

Penthera Download2Go™

Client Developer Guide (iOS)

This document describes integrating and using the **Penthera Download2Go iOS SDK**.

The SDK is the client piece of Penthera's Download2Go (Download2Go) platform, a software system that manages download, storage, and playback of videos on mobile devices. We assume that you will integrate the SDK into your own streaming video app that handles all UI/UX, user authentication, DRM, and video playout.

This document is a how-to guide. It will teach you to:

1. compile and run a sample iOS app using the SDK
2. import the SDK embedded framework into your iOS app
3. perform common functions using the SDK: enqueue, play, expire, configure, etc.

We assume you are an experienced iOS developer who knows your way around XCode and related iOS development tools.

The SDK communicates with a server, the **Download2Go Backplane**, using an internal, proprietary web services protocol. This communication occurs via regular client-server syncs and via server-to-client APN messages. Penthera hosts a developer server instance, at `demo.penthera.com` which you may use to build a proof-of-concept app.

Internally, the SDK is code-named "Virtuoso." You'll notice this a lot in the headers.

We're here to help! Email support@penthera.com if you run into any problems.

NOTE: This document contains method signatures and reference source code. We try to keep this document up-to-date, but you'll find the **authoritative** header files and reference source in the iOS developer package.

Table of Contents

Other Documentation

Let's Get Started

[Build And Use The Demo App](#)

[Add The Frameworks To Your Project](#)

[Add The Frameworks Directly To Your Project](#)

[Add The Frameworks Using CocoaPods](#)

[Other Project Setup](#)

Common Functions

[Enqueue an Item](#)

[Working With DRM](#)

[Setting Up Apple FairPlay](#)

[Setting Up Google Widevine](#)

[Background Downloading](#)

[Pause/Resume Downloading](#)

[Cancel Downloads](#)

[Clear SDK State](#)

[Set Availability Window for an Item](#)

[Enable/Disable Downloading](#)

[Configure Download Rules](#)

[Set up Logging](#)

[Play a Downloaded Item](#)

[Subscriptions](#)

Appendix A: How Downloading Works

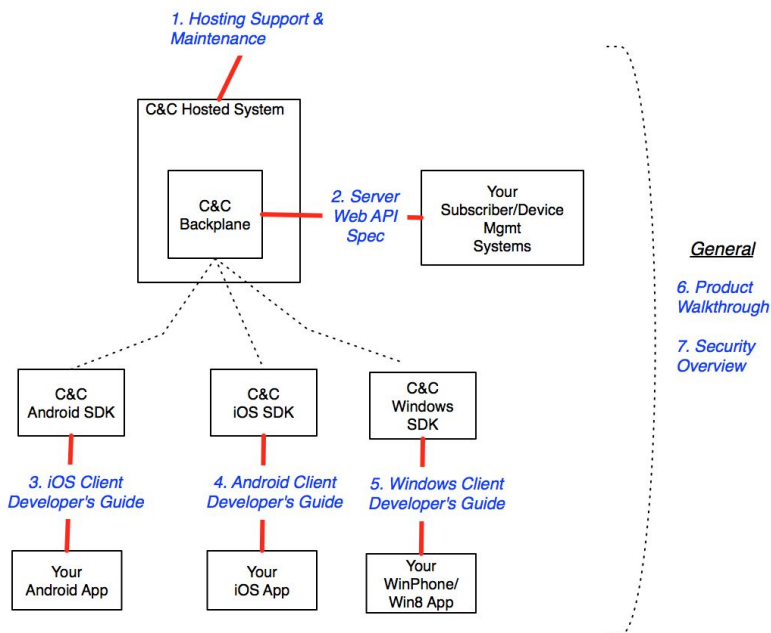
Appendix B: Upgrading

Appendix C: FAQ

Other Documentation

This document is part of a family of documents covering Download2Go:

Cache&Carry: Documentation "Map"



Let's Get Started

We'll provide you access to a private github repository where you can access the iOS developer package. You'll discover two pieces in the package:

- **VirtuosoClientEngine:** The libraries and supporting files that you'll include in your own iOS project
- **VirtuosoClientEngineDemo:** A standalone iOS app that includes the VirtuosoClientEngine.

Build And Use The Demo App

We provide the VirtuosoClientEngineDemo app as a convenience, so you can see how to use the SDK in a real app. The demo uses public-domain videos (HLS and mp4), hosted by Penthera on Amazon AWS.

To build and run the demo app:

1. Open the project. Make sure the provisioning profiles are set correctly. The SDK currently supports iOS 8.0 and later, so ensure that your build settings are configured for an appropriate deployment target.
2. Add the public/private key that Penthera gave you. (If we didn't, then just ask!) Look for the placeholder in `ViewController.m` where you need to include these keys. If you forget this step, the app will not compile.
3. Compile and run the app from XCode.

Congratulations! You've now got a video-downloading app up and running. You're ready to

develop your own apps with the SDK.

Add The Frameworks To Your Project

The SDK is packaged as an iOS framework. This allows XCode to automatically access header and resource files that it needs.

There are two ways to include the SDK in your project: directly importing and configuring the frameworks, or via CocoaPods.

Important Note: There are two versions of the SDK. The *VirtuosoClientDownloadEngine* and *VirtuosoClientSubscriptionManager* support MPEG-DASH/Widevine, but due to security restrictions cannot enable bitcode. The *VirtuosoClientDownloadEngine-Bitcode* and *VirtuosoClientSubscriptionManager-Bitcode* versions disable MPEG-DASH support and enable bitcode. If you do not need MPEG-DASH support, we recommend that you use the bitcode versions. All references to the frameworks in the below document use the base names for simplicity.

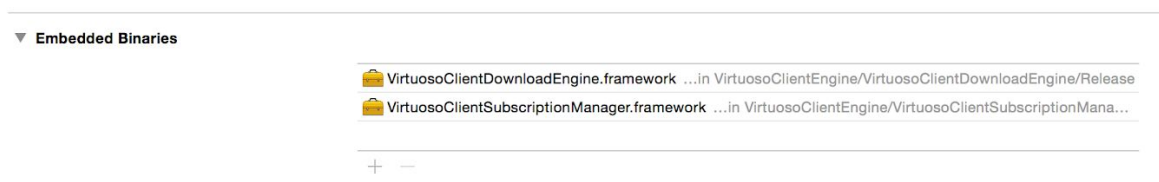
Add The Frameworks Directly To Your Project

To include the SDK in your project, navigate to the SDK *VirtuosoClientEngine* directory and drag both the *VirtuosoClientDownloadEngine.framework* and *VirtuosoClientSubscriptionManager.framework* directories into your XCode project.

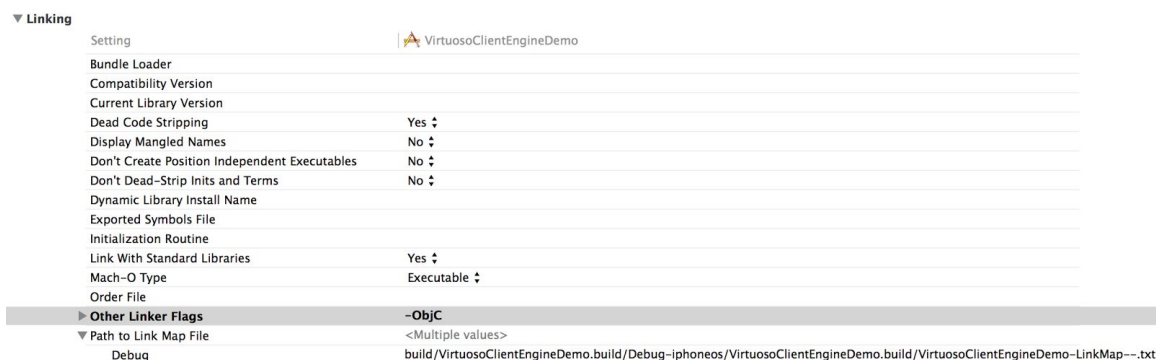
Note: As of iOS 8/XCode 6, Apple requires that frameworks be signed. Generally, unless there's a specific reason to do so, Penthera recommends that you always build with the release version of the SDK framework, regardless of the type of build you are creating in your own project.

Change your build settings

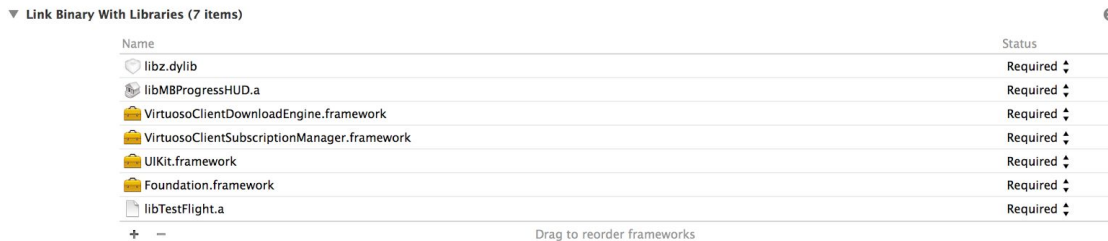
Under the General section in “Embedded Binaries”, add the *VirtuosoClientDownloadEngine.framework* and the *VirtuosoClientSubscriptionManager.framework*:



Under the Linking section in “Other Linker Flags,” , add the `-ObjC` flag:



In the “Link Binary With Libraries” build phase, ensure that both Virtuoso frameworks are included, and add the stock library, `libz.dylib`:



Add The Frameworks Using CocoaPods

You can also setup your project using the popular dependency management system, CocoaPods. To use CocoaPods, you'll need to follow these steps:

1. Add Penthera's pod spec repository to your CocoaPods installation by executing the command:
`pod repo add Cache-and-Carry-specs https://github.com/penthera/Cache-and-Carry-specs.git`

2. Reference Penthera's repository as a source in your Podfile:

```
source 'https://github.com/penthera/Cache-and-Carry-specs.git'
source 'https://github.com/CocoaPods/Specs.git'
```

3. Include the Penthera frameworks in your Podfile.

```
pod 'VirtuosoClientDownloadEngine', '~> 3.14.10'
pod 'VirtuosoClientSubscriptionManager', '~> 3.14.10'
```

4. Execute 'pod install' and you are ready to start coding.

Other Project Setup

Modify your app's info.plist

In your app's info.plist:

1. Add "Application Uses Wifi" and set it to YES.

This tells iOS to keep WiFi connections open in the background, rather than automatically timing out and transferring the connection to cellular. This improves download performance and minimizes cellular data usage.

2. In "Required Background Modes", add the option "App downloads content in response to push notifications".

Alternatively, do this by going to the "Capabilities" application tab, in the "Background Modes" section, and enabling the "Remote Notifications" checkbox.

3. Optionally, in "Required Background Modes", add the option "App downloads content from the network".

Alternatively, do this by going to the "Capabilities" application tab, in the "Background Modes" section, and enabling the "Background fetch" checkbox.

Add a run script build phase

In the Build Phases tab of your application project:

1. Click the "+" button in the upper left corner to add a new build phase. Select "New Run Script

Phase” and make sure that the new build phase gets added to the end of the list.

2. Copy the following command into the script text area:

```
bash
"${BUILT_PRODUCTS_DIR}/${FRAMEWORKS_FOLDER_PATH}/VirtuosoClientDownloadEngine.framework/strip-frameworks.sh"
```

These actions work around a known XCode 7.2 app submission bug that disallows universal (device + simulator) frameworks to be submitted to the App Store or TestFlight. The script strips out the unused architectures from the SDK frameworks based on the type of build you are creating and automatically resigns the frameworks appropriately.

You should now be able to compile your project successfully with no build errors.

A note about iOS Application Transport Security

In iOS 9, Apple added “Application Transport Security” (ATS), which causes non-SSL connections to be rejected in apps built with the iOS 9 SDK or greater. Apple has also announced that all apps will be required to use ATS by 2017. All Penthera Backplane communications are SSL enabled. In iOS 9, the ATS system also disallowed non-SSL connections to localhost. If you are supporting iOS 9, for video playback to function, you need to add an ATS exception for “localhost” into your info.plist NSAppTransportSecurity key, as follows:

```
<key>NSAppTransportSecurity</key>
<dict>
  <key>NSExceptionDomains</key>
  <dict>
    <key>localhost</key>
    <dict>
      <key>NSExceptionAllowsInsecureHTTPLoads</key>
      <true/>
    </dict>
  </dict>
</dict>
```

In iOS 10, Apple allows connections to localhost without any ATS exceptions, and the Download2Go SDK will fully function without any security exceptions.

Include Header

Anywhere you want to access SDK classes or methods, you must import the main header:

```
#import <VirtuosoClientDownloadEngine/VirtuosoClientDownloadEngine.h>
```

We’ll assume this line is included in all the examples below. If you have enabled precompiled headers for your project, we suggest you include this import in your app’s .pch prefix file. This will automatically import the .h file in all your code by default.

Initialize the Engine

VirtuosoDownloadEngine is the main SDK class. It is a singleton. Call the instance method to initialize the object and get a pointer to the valid instance:

```
VirtuosoDownloadEngine* engine = [VirtuosoDownloadEngine instance];
```

VirtuosoDownloadEngine automatically cleans itself up upon receiving critical system events, like app termination or low memory conditions.

Startup

Penthera will provide you the URL of your Backplane instance, along with an app-specific public/private key pair that allows your SDK instances to authenticate to your Backplane instance.

To prepare the SDK for use, call `startupWithBackplane`, and provide this URL. Invoke this method as early on in the startup process as possible.

Sample Startup Code

Insert the following code inside your application:didFinishLaunchingWithOptions AppDelegate method:

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *) launchOptions {

    // Configure logger delegate messages
    [VirtuosoLogger addDelegate:self];

    // Configure logging
    [VirtuosoLogger setLogLevel:kVL_LogWarning];
    [VirtuosoLogger enableLogsToFile:NO];

    // Initialize engine and fetch a handle to the singleton instance
    VirtuosoDownloadEngine* engine = [VirtuosoDownloadEngine instance];

    // Global "on switch" for downloading
    [engine setEnabled:YES];

    [engine startupWithBackplane:@https://penthera.provided.url"
        user:@"an_id_you_assign_for_this_user"
        externalDeviceID:@"an_id_you_assign_for_this_device"
        privateKey:@"penthera_provided_private_key"
        publicKey:@"penthera_provided_public_key"];

    ...
}
```

It's your responsibility to supply a unique user ID to the SDK when you call `startupWithBackplane`. The SDK uses this user ID in reporting, and to enforce business rules (such as "max number of download-enabled devices per user"). If you don't know the user ID by the time `didFinishLaunchingWithOptions` executes, you'll need to delay calls to `startup` until you know the user ID, such as after an opening user login dialog. You'll also need to delay registering for push notices in your app until after the engine has been started. For an example, see the SDK Demo app.

For more details on UserID, DeviceID, and related topics, see the FAQ at the end of this document.

Enabling Push Messages

The Download2Go Backplane sends APN (push messages) to the SDK to trigger certain actions, e.g. download and delete. To send a push message to a device, the Backplane needs a push token from your app on the device. Within your app, you need to feed this push token to the SDK, so the SDK can upload it to the Backplane.

Here's how:

```
// At this point, tokenString should contain the app-specific push token
- (void)application:(UIApplication *)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)deviceToken {
    [VirtuosoSettings instance].devicePushToken = [deviceToken description];
}

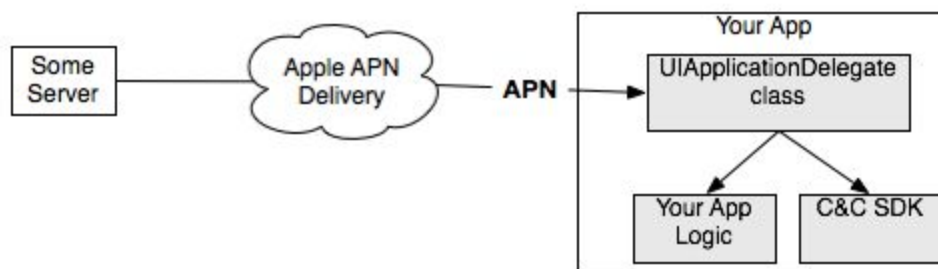
- (void)application:(UIApplication *)application
didFailToRegisterForRemoteNotificationsWithError:(NSError *)error {
    // oops, something went wrong with the token
    [VirtuosoSettings instance].devicePushToken = nil;
}
```

Handling SDK-Related Events

Push Notices

From time to time, a server (the Download2Go Backplane or another server associated with your app) may send your app an APN signal.

The signal will land in your UIApplicationDelegate class. From there, you need to dispatch the signal to whoever it belongs to: either the Download2Go SDK or your own code.



If you drop the following method into your UIApplicationDelegate class, the APN messages will go where they're supposed to:

```

- (void)application:(UIApplication *)application didReceiveRemoteNotification:(NSDictionary *)userInfo fetchCompletionHandler:(void (^)(UIBackgroundFetchResult))completionHandler {
    if( [VirtuosoEventHandler processRemotePushNotice:userInfo
        withCompletionHandler:completionHandler] ) {
        // SDK handled the push signal. No action required.
    }
    else {
        // SDK did not handle the push signal.
        // Handle the signal yourself - call your handler code or insert it here
    }
}

```

Background Download Session Notices

The SDK uses an internal background download session. To handle background session events properly, insert the following into your UIApplicationDelegate class.

```

- (void) application:(UIApplication*)application
    handleEventsForBackgroundURLSession:(NSString *)identifier
    completionHandler:(void (^)(void))completionHandler
{
    if( [VirtuosoEventHandler processBackgroundSessionWake:identifier
        completionHandler:completionHandler] ) {
        // SDK handled the wake. No action required.
    }
    else {
        // SDK did not handle the wake.
        // Handle the signal yourself - call your handler code or insert it here
    }
}

```

Background Fetch Requests

If you choose to enable the background fetch option for your app, the SDK can also make use of this background processing time to perform certain maintenance activities, such as syncing with the backplane and posting log events. To enable this feature, insert the following into your UIApplicationDelegate class.

```

- (void)application:(UIApplication *)application performFetchWithCompletionHandler:(void (^)(UIBackgroundFetchResult))completionHandler
{
    // Process the fetch in the SDK.
    [VirtuosoEventHandler
        processFetchWithCompletionHandler:^(UIBackgroundFetchResult result) {
            // Do any synchronous or asynchronous processing you need to do here.
            // Just make sure to merge your own fetch result with the SDK fetch result....
            if( completionHandler != NULL )
                completionHandler(result);
        }];
}

```


Listening for the SDK's NSNotification messages

The SDK sends out various NSNotification messages to let your app know about changes in SDK status. Make sure your app registers to receive whichever notifications you care about:

```
extern NSString* kDownloadEngineStatusDidChangeNotification;
extern NSString* kDownloadEngineDidStartDownloadingAssetNotification;
extern NSString* kDownloadEngineProgressUpdatedForAssetNotification;
extern NSString* kDownloadEngineProgressUpdatedForAssetProcessingNotification;
extern NSString* kDownloadEngineInternalQueueUpdateNotification;
extern NSString* kDownloadEngineDidFinishDownloadingAssetNotification;
extern NSString* kDownloadEngineDidEncounterErrorNotification;
extern NSString* kDownloadEngineDidEncounterWarningNotification;
extern NSString* kDownloadEngineIsEnteringBackgroundNotification;
extern NSString* kBackplaneDidUnregisterDeviceNotification;
extern NSString* kBackplaneDeviceLimitReachedNotification;
extern NSString* kBackplaneInvalidCredentialsNotification;
extern NSString* kBackplaneDeviceAlreadyRegisteredNotification;
extern NSString* kBackplaneCommunicationsFailureNotification;
extern NSString* kBackplaneSyncResultNotification;
extern NSString* kBackplaneDeviceSaveResultNotification;
extern NSString* kBackplaneLogsSentNotification;
extern NSString* kBackplaneRemoteKillNotification;
extern NSString* kProxyDidEncounterErrorNotification;
extern NSString* kDownloadEngineDidResetExpiredAssetsNotification;
extern NSString* kDownloadEngineDidBeginDataStoreUpgradeNotification;
extern NSString* kDownloadEngineDidFinishDataStoreUpgradeNotification;
```

Look for the latest set of these notifications in `VirtuosoNotifications.h`. Implementing any of these notifications will give you the appropriate status update.

Common Functions

Here we list common ways to use the SDK. This is just a sliver of the overall SDK functionality; after you're done here, have a look at the API to see what else is available.

Enqueue an Item

In the SDK, every downloadable object is a `VirtuosoAsset`. You may enqueue a `VirtuosoAsset` by calling one of the below methods:

Enqueue a Single (Flat) File (e.g. mp4)

To enqueue a single file, create an instance of `VirtuosoAsset`, then add the asset to the SDK's queue after the completion block is invoked:

```
[VirtuosoAsset
  assetWithRemoteURL:@"http://path/to/file.mp4" // downloaded file
    assetID:@"your_unique_asset_id"
    description:@"Test File" // not used by the SDK
    publishDate:nil // available immediately
    expiryDate:nil // don't ever expire
    permittedMimeTypes:[@"video/mp4"] // for validation
    userInfo:nil // your own info; not used by SDK
  onReadyForDownload:^(VirtuosoAsset* parsedAsset)
  {
    [[VirtuosoDownloadEngine instance] addToQueue:parsedAsset atIndex:NSUIntegerMax];
  }
  onParseComplete:^(VirtuosoAsset* parsedAsset)
  {
  }
];
```

Refer to the SDK header files for a full description of the behavior and syntax of every parameter.

Enqueue an HLS Video

The SDK treats HLS videos as a special case. The SDK can automatically configure and download all the required HLS fragments from an m3u8 manifest.

To enqueue an HLS manifest, create an instance of `VirtuosoAsset`, then add the asset to the SDK's queue after the completion block is invoked:

```
[VirtuosoAsset
    assetWithAssetID:@"your_unique_asset_id"
    description:@"Test HLS Video"
    manifestUrl:@"http://path/to/main/manifest.m3u8"
    protectionType:kvDE_AssetProtectionTypePassthrough
    includeEncryptionKeys:YES
    maximumBitrate:INT_MAX // i.e. use the highest available profile
    publishDate:nil
    expiryDate:nil
    userInfo:nil
    onReadyForDownload:^(VirtuosoAsset* parsedGroup)
    {
        [[VirtuosoDownloadEngine instance] addToQueue:parsedAsset atIndex:NSUIntegerMax];
    }
    onParseComplete:^(VirtuosoAsset* parsedAsset)
    {
    }
    }];
```

Note: `maximumBitrate` specifies which HLS profile the SDK should select for download, from among the HLS profiles available in the manifest. The SDK will download the the highest bitrate not exceeding `maximumBitrate`. If no profile exists lower than the provided maximum, then Virtuoso will select the lowest bitrate profile. Set `maximumBitrate=0` to force the SDK to use the lowest profile, or to `INT_MAX` to use the highest profile.

Note: You should set `protectionType` to be `kvDE_AssetProtectionTypePassthrough` if you are not using DRM, or if you are using a DRM type that is not built-into `Download2Go`. If you are using DRM protection, you should most likely set `includeEncryptionKeys` to `NO`. Take a look at the definition of `kvDE_AssetProtectionType` in `VirtuosoConstants.h` for a list of built-in DRM systems.

Enqueue an HSS Video

HSS works similarly to HLS:

```
[VirtuosoAsset
    assetWithAssetID:@"your_unique_asset_id"
    description:@"Test HSS Video"
    manifestUrl:@"http://path/to/main/video.ism/Manifest"
    maximumVideoBitrate:INT_MAX // i.e. use highest available video profile
    maximumAudioBitrate:INT_MAX // i.e. use highest available audio profile
    publishDate:nil
    expiryDate:nil
    userInfo:nil
    onReadyForDownload:^(VirtuosoAsset* parsedAsset) {
        [[VirtuosoDownloadEngine instance] addToQueue:parsedAsset atIndex:NSUIntegerMax];
    }
    onParseComplete:^(VirtuosoAsset* parsedAsset) {
    }
    }];
```

Unlike HLS, the HSS format uses a separate audio and video data stream. So instead of specifying `maximumBitrate`, you need to specify both `maximumVideoBitrate` and `maximumAudioBitrate`. As above, the SDK will select and download the profile with the highest bitrate not exceeding the values you specify.

Working With DRM

Most Penthera customers use Download2Go in conjunction with a commercial DRM system. Download2Go includes built-in support for Google Widevine (iOS 8+) and Apple FairPlay (iOS 10+). Penthera's SDK doesn't require any integration with most other DRM systems. The DRM system handles license delivery and secure playout of the video, while Download2Go handles delivering the (DRM-encrypted) video to the mobile device and serving the downloaded video to the player via a local web proxy.

If you are using Penthera's built-in FairPlay or Widevine support, then you will not need to do anything yourself to fetch or persist DRM licenses. The SDK will automatically request a cachable 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. **If you are integrating with a different DRM system, you must handle DRM license management yourself. We recommend that you fetch the DRM license before you call a Download2Go "enqueue" method.** 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 download finishes while the app is in the background, your app may not receive a 'download-finished' notification from the SDK until the user re-opens the app. At this point, the device may be offline and the app can't fetch the license.

Setting Up Apple FairPlay

The Download2Go SDK automatically supports offline playback of FairPlay-protected videos, provided that your FairPlay license server grants the appropriate permissions for the license. Enabling FairPlay in Download2Go requires only the following:

1. Configure your FairPlay License Server and download the client's FairPlay application certificate during application startup:

```
[VirtuosolicenseManager
    setLicenseServerURL:<license_server_url>
    forDRM:kVLM_FairPlay];
[VirtuosolicenseManager
    downloadClientAppCertificateFromURL:<certificate_url>
    forDRM:kVLM_FairPlay];
```

2. When you start a new download, pass `kVDE_AssetProtectionTypeFairPlay` as the `protectionType` parameter.

If you are using one of Download2Go's built-in player classes, that's all you need to do.

Setting Up Google Widevine

The Download2Go SDK automatically supports offline playback of Widevine-protected videos, provided that your Widevine license server grants appropriate permissions for the license. Enabling Widevine in Download2Go 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 Download2Go 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 Download2Go's built-in player classes, that's all you need to do.

Background Downloading

Apple strictly constrains what an app can do when it's not in the foreground. The SDK provides as much functionality as possible given these constraints.

Download Limits: The SDK makes use of iOS's BTS (Background Transfer Service) to download while the enclosing app is not actively running. The BTS policies around download are intentionally opaque, but empirically, Penthera has observed that BTS will download up to around 4GB per 24-hour period per app, then will suspend BTS downloading for that application until the next day. This is unpublished behavior by Apple and may change with future iOS versions.

Downloading Segmented Video: The iOS BTS system is not designed to process hundreds or thousands of individual files, and bad things happen when you try to enqueue lots of files with BTS. Download2Go solves this problem using a *StreamPackager*. The *StreamPackager* is a proxy server that sits between your origin server and the SDK, and serializes many .ts files into a monolithic stream which it delivers to the device.

To instruct the SDK to use the *StreamPackager*, set the `useStreamPackagerForBackgroundDownloads` property to YES (the default). To disable this feature, you must set the property to NO **prior** to calling the SDK startup method. If set to NO, downloading segmented video will proceed more slowly (or not at all) when the app enters a suspended state.

Pause/Resume Downloading

Use the 'enabled' flag to start and pause downloads:

```
Pause:  [[VirtuosoDownloadEngine instance]setEnabled:NO];
Resume: [[VirtuosoDownloadEngine instance]setEnabled:YES];
```

Cancel Downloads

To cancel one download:

```
[[VirtuosoDownloadEngine instance] removeFromQueue:asset];
```

To cancel *all* downloads:

```
[[VirtuosoDownloadEngine instance] flushQueue];
```

Clear SDK State

If the enclosing app were to de-authenticate a user, the app may at the same time also wish to delete all downloaded videos:

```
[VirtuosoAsset deleteAll];
```

In addition, the administrator may choose to schedule a remote wipe of any device. See the Backplane documentation for additional details.

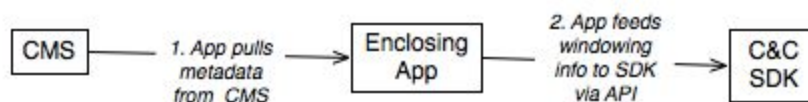
Set Availability Window for an Item

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

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 Backplane stores a global default value for EAP and EAD. You may set these values through the Backplane web API. The Backplane transmits these default values to all SDK instances.

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:



Step 2 in the above diagram occurs when you create the VirtuosoAsset object. You can also modify the values later, via the appropriate class properties.

```

[VirtuosoAsset
  assetWithRemoteURL:@"http://path/to/your/asset.mp4"
  assetID:@"your_unique_asset_id"
  description:@"Test Video"
  publishDate:[NSDate date] // Available now
  expiryDate:[NSDate dateWithTimeIntervalSinceNow:604800] // Expires in 7d
  expiryAfterDownload:86400 // Expires 24h after download
  expiryAfterPlay:43200 // Expires 12h after play
  permittedMimeTypes:nil
  userInfo:nil
  onReadyForDownload:^(VirtuosoAsset* parsedAsset)
{
  [[VirtuosoDownloadEngine instance] addToQueue:parsedAsset atIndex:NSUIntegerMax];
}
  onParseComplete:^(VirtuosoAsset* parsedAsset)
{
}
];
  
```

Each of the API'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.

Enable/Disable Downloading

Download2Go provides two switches that control download. They have different purposes. The SDK only downloads if both toggles are 'on'.

SDK "Master Switch": A Boolean property on the `VirtuosoSDK` instance in the SDK that you can set to toggle downloading. Use this to disable downloads, for example, when your app is streaming video and you don't want to share bandwidth between streaming and download. **Be careful: this value persists across app restarts.**

```
@property (nonatomic,assign) Boolean enabled;
```

Backplane-Enforced Toggle: The Backplane enforces a limit on the number of Devices that each User may have enabled for download. To do so, the Backplane maintains, for each known Device, a flag indicating whether the Device is permitted to download. If new devices are created in a user account, and the limit hasn't yet been reached, then the new devices are automatically enabled on the Backplane. If the limit has been reached, then newly-added Devices are disabled from downloading via the device property:

```
@property (nonatomic,readonly) Boolean downloadEnabled;
```

You can request a change to this flag by calling the following method. The device needs to be online and able to connect to the Backplane for this to succeed.

```
- (void) updateDownloadEnabled:(Boolean)enabled
onComplete:(DeviceUpdateResultBlock)onComplete;
```

NOTE: The Backplane also offers a web API to do this; see the server documentation.

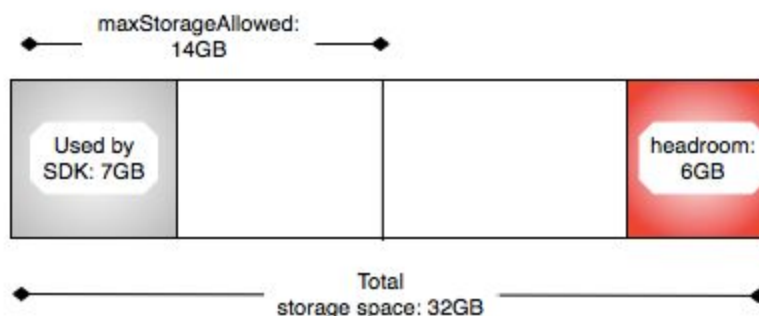
Configure Download Rules

Several behavioral properties control the SDK's behavior:

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

long long maxStorageAllowed (in MB; default=LONG_LONG_MAX): disk space that the SDK will download and manage on the device. While downloading in the foreground, the SDK will continue to download until 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`.

long long headroom (in MB; default=1024): disk space that the SDK will leave available on the device. While downloading with the app in the foreground, the SDK will continue to download until headroom space is left 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.



Visualizing `maxStorageAllowed` and `headroom` parameters. Here the device has 32GB disk space. The SDK will always preserve 6GB free space on disk. Currently, the SDK is using 7GB, and will never use more than 14GB total.

Boolean **`downloadOverCellular`** (default: false): 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.

NSString **`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.

Set up Logging

There's two separate logging paths in the SDK:

Event Logging

The SDK can capture many different events of potential business value, e.g. when a download starts and stops, offline playout, etc.

You can configure which of these events the SDK uploads to the Backplane. By default, most events are enabled; you should choose which events you want to enable or disable. Available events are listed in the `kVL_LogEvent` enumeration.

Use the `setLoggingEnabled:forEvent:` or `setLoggingEnabledForAllEvents:` methods to enable event logging. We suggest you do this prior to the logger startup call.

Note: You can't disable the "download queued", "download start", "download complete", "download error", "max errors reset", or "reset" events. Attempts to do so will have no effect.

Debug Logging

The SDK generates lots of developer-friendly debug information. You can send this to various locations (console, log file) or you can implement a logger delegate and handle it manually.

By default, the SDK logging system is very quiet. To configure logging, use these just after the logger startup call:

- `[VirtuosoLogger addDelegate:id<VirtuosoLoggerDelegate>]:` Adds the indicated delegate to the logger. Delegates must follow the `VirtuosoLoggerDelegate` protocol. Methods in the protocol can be used to receive SDK events and handle them in your own custom logging mechanisms.
- `[VirtuosoLogger setLogLevel:kVL_LogVerbose]:` Configures verbose console logging. See SDK headers for available log levels.
- `[VirtuosoLogger enableLogsToFile:YES]:` Enables all log output to be logged to a file in the app's documents directory.

Play a Downloaded Item

The SDK provides a set of player-related classes to help make playback easier. Use of these classes is optional, but does provide some convenience. Using the built-in classes automatically handles built-in DRM licensing and appropriate logging of the “play start” and “play stop” log events.

If you need to use your own closed-source player, or need more control over aspects of playback than the built-in classes allow, you may also choose to use the SDK’s local HTTP proxy, `VirtuosoClientHTTPServer`, that sits between a media player and downloaded HLS and HSS items.

There are three built-in classes you can use to make playback easier:

- **VirtuosoAVPlayer:** A drop-in replacement for Apple’s AVPlayer class. You can use `VirtuosoAVPlayer` anywhere you already have an AVPlayer to enable automatic FairPlay and Widevine licensing and event logging.
- **VirtuosoPlayerView:** A UIView subclass built on top of `VirtuosoAVPlayer`. It provides a basic user interface that you can build from, or disable entirely and add your own user interface elements.
- **VirtuosoPlayerViewController:** A full view controller built on top of `VirtuosoPlayerView` and `VirtuosoAVPlayer`. For rapid prototypes and proof of concept applications, this class provides the fastest mechanism to play back MP4, HLS, and DASH videos encoded with FairPlay or Widevine.

There are three ways to play a `VirtuosoAsset`:

- **filePath:** If you have downloaded a standalone video file, such as an MP4 or ISMV file, then you can access the downloaded file directly for playback.
- **playUsingPlaybackType:fromViewController:onSuccess:onFail:** This method automatically handles windowing, creation and maintenance of the `VirtuosoClientHTTPServer` instance used for playback, and playback itself. It uses a `VirtuosoPlayerViewController` for playback. Call this method from the view controller you wish to present the video player from. This method is unsupported for HSS video assets.
- **playUsingPlaybackType:andPlayer:onSuccess:onFail:** If you need to use a custom player, you can use this method instead. This method automatically handles windowing and creation and maintenance of the `VirtuosoClientHTTPServer` instance used for playback. It is your responsibility to further configure the player, to start playback if necessary, and to present the player in the UI hierarchy. If you are using a more complex player, additional integration steps may be required, such as using a `VirtuosoClientHTTPServer` instance directly.

NOTE: Be sure to set the first playback timestamp, as shown below, when the item starts playback, so the SDK can later enforce “expiry after playback.”

```
VirtuosoAsset* asset = [VirtuosoAsset assetWithUUID:
                        [files objectAtIndex:indexPath.row]
                        availabilityFilter:NO];

self.player = [VirtuosoPlayerViewController alloc]init];

[asset playUsingPlaybackType:kVDE_AssetPlaybackTypeLocal
    andPlayer:(id<VirtuosoPlayer>)self.player

    onSuccess:^(
    {
        // Present the player
        [self.navigationController presentViewController:self.player
            animated:YES
            Completion:nil];

        // We need to set the playback date here.
        [asset setFirstPlayDateTime:[NSDate date]];
        [asset save];
    }

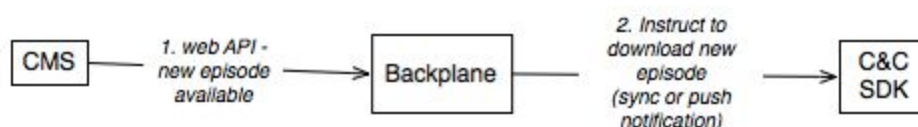
    onFail:^(
    {
        [self showErrorForAsset:asset atIndexPath:indexPath];
    }
    ]];
```

Subscriptions

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.”

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.

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 Data Source

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.

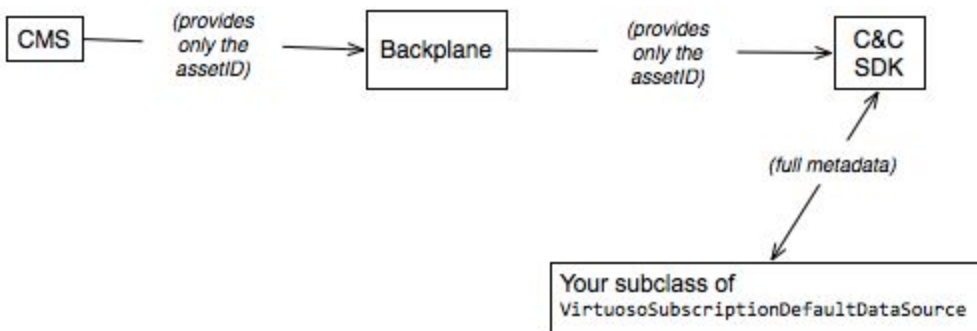
How does the client get this metadata? There's two options:

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"};
}
```

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
    }];
```

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.

}];
```

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.

- **int 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."
- **Boolean 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.

Appendix A: How Downloading Works

This section is for the curious developer. You don't need to understand this in order to use the SDK.

Virtuoso follows a "Rule of Threes" in downloading:

1. Proceed through the download queue in order. Virtuoso will download multiple file segments at a time (in parallel), but avoids downloading segments belonging to different assets simultaneously.
2. If Virtuoso encounters an error downloading the asset, it will try that asset two more times before moving on. (This is the "inner" three).
3. When it reaches the end of the download queue, Virtuoso will return to the beginning of the queue and make another pass, trying to download the errored files.
4. Once it has encountered an asset in three separate passes through the queue, Virtuoso will mark the asset in error and will no longer try to download the asset, until you reset that asset using the `VirtuosoAsset clearRetryCount` method. (This is the "outer" three).

In fact, the above is a slight simplification. If Virtuoso encounters a "fatal" error in step 2, it will not retry the file later. An error is fatal if there's no point retrying later (e.g. the HTTP server advertises an unexpected mime type.) Virtuoso may also choose to retry failed assets at opportune moments, such as when the user returns to the app.

Virtuoso communicates download issues to the enclosing app in two ways.

1. When the Virtuoso encounters an issue that will eventually cause the file to be marked as blocked, it issues a `kDownloadEngineDidEncounterErrorNotification`. It will send this notice even if the file download will be retried.
2. When Virtuoso determines a potential issue exists (such as when the server reported size and expected size do not match), it issues a `kDownloadEngineDidEncounterWarningNotification`. This notice indicates that something unexpected happened, but the file download will still finish and be marked as successfully completed.

In the case of both notices, the `userInfo` dictionary sent with the notice will contain an `NSError` object in the `kDownloadEngineNotificationErrorKey` that contains detailed information about the error that was encountered and the `VirtuosoAsset` that caused the error will be contained in the `kDownloadEngineNotificationAssetKey`.

The following chart summarizes error conditions and Virtuoso's behavior. For the most up-to-date list of errors, see the `kVDE_DownloadErrorCode` enumeration in `VirtuosoConstants.h`.

Condition	Description	Retry?
Invalid mime type	The MIME type advertised by the HTTP server for the file isn't included in the MIME types whitelist you provided earlier.	No
Final file segment size disagrees with server-provided segment size	After a segment download completes, the on-disk file size didn't match the expected size, as reported by the server.	Yes
Network error	Some network issue (HTTP 404,416, etc.) caused the download to fail.	Yes
File System Error	The OS couldn't write the file to disk. In most cases, the root cause is a full disk.	Yes

Appendix B: Upgrading

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:

- `kDownloadEngineDidBeginDataStoreUpgradeNotification`
- `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.

Appendix C: FAQ

I'm confused about UserID, DeviceID, and External Device ID

A “User” is a person, household, or other entity that owns a device. When you call VirtuosoDownloadEngine’s `startupWithBackplane` method, you must supply a **UserID**. but Meanwhile, the SDK uses its own internal logic to assign a unique **DeviceID** to the device. The SDK uploads this (UserID, DeviceID) pair to the Download2Go Backplane, which associates the Device with the User. The Download2Go Backplane only uses this (User, Device) pair to enforce the “max download-enabled Devices per User” rule.

The **External Device ID** is a field we maintain for your convenience. You can look up a device’s activity on the Backplane using the external Device ID. You can perform a “remote-delete” or “remote-wipe” from the Backplane, providing it an External Device ID. The Backplane provides convenience mechanism to map an External Device ID to Penthera’s internal DeviceID.

How does the SDK guarantee the DeviceID persists across installs?

The SDK stores the DeviceID in the device keychain. This allows the DeviceID to persist across installs, and prevents it from changing under most conditions. During development, it’s possible for the DeviceID to change, if you do anything on the device or within your build that would reset the application area in the device keychain. This could include a factory reset on the device, changing the app bundle ID, or other infrequent operations. Since Apple removed the iOS SDK “device ID” value and forbid using the “advertising ID,” there’s no way to 100% guarantee a consistent, perpetual device ID. And, Apple’s “identifier for vendor” does not persist across installs. The DeviceID value that Penthera uses will persist across app upgrades/installs, but some limited user actions may reset it. This will be a very infrequent thing, and the only reason the device ID value should change in production code would be if the user actively took steps to reset their device keychain. Apple’s stance is that app developers should treat that as a “new device” anyway.

How does the SDK decide what order to download queued assets?

In general, assets will be downloaded in the order that they were added to the queue. This is not, however, guaranteed, and various internal rules may allow one asset to be downloaded before another.

**** END OF DOCUMENT ****