



Penthera 301: iOS SDK Beyond the Basics

iOS SDK 4.0.2

August 13, 2020

Advanced development documentation for Penthera iOS SDK 4.0.2. Covers intermediate and advanced topics, as well as extended SDK features.

Copyright © 2020 Penthera Partners

CONFIDENTIAL - Licensed Use Only

Table of Contents

| | |
|--|----|
| Intermediate Topics | 3 |
| Configure Device-level Download Behaviors: iOS | 3 |
| Download Ancillary Files: iOS | 5 |
| Enable/Disable Other Devices: iOS | 5 |
| Push Notifications: iOS | 6 |
| Availability Windows for Assets: iOS | 8 |
| Advanced Topics | 11 |
| Upgrading the SDK: iOS | 11 |
| Observing Manifest Parsing: iOS | 11 |
| Digital Rights Management (DRM): iOS | 12 |
| Apple FairPlay DRM | 12 |
| Configure FairPlay DRM | 12 |
| Utilize FairPlay Assets | 13 |
| Extended FairPlay Server Requirements | 14 |
| Google Widevine DRM: iOS | 14 |
| Extended Widevine Server Requirements | 15 |
| DRM Extensions: iOS | 15 |
| Enhancing DRM with the VirtuosoLicenseManager | 15 |
| Extending DRM with a VirtuosoLicenseManagerDelegate | 15 |
| Extending DRM with a VirtuosoAVAssetResourceLoaderDelegate | 17 |
| Using BuyDRM or CastLabs DRM | 19 |
| Extended Features | 20 |
| FastPlay: iOS | 20 |
| Subscriptions: iOS | 21 |
| VirtuosoSubscriptionManager | 21 |
| Subscribe to a Feed | 22 |
| Receiving Updates from the Subscription Manager | 22 |
| Register a VirtuosoSubscriptionDataSource | 22 |
| Configure Subscription Rules | 24 |
| Optional: Force a Sync | 24 |
| What Next? | 26 |

Intermediate Topics

Configure Device-level Download Behaviors: iOS

The SDK obeys several device-level behavioral settings. You may access and configure these settings through the `VirtuosoSettings` instance:

maxDownloadedAssetsOnDevice: Maximum number of assets that may be downloaded on this device at any given time. Any assets added to the queue beyond this limit will be blocked until existing assets are deleted. (default=100)

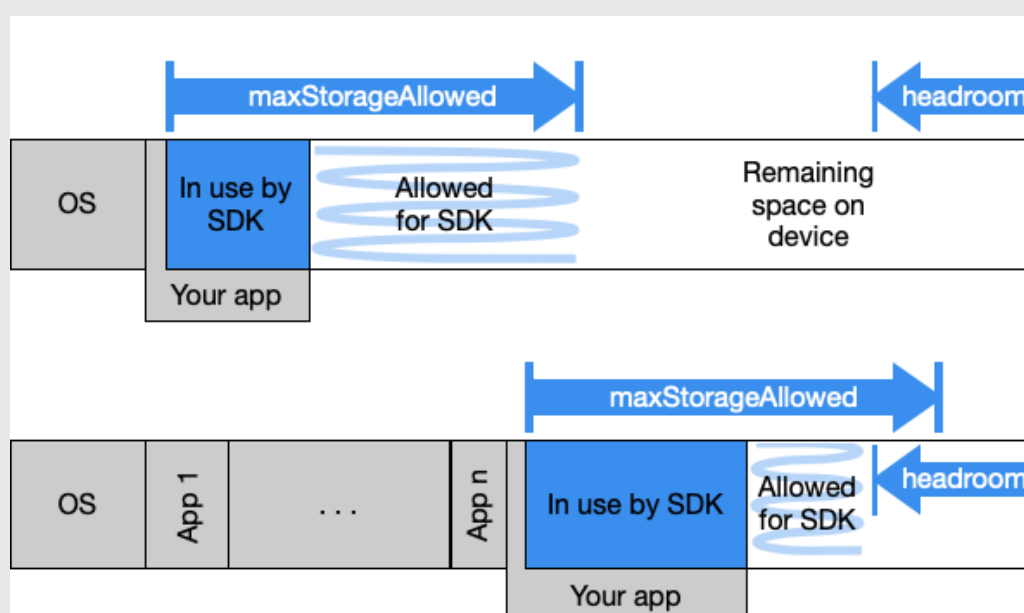
maxStorageAllowed: Maximum disk space that the SDK will use on the device. While downloading in the foreground, the SDK will cease download if it reaches `maxStorageAllowed`, even if that means stopping with a partially-completed file. While downloading in the background, the SDK will not initiate a download unless, after downloading that item, the total device storage used by the SDK will still be under `maxStorageAllowed`. (in MB; default=LONG_LONG_MAX)

headroom: Storage capacity that the SDK will leave available on the device. While downloading with the app in the foreground, the SDK will cease download if the headroom space is all that remains on the device, even if that means stopping with a partially downloaded item. When downloading in the background, the SDK will not initiate a download unless, after downloading that item, at least headroom space will still be free on the device. (in MB; default:1024)



NOTE

This diagram illustrates the relationship between `maxStorageAllowed` and `headroom` parameters. The SDK will always preserve a minimum free space on disk ("headroom"). In the first scenario the device has ample total available storage, so the SDK storage would be capped only by its `maxStorageAllowed`. In the second scenario, the device is beginning to fill up with other apps and is running low on available storage, so the SDK `headroom` requirement is restricting storage usage by the SDK. In the second scenario the SDK is no longer allowed to reach its `maxStorageAllowed` due to a lack of total available storage on the device.



**NOTE**

Once you have received a blocked state from the observer, you can call `[engine diskStatusOK]` which returns a boolean, indicating whether the SDK is blocked on disk/storage restrictions or something else.

Alternatively, you could also check `[VirtuosoAsset allowableStorageRemaining]`, which in that state will return 0.

Note that having insufficient space on the device is not considered to be a permanent fatal error, and so the SDK will continue its attempt to complete downloads once space has become available. For a headroom issue, downloads may continue after the deletion of anything on the device which provides more headroom. For a max storage issue, downloads may continue after removal of previously-downloaded items.

**NOTE**

Without knowing the file sizes ahead of time, or hosting content with a constant bitrate it can be extremely difficult to estimate file sizes. The estimated file size for videos is based on the selected bitrate to download and the duration of the asset. The SDK uses that information to calculate the initial expected size. Once download starts, we take the existing downloaded segments and create an "average segment size" for each segment type (audio, video, etc.) and then calculate an estimate based on the total number of segments multiplied by the average downloaded segment size. Naturally, with this approach, the estimated size will become more accurate over time.

downloadOverCellular: (default=NO) Whether the SDK is permitted to download over a cellular network. This value is a permission, and does not guarantee that downloads will continue on a cellular network. It only indicates the the SDK may download over cellular, should internal business rules allow it. If this value is NO, downloads will never happen over a cellular connection.

destinationPath: An additional relative path component added to the enclosing app's Documents directory. The SDK will store all downloaded files here. By default, the SDK stores all downloads in the Documents directory itself, under appropriate sub-directories.

**TIP**

The iOS SDK, with `permittedSegmentDownloadErrors` on `VirtuosoSettings`, allows to configure the SDK to tolerate a non-zero number of segment failures before marking the asset as failed. See `VirtuosoSettings.h` for this and other related settings.

**TIP**

The Penthera iOS SDK has a read-only class-level property on `VirtuosoAsset` to read the `downloadRetryLimit`, which is set to 2.

Download Ancillary Files: iOS

The Penthera SDK provides the ability to download arbitrary additional files, which are managed by the SDK within the same lifecycle of the related asset. When the asset is later removed from the user device, the SDK will also automatically clean up the related ancillary files.

You may indicate ancillary files when you are configuring to construct your asset:

```
VirtuosoAncillaryFile *poster = [[VirtuosoAncillaryFile alloc]
initWithDownloadURL:posterURLString andTag:@"poster"];

VirtuosoAncillaryFile *thumbnail = [[VirtuosoAncillaryFile alloc]
initWithDownloadURL:thumbnailURLString andTag:@"thumbnail"];

VirtuosoAssetConfig *myConfig = [[VirtuosoAssetConfig alloc]
initWithURL:myAssetURLString assetID:myAssetID description:myDescription
type:kVDE_AssetTypeHLS];

[myConfig setAncillaries:@[poster, thumbnail]];

VirtuosoAsset *myAsset = [VirtuosoAsset assetWithAssetConfig:myConfig];
//now when the asset is processed on the download queue, the ancillaries
will also be retrieved
```

Once the SDK has completed download of the asset, the ancillary files can be accessed from the asset instance. You could access the ancillaries all at once with `findAllAncillaries`, but notice in the example that tags were used when declaring the ancillary files. These tags can be used to easily retrieve a specific ancillary file, such as poster images, ancillaries in other languages, or for any other purpose.

```
NSArray<VirtuosoAncillaryFile*> myPosters = [myAsset
findCompletedAncillariesWithTag:@"poster"];
```

Enable/Disable Other Devices: iOS

In addition to startup and shutdown of the Virtuoso engine on a given device, you also have the option of enabling and disabling the download capability on a device. For example, if you use the Penthera Cloud setting to limit the number of download-enabled devices a user is allowed on their account, you may also want to provide the ability for your users to choose which of their devices is enabled for download at any given time. To support this, the SDK can change the enable/disable setting for the local device, or for any other device associated with the user's account. The Penthera Cloud manages which devices are download-enabled, and enforces the global "max download-enabled devices per user" policy.

The example below shows how to retrieve the listing of User devices and change the download setting on one of them. Notice that because the Penthera Cloud is involved in enforcing your business rules, this code for enabling and disabling devices behaves differently if the local device is offline or not authenticated when the request is made.

```
NSString *anExternalIdToMatch = @"AN_EXTERNAL_ID"; //device we wish to
change

NSArray *userDevices = [[VirtuosoDownloadEngine instance] devices];

... // find the device with matching external ID

[matchingDevice updateDownloadEnabled:FALSE onComplete:^(Boolean success,
NSError *error){
    //handle success or failure
}];
```

**TIP**

If a request to enable or disable a remote device is made while the remote device is offline, the state change is still recorded in the Penthera Cloud and will propagate to the remote device when it syncs with the Penthera Cloud. In general the impact on a remote device may experience various delays if the remote device is offline, if syncing is delayed for any reason, or if push notifications are delayed. As long as the device is eventually brought online and the SDK is running, it will eventually receive the state change.

**DEVICE STARTUP & SYNCING MAY IMPACT BILLING**

Startup of the Virtuoso engine from a client device, and the periodic sync the engine performs with the Penthera Cloud servers, are what identify a device as being "active" in the Penthera ecosystem. The number of active devices in a given month impacts Penthera resource utilization, and is the typical mechanism from which Penthera derives its client billing.

The typical best practice for a client application is to leave users in the active state who rely on download and other SDK features. For such users you would startup the Virtuoso engine soon after they launch or log in to your app, and you would leave the engine running so it can manage background downloads and other features automatically.

If you limit download and other SDK features to a subset of your users, such as a premium user account, you will normally only startup the Virtuoso engine for those premium users (and not for users who are unable to use Virtuoso SDK features).

Push Notifications: iOS

While not required, we do recommend integration of push notification messaging as it can produce the most timely execution of some Penthera features. Note that push notifications are not used by Penthera to pop up user messages in your mobile app UI; the SDK uses push notifications to receive internal messages from the Penthera Cloud.

For iOS devices, the Penthera Cloud can use Apple Push Notifications (APN) to send messages to the SDK which trigger certain actions (e.g., download and delete). What follows explains the basic steps for enabling APN with the Penthera SDK. For broader coverage of how APN works, see Apple's developer documentation.

Provide APN config to the Penthera Cloud

Log in to the Penthera Cloud admin console and provide your APNS configuration information.

Register device push tokens with SDK

To send a push message to a specific device, the Penthera Cloud (or "backplane") needs to know the push token from your app running on that device. The Penthera SDK will inject code into your AppDelegate chain to feed push tokens to the SDK, so the SDK can upload them to the Penthera Cloud. You do not need to write additional code for that process.

Handling Received Push Notices

From time to time, a server (the Penthera Cloud or possibly another server associated with your app) may send your app an APN message. Upon arrival the APN message is delivered to well-known meth-

ods in your UIApplicationDelegate class, otherwise known as the AppDelegate, as according to Apple documentation. A typical implementation of those AppDelegate methods will dispatch the message to whatever app code should handle it. The Penthera SDK will automatically inject (aka, "swizzle") the necessary code into your AppDelegate chain so that you do not need to write any custom code to route Penthera APN messages to the Penthera SDK. If your ecosystem is sending additional APN messages to your app which are not sent by the Penthera Cloud, these will not be handled by the Penthera SDK, and will continue to route to the methods in your AppDelegate implementation so your code can handle them as expected.

**IMPORTANT**

If you use push notifications within your app for other purposes, to ensure your push notifications work well together with the push notification handling performed automatically by the Penthera SDK, make sure you implement your push notification handling with Apple's recommended `didReceiveRemoteNotification:fetchCompletionHandler:` approach.

**IMPORTANT**

The automatic injection of APN handling code by the Penthera SDK is new as of SDK 4.0. If you are upgrading from versions of the Penthera SDK prior to 4.0, and you were already handling Penthera APN messages, see [Upgrading the SDK: iOS \[11\]](#) for info about what you should remove from your existing code during your upgrade.



PERFORMANCE WITH PUSH NOTIFICATIONS

For performance reasons, Penthera batches up "push notice jobs" into groups, and the job processor fires them at intervals. The default interval is every 5 minutes. When the push notices go out from the server, the platform push service accepts them and then they may be delivered, or not.

For instance, some push messaging providers reserve the right to delay push notices until "convenient" times for the device, which could be a device unlock or when the device starts up its wifi / cellular radio. So there can be a delay between the push being sent to the push notification provider and the actual push notice getting to the device. In addition, on some systems pushes are delivered quickly when the user is frequently using your app, but are delivered "less quickly" (the timing of which is imprecise) when a user less frequently uses your app.

Push notice delivery is also not guaranteed. Because push messaging is not a guaranteed service, the commands that Penthera sends in the push notice are also sent down to the device during application syncs.

If the device misses a push for some reason, the next time the SDK syncs with the backplane server, it will pickup whatever commands it missed and process them at that time.

The Penthera iOS SDK limits sync frequency to no more than one sync every 15 minutes. If a sync has not already happened within that time window, a sync will be attempted if the app starts, if the app connects to a network, if the SDK receives a push notification, or if a background wake occurs during background downloading.

Apple's push notification system is one which may delay push notifications. We recommend implementing the "background fetch" feature on iOS to allow for syncs to happen occasionally when the app is backgrounded, although the timing of those is not guaranteed by Apple.

Availability Windows for Assets: iOS

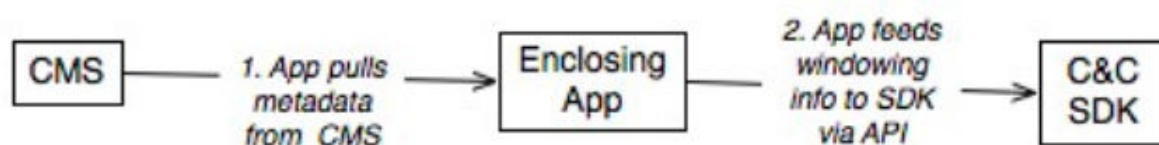
The 'Availability Window' governs when the video is actually available for playback by the user. The SDK enforces several windowing parameters on each video:

Table 1. Parameters for Availability Windows

| Windowing Parameter | Description |
|-----------------------------|--|
| Publish Date | The SDK will download the video as soon as possible, but will not make the video available through any of its APIs until after this date. |
| Expiry Date | The SDK will automatically delete the video as soon as possible after this date. |
| Expiry After Download (EAD) | The duration a video is accessible after download has completed. As soon as possible after this time period has elapsed, the SDK will automatically delete this video. |
| Expiry After Play (EAP) | The duration a video is accessible after first play. As soon as possible after this time period has elapsed, the SDK will delete this video. To enforce this rule, the SDK has to know when the video is played, so be sure to register a play-start event when the video is played. |

The Penthera Cloud stores a global default value for EAP and EAD. You may set these values through the Penthera Cloud web UI. The Cloud transmits these default values to all SDK instances. Defaults provided by the Penthera Cloud can be overridden by values provided to the SDK by your app.

Typically, a Content Management System (CMS) stores the windowing information for an item, and communicates it through a web API to the enclosing app. The app then feeds this windowing information to the SDK:



Overriding defaults with asset-specific values is performed when you create the `VirtuosoAsset` object. You can also modify some of the values later, via the appropriate properties.

```

VirtuosoAssetConfig *config = [[VirtuosoAssetConfig alloc]
    initWithURL:@"http://path/to/your/asset.mp4"
    assetID:@"your_unique_asset_id"
    description:@"Test Video"
    type:kVDE_AssetTypeNonSegmented];

// Available now
config.publishDate = [NSDate date];

// Expires in 7d
config.expiryDate = [NSDate dateWithTimeIntervalSinceNow:604800];

// Expires 24h after download
config.expiryAfterDownload = 86400;

// Expires 12h after play
config.expiryAfterPlay = 43200;

[VirtuosoAsset assetWithConfig:config];
  
```

Each of the SDK's content lookup methods (e.g. `assetsWithAvailabilityFilter:`) contains an `availabilityFilter` parameter. Set this parameter to YES to filter for only items still valid given windowing constraints. Set the parameter to NO to list all items, regardless of windowing.

The SDK will delete a video as soon as possible after the video expires. Also, when a caller tries to access an expired item via the API, any downloaded files associated with that item will be auto-deleted from disk, calls to play the item via the `VirtuosoClientHTTPServer` will fail, and any attempts to access the local file URLs will return nil.



NOTE

The expiry date can be changed remotely after an asset has been downloaded to disk, but the `expiryAfterDownload` and `expiryAfterPlay` values cannot.

**NOTE**

The Penthera SDK has an internal secure clock implementation that is entirely independent of the local clock time. The secure clock time is based on a combination of NTP calls and Penthera Backplane time stamping for "ground truth time" combined with tracked differentials of the system "up-time clock". All expiry calculations are done against this internal secure clock, and expiry is independent of the local clock settings. On both iOS and Android, you can check what the SDK secure clock is set to by using the public interfaces.

Advanced Topics

Upgrading the SDK: iOS

Unless the release notes indicate otherwise, SDK versions are back-compatible with older versions. If you had previously deployed a version of your app using an earlier version of the SDK, then the old data store will automatically be upgraded to the new data store. There are a few important considerations for handling of this process:

Asynchronous Update

Depending on how many assets the user has created in the app, the upgrade process may take a fair amount of time. To ensure that you can startup the engine without worrying about thread considerations, the SDK upgrades the data store in a background process. This allows your app to appear fully functional to the user, but it does mean that while the upgrade process is proceeding, previously-downloaded assets may be unavailable.

The `VirtuosoDownloadEngine` will issue two notifications via `NSNotificationCenter` to indicate when the upgrade process is starting and when the upgrade process has finished:

```
extern NSString* kDownloadEngineDidBeginDataStoreUpgradeNotification;  
extern NSString* kDownloadEngineDidFinishDataStoreUpgradeNotification;
```

You can use these notifications as a trigger in your user interface to indicate to the user that previously downloaded assets are temporarily unavailable.

Download Continuity

All asset metadata will be fully available as soon as the data upgrade process completes. Any assets that had already been downloaded will be immediately available for offline playback. Any assets that had not started downloading yet will remain enqueued and will download normally.

Observing Manifest Parsing: iOS

It is possible to be notified when manifest parsing completes for a newly instantiated `VirtuosoAsset`. When constructing the asset, use the `assetWithConfig:onReadyForDownload:onParsingComplete:` method of `VirtuosoAsset`. Pass the appropriate completion block into the `onParsingComplete` parameter.

Note that the engine is optimized such that for manifest-driven asset it may begin download before the manifest is completely parsed. Thus you may receive a callback to `onReadyForDownload` completion blocks before `onParsingComplete` completion blocks.

Digital Rights Management (DRM): iOS

The Penthera SDK provides out-of-box support for Apple FairPlay (iOS 10+) and Google Widevine (iOS 8+) DRM, together with a mechanism to easily provide custom support for specialized DRM server requirements as necessary. With a few easy steps, the SDK will not only manage retrieval of DRM for you, it will also manage the lifecycle of keys for various assets, including refreshing the keys at appropriate intervals to ensure the best likelihood that keys are valid for offline playback when your user desires it. At playback, the SDK runs a local proxy server which provides the DRM keys and the video assets, so that SDK-managed keys are provided to your player of choice whenever necessary.

If you are using Penthera's built-in FairPlay or Widevine support you may not need additional code to fetch or persist DRM licenses. Providing little more than the URL of your DRM server, the SDK can automatically request a cacheable license when the download starts and will attempt to renew it when the download finishes. The SDK will also automatically delete DRM licenses when the asset is deleted.

A DRM server may also require special formatting or tokens in the request and/or response, so the SDK allows your implementation to provide increasing levels of custom configuration to meet common needs. For the most extreme cases you can create a deeply custom implementation which performs all the server interaction within the DRM and asset management framework provided by the SDK.

While not preferred, it is even possible in extreme cases to handle DRM license retrieval in your own code, and provide the licenses to the SDK to manage alongside the assets. For many DRM systems other than FairPlay and Widevine, no custom integration is necessary.



CAUTION

If you must handle DRM license retrieval with entirely custom code, we recommend that you fetch the DRM license before you tell the Penthera SDK to queue an asset for download. That way, you can be sure the DRM license is on the device when the video download finishes. If you wait until the video download finishes before fetching the license, you may run into trouble. If the asset download finishes while the app is in the background, your app may not have an opportunity to download the license before the device goes offline and the app can't fetch the license.

Apple FairPlay DRM

The Penthera SDK can support FairPlay DRM with little more than the URL to retrieve the client FairPlay certificate, the URL of the FairPlay DRM server, and the URL of the manifest for FairPlay-protected assets. Once provided with that information, the Penthera SDK supports offline playback of FairPlay-protected videos, provided that your FairPlay license server is configured to grant the appropriate permissions in the license.

Configure FairPlay DRM

Configuration of FairPlay with the SDK requires only to configure the client certificate URL and DRM server URL.

Configure Client Certificate URL

Provide the URL where the SDK can download the client's FairPlay application certificate during application startup:

```
[VirtuosoLicenseManager
    downloadClientAppCertificateFromURL:<certificate_url>
    forDRM:kVLM_FairPlay];
```

Configure DRM Server URL

Provide the URL to your FairPlay License Server during application startup

```
[VirtuosoLicenseManager
    setLicenseServerURL:<license_server_url>
    forDRM:kVLM_FairPlay];
```

Utilize FairPlay Assets

To utilize FairPlay assets, you need to identify them as such when asking the SDK to download them, and then play them back from the URL provided by the SDK.

Identify FairPlay Assets

When you start a new download by constructing a `VirtuosoAsset`, pass `kVDE_AssetProtection-TypeFairPlay` as the `protectionType` parameter to identify that asset as using FairPlay DRM. In addition to fetching the video assets, the SDK will send the raw encrypted SPC request and parse the raw encrypted response. The queue for download procedure is discussed further in [Queue an Asset for Download: iOS](#).

Since it is typical for FairPlay metadata to be encrypted within the keys (and thus unreadable by the SDK), you may supply our SDK with the key renewal date using the `VirtuosoLicenseManagerDelegate` defined in `VirtuosoLicenseManager.h`. Otherwise, if the device is online at the time of playback, Apple will identify if a FairPlay license has expired and will make a renewal license call when the user attempts to play it. See [DRM Extensions: iOS \[15\]](#) for more details on the `VirtuosoLicenseManagerDelegate`.

Playback FairPlay Assets

If you are using one of player classes provided by the Penthera SDK, there is nothing special necessary for playback. Play assets just as indicated in [Play Downloaded Content: iOS](#). The SDK will provide the DRM key to the player.

Playback of FairPlay Assets in a Custom Player

If you require a custom player for your application you may use methods directly on the `VirtuosoClientHTTPServer` class to perform the steps that the player classes we provide in the SDK would normally perform for you. These include instantiating a local playback proxy (the `VirtuosoClientHTTPServer` instance) for a given asset, obtaining the local proxy URL the player will use for playing the local asset, obtaining the local proxy URLs the player will use to obtain the DRM certificate and DRM key, and finally calling shutdown on the local proxy when your custom player is done. Below is specific guidance on how to playback FairPlay assets with a custom player.

1. Instantiate and hold a reference to an instance of our `VirtuosoClientHTTPServer`. The appropriate location for this is likely within a `ViewController` where you embed a player or from which you will open a child view controller containing the player. Hold a reference to our server instance until the user is done with playback.

```
//Have access to the VirtuosoAsset you wish to play
VirtuosoAsset *myAsset = ...

//Instantiate a local proxy server instance with your desired asset
VirtuosoClientHTTPServer *server = [[VirtuosoClientHTTPServer alloc]
    initWithAsset:myAsset];
```

2. Retrieve the playback URL and DRM URLs from the local proxy server instance you just created. Use these to configure your custom player for playback, which enables the player to have access to both the asset and DRM files being managed by the Penthera SDK.

```
if (server) {
    //The now-running server instance provides local proxy URLs for
    playback and DRM

    NSString *licenseURL = [server fairPlayLicenseServerURL];
    NSString *certificateURL = [server
    fairPlayCertificateDataURLForSubType:nil];
    //use these license and certificate URLs to configure DRM support in
    your custom player

    NSString *playbackURL = [server playbackURL];
    //pass this playbackURL to your player

    //playback now happens in your player
} else {
    //returned nil because of some error... error would have been sent to
    NotificationCenter
}
```

3. Later, after playback ends (e.g., player view closed), you want to shut down the instance of `VirtuosoClientHTTPServer`. At this point you can release the server reference. You must hold a reference to our server instance throughout playback, or our local proxy server will shut down prematurely.

```
[server shutdown];
```

Extended FairPlay Server Requirements

Frequently, DRM servers use specially-formatted requests and responses, such as specialized JSON-structured payloads, base-64 encoded elements, and custom parameters in the request string or headers. For those cases, read our section on [DRM Extensions: iOS \[15\]](#)

Google Widevine DRM: iOS

The Penthera SDK automatically supports offline playback of Widevine-protected videos, provided that your Widevine license server grants appropriate permissions for the license. Enabling Widevine support requires only the following:

1. Include the Google Widevine framework in your project. Location `widevine_cdm_sdk_release.framework` in the ThirdParty directory of your Penthera SDK distribution. Drag and drop the framework into your XCode project. Under the General section in “Embedded Binaries”, add the `widevine_cdm_sdk_release.framework` to the list.
2. Configure your Widevine License Server during application startup:

```
[VirtuosoLicenseManager
    setLicenseServerURL:<license_server_url>
    forDRM:kVLM_Widevine];
```

3. When you start a new download, pass `kVDE_AssetProtectionTypeWidevine` as the `protectionType` parameter.

If you are using one of Penthera SDK built-in player classes, that’s all you need to do.

Extended Widevine Server Requirements

Frequently, DRM servers use specially-formatted requests and responses, such as specialized JSON-structured payloads, base-64 encoded elements, and custom parameters in the request string or headers. For those cases, read our section on [DRM Extensions: iOS \[15\]](#)

DRM Extensions: iOS

You have likely seen that the `VirtuosoLicenseManager` is involved in DRM configuration, where in the simplest cases you provide the URLs for your DRM client certificate and your license server. This class is also your starting point for further extending and customizing DRM support within the Penthera SDK.

If your customization needs extend beyond what is capable with the `VirtuosoLicenseManager`, you may need to extend the SDK license functions using a `VirtuosoLicenseManagerDelegate`, or much less likely using the `VirtuosoAVAssetResourceLoaderDelegate`. Let us explore when you might need each of these mechanisms.

Enhancing DRM with the `VirtuosoLicenseManager`

First of all, note that various additional DRM features are possible using other methods on the `VirtuosoLicenseManager` API. With that API you can do things like:

- configure the SDK to simultaneously support various different DRM types, by providing different cert and server URLs for various DRM types and subtypes,
- provide a client DRM cert manually, rather than having the SDK fetch it from a remote URL, in the event you need to access or store your client cert in a custom manner,
- manually induce the download of an offline license key for an asset,
- manually induce the refresh of a license key,
- manually delete a license key from SDK management,
- provide a license key to the SDK for management which you acquired in a custom manner, and
- retrieve the managed license key for an asset, in case you need to manually provide it to a player in some custom manner rather than letting the SDK provide it during playback.

All of those functions can be found in methods of the `VirtuosoLicenseManager` class as detailed in the `VirtuosoLicenseManager.h` header.

Extending DRM with a `VirtuosoLicenseManagerDelegate`

The `VirtuosoLicenseManagerDelegate` is the most commonly used DRM extension point. This delegate allows you to set a specific renewal date on an asset's license key, append a custom suffix to the URL used to retrieve a license key, or add custom query parameters and request headers to the license key request. In each case you have access to the asset, so your customizations can either be generic to all requests (such as with a custom authentication token) or specific to the asset (such as providing an asset ID).

The simplest way to achieve this is by implementing the `lookupLicenseForAsset:` method, which returns a `VirtuosoLicenseConfiguration` instance containing your customizations. Implement the delegate method, provide any necessary customizations, leave unused customizations nil, and register your delegate with the `VirtuosoLicenseManager`:

```
[VirtuosoLicenseManager setDelegate:self];

...

#pragma mark - VirtuosoLicenseManagerDelegate

-(VirtuosoLicenseConfiguration* _Nullable)lookupLicenseForAsset:
(VirtuosoAsset* _Nonnull)asset
{
    VirtuosoLicenseConfiguration *config =
        [[VirtuosoLicenseConfiguration alloc]
         initWithSuffix:nil
         renewal:nil
         parameters:@{@"paramToken": @"xyzzzy12345"}
         headers:@{
             @"Content-Type": @"application/json; charset=UTF-8",
             @"headerToken": @"token54321"
         }
        ];
    return config;
}
```

You may alternatively implement the older form method on the same delegate, which is called during license requests, and which provides values by implementing dereferenced parameters (leaving nil any which are not needed):


```

[VirtuosoLicenseManager setDelegate:self];

...

#pragma mark - VirtuosoLicenseManagerDelegate

- (void)lookupLicenseURLSuffix:(NSString * _Nonnull __autoreleasing *
_Nullable)urlSuffix
    andParameters:(NSDictionary * _Nonnull __autoreleasing *
_Nullable)customParameters
    additionalHeaders:(NSDictionary * _Nonnull __autoreleasing *
_Nullable)customHeaders
    renewalDate:(NSDate * _Nonnull __autoreleasing * _Nullable)renewalDate
    forAsset:(VirtuosoAsset * _Nonnull)asset
{
    // To customize elements of the DRM request, implement this method and
    provide
    // any required modifications to the licensing properties according to
    the provided asset.
    // For example:
    // *urlSuffix = @"asset-unique-suffix-for-drm";
    // Gets appended onto the end of the DRM license URL
    // *customParameters = @{@"key":@"value"};
    // key/value pairs that get added to the URL query string
    // *customHeaders = @{@"key":@"value"};
    // key/value pairs which become request headers
    // *renewalDate = [NSDate dateWithTimeIntervalSinceNow:48_HOURS];
    // Expected DRM renewal timestamp

    *customParameters = @{
        @"form": @"json",
        @"schema": @"1.0",
        @"account": @"http://someserver.com/data/Account/8675309",
        @"token": @"xyzzyl2345"
    };
    *customHeaders = @{
        @"Content-Type": @"application/json; charset=UTF-8",
        @"myToken": @"token54321"
    };
}

```

Extending DRM with a VirtuosoAVAssetResourceLoaderDelegate

Sometimes your DRM license server may require special formatting of the request body, or extraction of the key from within a custom response body. For example, the default request body contents of a FairPlay server request are the raw bits of the SPC block, and the default response body contents are the raw bits of the CKC. If your FairPlay server requires to submit the SPC as part of a custom JSON request body, and/or returns the CKC embedded somewhere in a JSON response body, and/or requires base64 encoding of the request or response, you may use a `VirtuosoLicenseProcessingDelegate` to achieve these customizations.

The Penthera SDK provides and uses a default `AVAssetResourceLoaderDelegate`, in the form of our `VirtuosoDefaultAVAssetResourceLoaderDelegate`. If you need to implement our `VirtuosoLicenseProcessingDelegate` methods, you then register your `VirtuosoLicenseProcessingDelegate` delegate with our `VirtuosoDefaultAVAssetResourceLoaderDelegate` implementation, and the rest is handled for you.

To access or modify the request and/or response bodies, implement the request and response customization methods of the `VirtuosoLicenseProcessingDelegate`, and register your delegate instance with the `VirtuosoDefaultAVAssetResourceLoaderDelegate`:

```
[VirtuosoDefaultAVAssetResourceLoaderDelegate
setLicenseProcessingDelegate:self];

...

#pragma mark - VirtuosoLicenseProcessingDelegate

/* Deconstruct any custom DRM CKC response to extract and return the binary
license data. Returning nil causes the complete original binary CKC
response to be used as-is. */
- (NSData * _Nullable)extractCKCForAsset:(VirtuosoAsset * _Nonnull)asset
    inLicenseResponse:(NSData * _Nonnull)ckcData
{
    //handle the response body here, returning nil or the CKC bits
}

/* Construct any custom DRM SPC request from the asset and the default SPC
(which is a pure binary Fairplay SPC). The NSData returned from this method
is POSTed to the license server as the request body. Returning nil causes
the original binary SPC to be used as-is. */
- (NSData * _Nullable)prepareSPCForAsset:(VirtuosoAsset * _Nonnull)asset
    inLicenseRequest:(NSData * _Nonnull)spc
{
    //handle the request body here, returning nil or the customized request
    body
}
```

The `VirtuosoLicenseProcessingDelegate` also provides a custom method for determining the CID. Normally, the asset CID is extracted from the FairPlay license URL, which takes the form: `skd://<contentID>`. Some license servers put the required content ID elsewhere, so if necessary, implement the following method and return a CID value other than nil:

```
[VirtuosoDefaultAVAssetResourceLoaderDelegate
setLicenseProcessingDelegate:self];

...

#pragma mark - VirtuosoLicenseProcessingDelegate

/* Extract and return the desired content ID from either the provided asset
or from the default Fairplay request URL. */
- (NSString * _Nullable)extractCIDForAsset:(VirtuosoAsset * _Nonnull)asset
    fromFairPlayRequest:(NSURL * _Nonnull)fpRequest
{
    //return nil if the URL contains the CID in standard sky://<contentID>
    form, or
    //return another value, for example, if the asset ID is the content ID
    return asset.assetID;
}
```

If none of the above seems to resolve your custom DRM circumstances, contact Penthera Support. It is possible to leverage your own `AVAssetResourceLoaderDelegate` to serve as a child delegate to, or even as a replacement for, our default implementation `VirtuosoDefaultAVAssetResourceLoa-`

derDelegate. This is a more complex process, which is rarely required, but if it seems necessary please contact us first. We can help confirm it is actually necessary, and can assist you with the implementation.

Using BuyDRM or CastLabs DRM

As part of the SDK we also ship with support for two highly customized DRM adapters, one each for use with BuyDRM and CastLabs DRM servers. If you are using either of those DRM systems, you will replace our default `VirtuosoDefaultAVAssetResourceLoaderDelegate` with either our BuyDRM or CastLabs `AVAssetResourceLoaderDelegate`, which also we ship in the SDK. These are injected into the SDK similar to how you would use an entirely custom `AVAssetResourceLoaderDelegate`. For example:

1) Configure your FairPlay cert and CastLabs URLs. This is usually done in your `appDidFinishLaunching` method.

```
[VirtuosoLicenseManager downloadClientAppCertificateFromURL:@"<certificate URL>" forDRM:kVLM_FairPlay];

[VirtuosoLicenseManager setLicenseServerURL:@"https://myCastlabsServer.com/license-server-fairplay/" forDRM:kVLM_FairPlay];
```

2) Enable our built-in CastLabs integration. This can also be done right after the above setup:

```
// CastLabs Configuration
[VirtuosoLicenseManager registerAVAssetResourceLoaderDelegate:
[CastLabsAVAssetResourceLoaderDelegate class]];
```

3) Implement the `drmConfigForAsset:` method of the `VirtuosoDrmConfigDelegate`, and register that implementation with the custom `AVAssetResourceLoaderDelegate`. In our example, the `VirtuosoDrmConfigDelegate` method is implemented in the app delegate, but you can put this method into any logical component in your architecture as you see fit:

```
[CastLabsAVAssetResourceLoaderDelegate setDrmConfigDelegate:self];

...

#pragma mark - VirtuosoDrmConfigDelegate

- (VirtuosoDrmConfig*)drmConfigForAsset:(VirtuosoAsset*)asset
{
    // CastLabs Configured
    CastLabsDrmConfig *config = [CastLabsDrmConfig new];
    config.clUserID = @"<user id>";
    config.clSessionID = @"<session id>";
    config.clCustomerName = @"<customer name>";
    return config;
}
```

Extended Features

FastPlay: iOS

FastPlay is a feature which provides instantaneous playback of **streaming** videos. FastPlay eliminates the frustrating time your user would spend waiting for the player to buffer enough video to start playback. This provides a vastly improved experience when browsing your video catalog, and should also be applied to popular and featured videos which you highlight in your app UI.

As soon as it seems your user might be interested in playing a given streaming video, such as when that video is featured in your UI or when the user opens the detail page of a video, you should construct a FastPlay-enabled asset. You do not need to add FastPlay-enabled assets to your download queue. The SDK will immediately download and parse that manifest, and download the opening few seconds of the video. If the user then chooses to hit "play," you initiate playback of the FastPlay-enabled asset just as you would for an asset that had been added to the download queue. *The opening seconds of the video play instantly*, because they have already been downloaded to the local device, and when those opening seconds are exhausted the player automatically and seamlessly transitions into the buffered online stream. This allows your player to buffer a substantial amount of the video, with little or no wait to start playback.

If you have implemented even the most basic download features of the Penthera SDK, you can FastPlay-enable your application with very simple code. You construct the asset in the same manner as for an asset you wish to add to the download queue, except you set a flag to enable FastPlay, and do not add that asset to the download queue.

For example, simply enable FastPlay on your `VirtuosoAssetConfig`, which you use to create your `VirtuosoAsset`:

```
VirtuosoAssetConfig *newAssetConfigForFastPlay = [[VirtuosoAssetConfig
alloc] initWithURL:myAssetURLString assetID:myAssetID
description:myDescription type:kVDE_AssetTypeHLS];

//enable FastPlay
[newAssetConfigForFastPlay setFastPlayEnabled:YES];
//disable offline playback (optimizes readiness if not also needed for
offline playback)
[newAssetConfigForFastPlay setOfflinePlayEnabled:NO];
//set other config settings as necessary

VirtuosoAsset *myAsset = [[VirtuosoAsset alloc]
initWithConfig:newAssetConfigForFastPlay];
```

Hold the asset reference in your code, and the SDK will silently prepare it for FastPlay. When the user wants to start playback, check the asset to see if it is FastPlay-ready, and if so you initiate playback with the same code you would if the asset was from the download queue. See for [Play Downloaded Content: iOS](#) additional details.

**TIP**

For the fastest readiness with FastPlay, also disable offline playback on the `VirtuosoAssetConfig` when you create the asset. If the asset does not need to be downloaded for offline playback, this allows the engine to skip some steps and also optimizes bandwidth usage.

If the user has requested playback so soon that FastPlay is not yet ready, you simply fall back to playing the video direct from the online manifest as you would have without FastPlay.

```
if ([myAsset fastPlayReady]) {
    //initiate playback as you would for a downloaded asset
} else {
    NSString *onlineManifestURL = [myAsset assetURL];
    myAsset = nil;
    //now initiate playback with the online manifest URL
}
```

That is all that is required. The `fastPlayReady` flag on the asset is sufficient to know whether FastPlay preparations are complete when your user presses a button to start playback.

Subscriptions: iOS

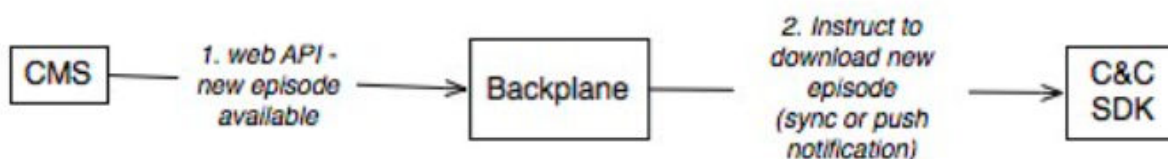
We've already described how you can enqueue a single video (flat file or fragmented) for download. In addition, the SDK can subscribe to a feed of assets, a.k.a. "episodes."

**IMPORTANT**

The subscriptions feature will be greatly simplified in an upcoming release of the SDK. Stay tuned for more details. The material as presented in this section will soon be deprecated.

The Backplane keeps track of which SDK instance is subscribed to which feeds. The Backplane also keeps track (by communicating with your CMS) of which episodes belong to which feeds.

As new episodes in a feed become known to the Backplane, the Backplane informs the SDK, through the normal SDK-Backplane sync and/or through a push notification. In response, the SDK automatically adds the new episode to its download queue.



VirtuosoSubscriptionManager

`VirtuosoSubscriptionManager` is the main class for handling subscription logic. It interfaces with the Backplane and maintains information on feeds and episodes in those feeds.

`VirtuosoSubscriptionManager` is a singleton (i.e. one and only one instance ever exists). Calling the instance method will initialize the object (if needed), link to the `VirtuosoDownloadEngine` configuration, and return a pointer to the valid instance:

```
VirtuosoSubscriptionManager* manager = [VirtuosoSubscriptionManager
instance];
```



NOTE

You need to start `VirtuosoDownloadEngine` before the above.

Subscribe to a Feed

Each feed is identified by an externally defined feed asset ID. To register for updates on that feed:

```
// Duplicate registrations do nothing
[[VirtuosoSubscriptionManager instance]
 registerForSubscription:@"customer_defined_feed_uuid"
 onComplete:^(Boolean success, NSError *error)
 {
     // Handle response
 }
];
```

Receiving Updates from the Subscription Manager

To get updates from the Manager (and thus keep your views refreshed), your code must register to receive the various `NSNotification` messages sent by the Manager. The Manager provides notices when status is updated, content is reset (downloaded data deleted), content is deferred (created but not enqueued), and when items are added (created and enqueued). You register for notifications in your own code as follows:

```
[[NSNotificationCenter defaultCenter]
 addObserverForName:kSubscriptionManagerAssetAddedNotification
 object:nil
 queue:[NSOperationQueue mainQueue]
 usingBlock:^(NSNotification *note)
 {
     // Get list of new VirtuosoAsset objects
     NSArray* newAssets = [note.userInfo
         objectForKey:kSubscriptionManagerNotificationVirtuosoAssetKey];
     // Handle new assets.
 }
];
```

You'll find the latest set and versions of these notifications in the `VirtuosoSubscriptionManager` header file. Registering for any of these notices will give you all the required data needed in order to update your UI.

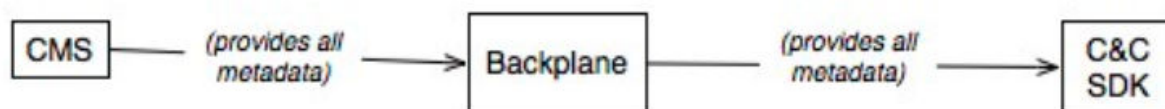
Register a VirtuosoSubscriptionDataSource

Let's say the SDK receives a push notification from the Backplane indicating a subscription update occurred. The SDK will then connect to the Backplane and retrieve a list of subscription updates. Depending on your Backplane configuration, that information may or may not contain all of the metadata required to download the new episodes. At this point, the SDK may need to request additional information

about an episode (e.g. the remote URL, which profile to use, expiry rules for the episode). The enclosing app may need a title, description, an image, and other metadata for the episode. A complete list of the metadata which will be processed by the SDK is included, with detailed descriptions, in the `VirtuosoSubscriptionDataSource.h` protocol declaration.

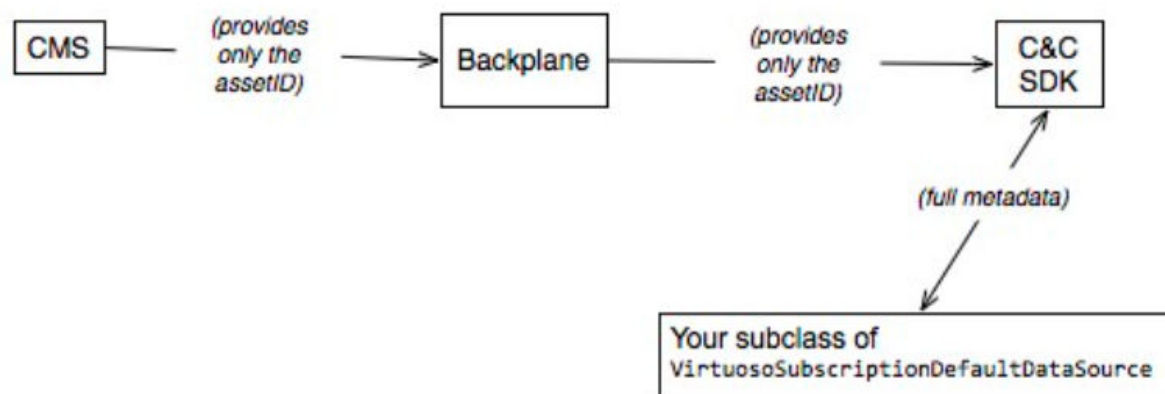
How does the client get this metadata? There are two options:

- **Scenario 1:** All required item metadata is provided to the Backplane



In this scenario, the `VirtuosoSubscriptionDefaultDataSource` class receives the metadata from the Backplane internally, automatically downloads and deletes episodes in the feed according to your defined settings, and no further configuration is required.

- **Scenario 2:** Only the feed item Asset UUID is provided to the Backplane via the web APIs when new feed content is announced



In this case, you must implement a custom data source class, which will provide metadata to the Subscription Manager. (The data source might fetch this information from anywhere, e.g. a web call to your CMS).

Penthera recommends that you subclass `VirtuosoSubscriptionDefaultDataSource` and implement at least `lookupMetadataForAssetID`. Once you have your custom data source, you must register it prior to calling the `VirtuosoSubscriptionManager` startup method:

```
[[VirtuosoSubscriptionManager instance] registerDataSource:
[CustomDataSource class]];
```

When implementing the `lookupMetadataForAssetID` method, you need to obtain all the required metadata in a synchronous manner and return that metadata via the method's return value. Review the `VirtuosoSubscriptionDataSource.h` protocol header for a complete list of required and optional metadata items. In addition to the protocol metadata items, you may return any custom metadata you wish as independent key-value pairs in the response dictionary or as serializable data in the user info dictionary metadata item.


```

- (NSDictionary*)lookupMetadataForAssetID:(NSString *)assetID
{
    // This example just returns the required metadata for
    // a hard-coded asset. Your implementation should lookup
    // all proper metadata for the provided asset ID.

    return @{@"SubscriptionContentDataAssetIDKey":assetID,
            SubscriptionContentDataDescriptionKey:@"my_asset_title",
            SubscriptionContentDataDownloadURLKey:@"http:download.url",
            SubscriptionContentDataCreatedDateKey:[NSDate date],
            SubscriptionContentDataMediaTypeKey:@(MediaTypeCodeVideo),
            SubscriptionContentDataUserInfoKey:@{@"arbitrary":@"info"},
            @"My_custom_metadata_key":@"my_custom_metadata_value"};
}

```

Configure Subscription Rules

The SDK follows two behavioral settings for subscriptions. These exist as properties on `VirtuosoSubscriptionManager`, and are global defaults. You may also set a feed-unique value when registering for a new subscription.

- **maximumSubscriptionItemsPerFeed** (default:0) The SDK tracks how many episodes it's downloaded in each feed. When this max value is reached, the behavior is dictated by the value of `autodeleteOldItems`. A value less than or equal to 0 indicates "unlimited."
- **autodeleteOldItems** (default:YES) Determines how the SDK behaves when a new episode is available and the device is already storing its quota from this feed. If YES, then the Subscription Manager will call the `VirtuosoAsset` `reset` method on the old items. The reset method deletes any downloaded files from local storage and resets the objects properties and status back to a pre-download state. The SDK reports the reset items in the proper notification. If NO, then the SDK will create a `VirtuosoAsset` object for the new episode, but won't automatically download it. These deferred items are reported in the proper notification.



NOTE

Only items downloaded through a subscription automatically count towards these rules. If you enqueue an episode of a feed manually, that enqueued item will not be automatically deleted or cause new downloads to be deferred unless you call the `VirtuosoSubscriptionManager` `includeItem:inTrackingFeed:` method with the episode.

Optional: Force a Sync

The Subscription Manager will automatically sync with the Backplane when the application is foregrounded, and at other opportune moments. If you would like to sync at some particular moment, with greater frequency, or in order to retrieve information not normally fetched during a standard sync, you can call one of the sync methods. The version of the method that takes no additional parameters will perform a standard sync immediately. The version that takes additional parameters can be used to retrieve metadata information about older items or about feeds that the user is not currently subscribed to.

These items will not be downloaded automatically, but will be returned in the data source, and can be used in your own internal processing.


```
// Standard sync
[[VirtuosoSubscriptionManager instance]
    syncSubscriptionsWithBackplaneNowOnComplete:^(Boolean success,
                                                    NSArray *subscriptions,
                                                    NSError *error)
    {
        // Handle response.
        // Note - Data from sync results will be sent via NSNotification
        // notices and via the data source processing methods.
    }
];
```

```
// Sync retrieving all items and all feeds
[[VirtuosoSubscriptionManager instance]
    syncSubscriptionsWithBackplaneNowForDataSince:[NSDate
dateWithTimeIntervalSince1970:0]
    returningOnlySubscribedFeeds:NO
    onComplete:^(Boolean success,
                  NSArray *subscriptions,
                  NSError *error)
    {
        // Handle response.
        // Note - Data from sync results will be sent via NSNotification
        // notices and via the data source processing methods.
    }
];
```

What Next?

The following publications may be of interest:

[Penthera 101: Introduction to Penthera Development](#)

[Penthera 201: Developing with the iOS SDK](#)

[Penthera 202: Developing with the Android SDK](#)

[Penthera 203: Best Practices](#)

[Penthera 301: iOS SDK Beyond the Basics \[3\]](#)

[Penthera 302: Android SDK Beyond the Basics](#)

[Penthera 310: DRM Deep Dive](#)

[Penthera 311: Server-to-Server with the Backplane API](#)

[Penthera 312: Known Issues](#)