery** – Gives users the ability to discover their friends that are using the same application.

Next we will look at these topics in detail.

#### Identity

As stated above, CloudKit provides a way for the application to uniquely identify a given user:



There is a client application running on a user’s devices and all of the specific User Private Databases inside the CloudKit Container. The client application is going to be linked to one of those specific users. This is based on the user that is logged into iCloud locally on the device.

Because this is coming from iCloud, there is a rich backing store of User Information. And because iCloud is actually hosting the Container, it can correlate users. In the above graphic, the user who’s iCloud account is [user@iCloud.com](mailto:user@iCloud.com) is linked to the current client.

On a Container by Container basis, a unique, randomly-generated User ID is created and associated with the user’s iCloud account (email address). This User ID is returned to the application and can be used in any way the developer sees fit.

**Note**: Different applications running on the same device for the same iCloud user will have different User IDs because they are connected to different CloudKit Containers.

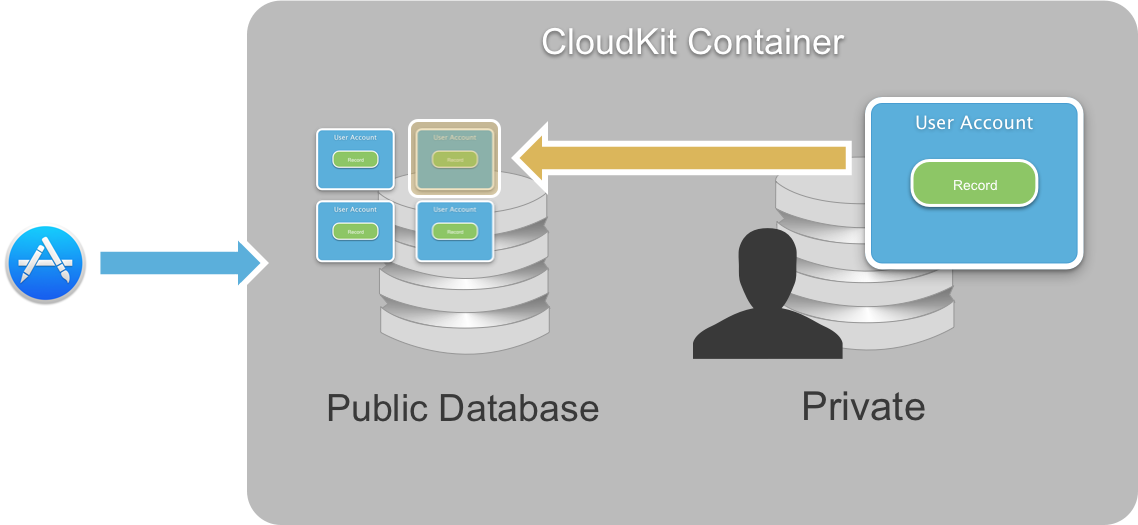
The following code gets the CloudKit user ID for the currently logged in iCloud user on the device:

public CKRecordID UserID { get; set; }  
...  
  
// Get the CloudKit User ID  
CKContainer.DefaultContainer.FetchUserRecordId ((recordID, err) => {  
    // Was there an error?  
    if (err!=null) {  
        Console.WriteLine("Error: {0}",err.LocalizedDescription);  
    } else {  
        // Save user ID  
        UserID = recordID;  
    }  
});

The above code is asking the CloudKit Container to provide the ID of the currently logged in user. Since this information is coming from the iCloud Server, the call is asynchronous and error handling is required.

#### Metadata

Each user in CloudKit has specific Metadata that describes them. This Metadata is represented as a CloudKit Record:



Looking inside the Private Database for a specific user of a Container there is one Record that defines that user. There are many User Records inside the Public Database, one for each user of the Container. One of these will have a record ID that matches the currently logged on user’s Record ID.

User Records in the Public Database are world readable. They are treated, for the most part, as an ordinary Record and have a type of CKRecordTypeUserRecord. These records are reserved by the system and are not available for queries.

Use the following code to access a User Record:

public CKRecord UserRecord { get; set; }  
...  
  
// Get the user's record  
PublicDatabase.FetchRecord(UserID, (record,er) => {  
    //was there an error?  
    if (er!=null) {  
        Console.WriteLine("Error: {0}",er.LocalizedDescription);  
    } else {  
        // Save the user record  
        UserRecord = record;  
    }  
});

The above code is asking the Public Database to return the User Record for the user who’s ID we accessed above. Since this information is coming from the iCloud Server, the call is asynchronous and error handling is required.

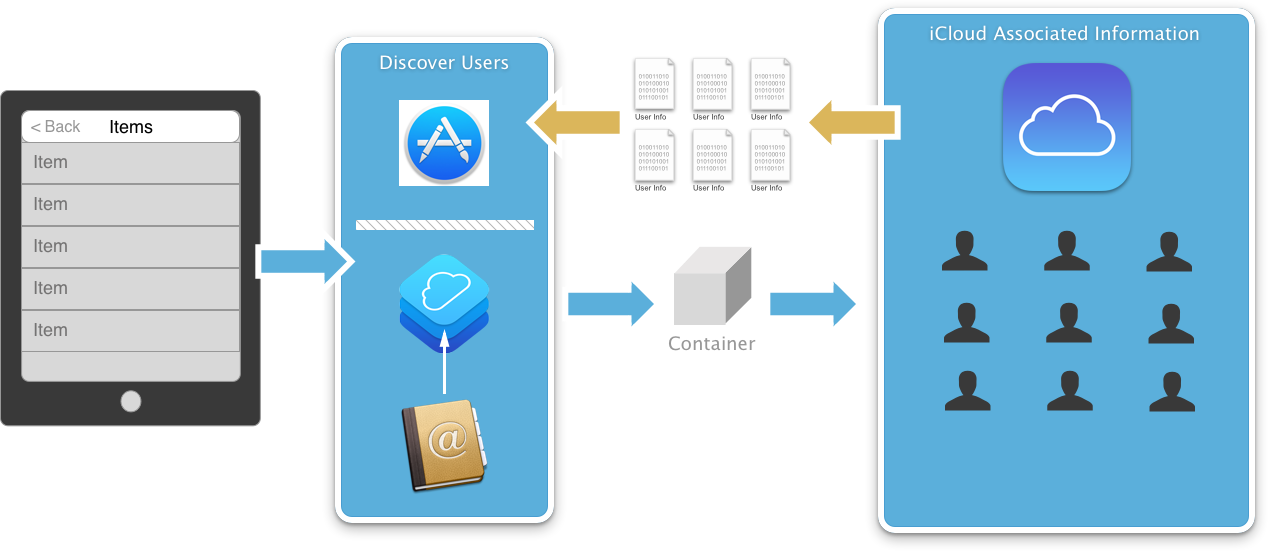
#### Privacy

CloudKit was design, by default, to protect the privacy of the currently logged on user. No personally identifying information about the user is exposed by default. There are some cases where the application will require limited information about the user.

In these cases, the application can request that the user disclose this information. A dialog box will be presented to the user asking them to opt-in to exposing their account information.

#### Discovery

Assuming that the user as opted-in to allowing the application limited access to their user account information, they can be discoverable to other users of the application:



The client application is talking to a Container and the Container is talking iCloud to access user information. The user can provide an email address and Discovery can be used to get information back about the user. Optionally, the User ID can also be used to discover information about the user.

CloudKit also provides a way to discover information about any user who might be friends of the currently logged on iCloud user by querying the Whole Address Book. The CloudKit Process will pull in the user’s Contact Book and use the email addresses to see if it can find other user’s of the application that match those addresses.

This allows the application to leverage the user’s Contact Book without providing access to it or asking the user to approve access to the contacts. At no time is the contact information made available to the application, only the CloudKit Process has access.

To recap, there are three different kinds of inputs available for User Discovery:

* **User Record ID** – discovery can be done against the User ID of the currently logged in CloudKit user.
* **User Email Address** – the user can provide an email address and it can be used for discovery.
* **Contact Book** – The user’s address book can be used to discover users of the application that have the same email address as listed in their contacts.

User Discovery will return the following information:

* **User Record ID** – The unique ID of a user in the Public Database.
* **First and Last Name –** As stored in the Public Database.

This information will only be returned for users that have opted-into Discovery.

The following code will discover information about the user currently logged into iCloud on the device:

public CKDiscoveredUserInfo UserInfo { get; set; }  
...  
  
// Get the user's metadata  
CKContainer.DefaultContainer.DiscoverUserInfo(UserID, (info,e) => {  
    // Was there an error?  
    if (e!=null) {  
        Console.WriteLine("Error: {0}",e.LocalizedDescription);  
    } else {  
        // Save the user info  
        UserInfo = info;  
    }  
});

Use the following code to query all users in the Contact Book:

// Ask CloudKit for all of the user's friends information  
CKContainer.DefaultContainer.DiscoverAllContactUserInfos((info,er) => {  
    // Was there an error  
    if (er!=null) {  
        Console.WriteLine("Error: {0}",er.LocalizedDescription);  
    } else {  
        // Process all returned records  
        for(int i=0;i<info.Count();++i) {  
            // Grab a user  
            var userInfo = info[i];  
        }  
    }  
});

In this section we have covered the four major areas of access to a user’s account that CloudKit can provide to an application. From getting the user’s Identity and Metadata, to the Privacy policies that are built into CloudKit and finally, the ability to Discover other users of the application.

## When to Use CloudKit

As we have seen in this article, CloudKit provides an easy way for an application to store and retrieve information from the iCloud servers. That being said, CloudKit does not obsolete or deprecate any of the existing tools or frameworks.

### Use Cases

The following use cases should help the developer decide when to use a specific iCloud framework or technology:

* **iCloud Key-Value Store** – Asynchronously keeps small amount of data up-to-date and is great for working with application preferences. However, it is constrained for a very small amount of information.
* **iCloud Drive –** Built on top of the existing iCloud Documents APIs and provides a simple API to sync unstructured data from the file system. It provides a full offline cache on Mac OS X and is great for document centric applications.
* **iCloud Core Data –** Allows data to be replicated between all of the user’s devices. The data is single-user and great for keeping private, structured data in sync.
* **CloudKit –** Provides public data both structure and bulk and is capable of handling both large dataset and large unstructured files. Its tied to the user’s iCloud account and provides client directed data transfer.

Keeping these use cases in mind, the developer should pick the correct iCloud technology to provide both the current required application functionality and provide good scalability for future growth.

## Summary

This article has covered a quick introduction to the CloudKit API. It has shown how to provision and configure a Xamarin iOS application to use CloudKit. It has covered the features of the CloudKit Convenience API. It has show how to design a CloudKit enabled application for scalability using Queries and Subscriptions. And finally it has shown the User Account information that is exposed to an application by CloudKit.