Penthera Download2Go™

Client Developer Guide (Android)

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

The SDK is the client piece of Penthera's Download2Go (Download2Go), 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 Android app using the SDK
2. link the SDK into your Android app
3. perform common functions using the SDK: enqueue, play, expire, configure, etc.

We assume you are an experienced Android developer, and you're using Android's latest SDK and platform tools. Although the SDK does not require it, some examples in this document assume you are using Android Studio.

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 FCM 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 codenamed "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.

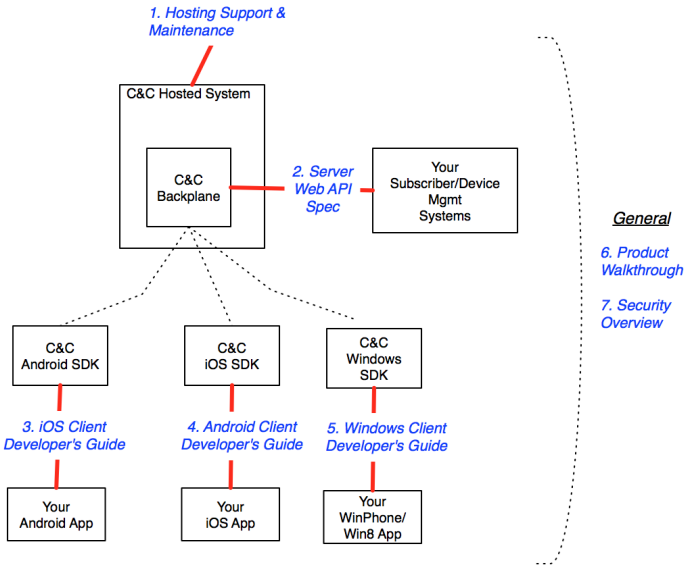
NOTE: This document contains method signatures and reference source code. We try to keep this document up-to-date, but you'll find the **authoritative** header files and reference source in the Android developer package.

Table of Contents

Other Documentation

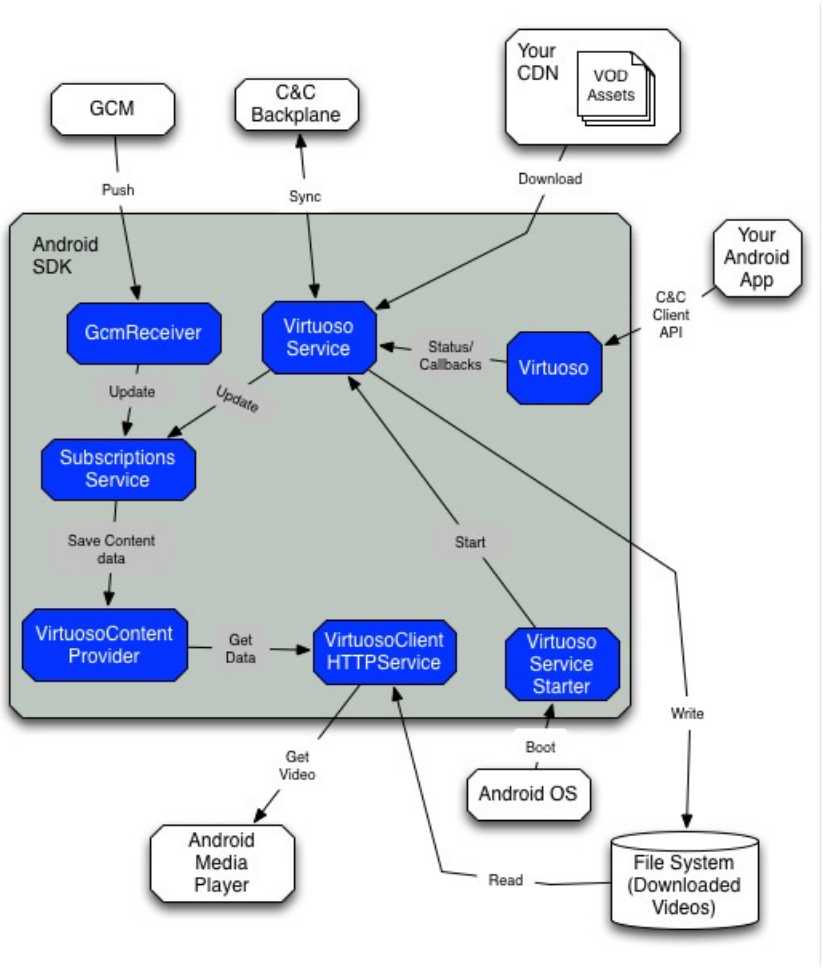
This document is part of a family of documents covering Download2Go:

Cache&Carry: Documentation "Map"



Download2Go Android Architecture

Overall SDK Architecture



Android SDK architecture overview. For simplicity, not all components/interconnects are shown.

The SDK consists of a few components:

- **Virtuoso**: For the most part, this is your app's main interface with the SDK. Provides interfaces to register the device, manipulate the download queue, subscribe to feeds and get the status on the download service.
- **Virtuoso Service**: A background service running in its own process. Downloads queued assets, deletes expired assets, communicates with the Backplane, and sends notifications to the enclosing app.
- **Push Notification Receiver and Services**:
    - ADMReceiver: A Receiver to handle the Amazon Device Messaging (ADM) Client broadcasts.
    - ADMService: The Service which handles Amazon Device Messaging (ADM) messages forwarded from the receiver.

    - FcmInstanceIdService: Service which handles Firebase Instance Id token refreshes.
    - FcmMessagingService: Service which handles the Firebase Cloud Messaging (FCM) notifications.
- **Subscriptions Service**: Handles the synchronization of subscribed feeds, manages when new episodes should be added for download and deletes old episodes.
- **Virtuoso Content Provider**: Keeps track of events and all information regarding assets known to the SDK.
- **Virtuoso Client HTTP Service**: A proxy for playout of segmented assets, e.g. HLS videos.
- **Virtuoso Service Starte**r: A small Broadcast Receiver based class that starts the Virtuoso Service. This ensures that a valid Notification is available from the client app to run as a foreground service. It also monitors environmental events to ensure the service restarts when required..

Upon installation, apps first begin in a stopped state. In a stopped state, an app won't receive any broadcast messages. This means that background services can't be automatically started, unless the app has first been launched by the user.

Foreground Download Service

The download service needs to run as a foreground service in Android to continue running for downloads, backplane comms, or expiry processing while the app is in the background. A notification must be displayed for an Android foreground service. Notifications for the service must be delivered via the service, which runs it a separate process. In order to ensure the service will run successfully, the SDK requests a first notification prior to requesting the service starts up. This process is controlled by the Service Starter component. This will occur every time the SDK tries to start the service, or check that it is running, and can be during app startup or during processing of push messaging or expiry processing while the app is not open. It is important to always return a valid notification to these requests. Each request for a notification includes details of an Intent for the action taking place, and optionally also an asset. On initial startup the Intent may be Null.

Once the service is running, updates to the notifications can be made either using the same process as the initial startup, or by registering a class with the SDK which will be instantiated and used to request notifications from within the service process. It is important to bear the following points in mind when choosing the approach to take:

1) Notifications that are generated via the Service Starter and passed into the service will be passed over a process boundary. Technically this should be fine as they implement the Parcelable interface, however there are limits to the size of objects passed across the process boundary and these limits will impact on the maximum size of images you can include in the notifications. This is important for the initial startup notification, which will always be provided using this approach.
2) Notifications that are generated within the service using a class implementing IForegroundNotificationProvider will not experience any size limits as they do not get passed across any process boundaries. However, the class to create the notifications will be generated within the service process, meaning that any dependencies it creates will also be instantiated within this process. It is important to bear this in mind as it can result in classes being instantiated in multiple processes and causing a large memory footprint for the application.

Broadcast Receivers

The SDK utilizes a number of broadcasts to pass events between the background download process and the UI process. A number of broadcast messages are also made available in the public interface for the UI to use directly to respond to events. Since Android API 26, implicit android broadcasts are no longer permitted within applications, as a result the broadcasts within the SDK are primarily targeted towards an internal component within the main download service. For service starter related broadcasts, the SDK seeks a broadcast receiver implementing the service starter interface, but for public broadcasts the appropriate receivers must be declared within the application manifest. Exclusion of any documented receivers will result in undelivered event messaging.

Specifically, the following receivers need to be declared in the manifest:

1) The service starter derived class, which will receive broadcasts relating to SDK specific events or operating system events and ensure the download service is started in order to respond. There are a fixed set of broadcast intents which must be declared in the intent filter for this component.
2) A notification receiver, which can optionally receive notification update messages depending on the chosen approach to foreground service notifications. Also optionally available are broadcasts relating to analytics events which are stored within the SDK and transmitted to the backplane. These intent actions are designed to provide a method of integration into other analytics systems or data-collection. They do not contain the same payloads as the main notification broadcasts and are not intended to be used for updating the application UI.

Asset Identifiers

Upon creation, each asset is assigned an internal identifier for the SDK, local to the device, and also a UUID which is used in analytics reporting to the backplane. Each asset creation method also contains a parameter to store an external string identifier which maps the asset to your catalog. The external identifier is required, and it is recommended that it should be unique to each asset, however the SDK does not enforce this requirement. If duplicate assets have been created with the same external identifier then retrieving an asset via that identifier will only return the first asset created. The backplane uses the external identifier to implement business logic rules, so non-unique identifiers will result in assets being treated as duplicates.

Database Versioning

The SDK uses an internal database to manage all the asset data. The database is versioned and often changes between releases to support new features. The SDK does not support downgrading the database as this would result in removing support for features that may have been used while downloading the assets. Managing the potential issues in feature downgrade in a production application would be difficult as it would depend on the treatment by the client application, therefore the safest approach is to not provide any path for downgrade. In turn, this means the SDK itself cannot be downgraded within a production application.

In order to catch the situation during app development where an application with an old SDK is deployed onto a device which has already a later version, we allow the Content Provider to throw the standard exception for a failed downgrade when it detects an older database version being installed over a newer one. This prevents the application from running, highlights the issue to developers, and prevents them from spending time investigating unexpected behaviour. If the SDK version is downgraded during development then the application will need to be uninstalled from the device and reinstalled to ensure consistent operation.

Security for Playback of Segmented Assets via localhost

Stored assets are delivered to the player via a localhost connection on 127.0.0.1 so that the

player is able to treat the offline content exactly the same as it would the original url. An HTTP proxy server is run within the SDK to deliver this content. Delivering content over HTTPS using this server would use additional processing resource and require the player to validate a self certification, therefore the proxy delivers content over unsecured HTTP. The proxy is designed to only deliver content to localhost, so it cannot be used to stream the assets outside of the device.

From Android API 28, cleartext network support is disabled by default in order to improve app security. The following network security configuration will need to be added to the application in order to allow local network connections over HTTP for playback:

Android Manifest:

```
<application
        …
        android:networkSecurityConfig="@xml/network_security_config"
>
```

res/xml/network_security_config.xml:

```
<network-security-config>
        <domain-config cleartextTrafficPermitted="true">
          <domain includeSubdomains="true">127.0.0.1</domain>
        </domain-config>
</network-security-config>
```

AndroidX

The SDK now uses AndroidX dependencies, it will no longer build against apps using support libraries and the android.arch package, and no longer requires Jetifier if building against an AndroidX application.

WorkManager Time Based Scheduling

The SDK uses the android WorkManager for scheduling timed actions such as expiration. It is declared using the androidx.work package and is currently built against v2.2.

WorkManager is started automatically using an initializer class which is defined as a content provider and added to the component manifest to ensure the manager is started early in the app lifecycle. The current Google implementation of WorkManager is not process safe for operation across multiple processes within an application, and the default manifest declaration sets multiprocess=true to protect against this. The SDK is designed with the assumption that all WorkManager activities should take place within the primary process of the application, such that any customisation applied to the WorkManager will apply to all jobs to be processed. The SDK therefore directs all timed activities into the primary process via a content provider.

The SDK, by default, declares a specialized WorkManagerInitializer content provider class to initialize the WorkManager and perform the SDK specific actions. It uses the default configuration of the WorkManager, but the manifest entry in the SDK sets the multiprocess flag false. The default WorkManagerInitializer is disabled in the SDK manifest to avoid unintentional changes to the default implementation flags without consideration.

If a specialized version of the WorkManager is desired then it will need to be declared via a new Content Provider class, and not in an Activity or Service. If such a content provider starts the workmanager then the SDK initializer will use the running Work Manager. Content providers are initialized in an order which may be specified in a manifest, or otherwise is determined by the operating system. If the application requires a workmanager initialisation with different parameters then a new Content Provider must be defined and the initOrder attributes set to ensure that it starts before the SDK initializer.

Lines as below in the application manifest can be used to set the SDK initOrder to the value of

your choice, in this case 2:

```
<provider
      android:name=”com.penthera.virtuososdk.data.imple.provider.Virtuos
oWorkManagerInitializer”
      android:authorities=”${applicationId}.VirtuosoWorkManager”
      android:initOrder=”2”
 />
```

A provider with a higher initOrder will be started first, for instance:

```
<provider
      android:name=”com.example.MyWorkManagerInitializer”
      android:authorities=”${applicationId}.MyWorkManager”
      android:initOrder=”10”
 />
```

Let's Get Started

We provide details below of all the components which need to be setup to make the SDK work within your application. We walk through all of the steps that will be required for the SDK to function fully. A summarized list of these steps can be found in Appendix B, to be used as a checklist during integration of the SDK.

Unpacking

We'll provide you details of a public github repository where you can access the Android developer package. The contents of the deliverable are:

- **CnCDemo**: Android Studio project - contains source code for demo app that uses the SDK
- **javadocs**

SdkDemo: A Reference Client

SdkDemo is a reference standalone Android app written in Java that uses the SDK. We provide this as a convenience so you can see how to call the SDK to enqueue, configure, download, and play video. It uses public-domain videos (HLS, HSS, and mp4), hosted by Penthera on Amazon AWS.

To build and run SdkDemo:

1. In Android Studio Select `File-New-Import Project`
2. In the Select Project window navigate to the `CnCDemo` folder of the deliverable and select the `build.gradle` file.
3. Run or Debug the SdkDemo module in an emulator or on a device.
4. You will need a public and private key to build the application. Please contact support@penthera.com to get them. The keys must be entered in the commented lines of com.penthera.sdkdemo.Config.java
5. You will need to provide a `google-services.json` and an `api_key.txt` to use Firebase Cloud Messaging and/or Amazon Device Messaging. There are placeholder files which need to be populated or removed. If not using Push Notifications then remove the files and comment out or remove the relevant sections from the Android Manifest and the gradle build. These sections are detailed here: Setting up Push Notifications

Congratulations! You've now got a video-downloading app up and running. You're ready to develop your own apps with the SDK.

Note: The SDK Demo uses ExoPlayer it will only play videos on Kindle Devices running Fire OS 5 or greater. If you need to support earlier Fire OS versions then a different player will need to be used. Penthera will be happy to help you with an implementation if needed.

Building Your Own App

Either create a new project within Android Studio or use an existing one.
In the `build.gradle` file make sure you include the following snippet:

```
allprojects {
   repositories {
       jcenter()
       maven {
           url
'http://clientbuilds.penthera.com:8081/repository/releases/'
       }
   }
}
```

This will allow Android Archives to be read from the Penthera hosted Maven repository

Add the following implementation statement to the dependencies list in your applications `build.gradle`:

```
implementation ('com.penthera:cnc-android-sdk:3.15.14')
```

Sync the project and you are now ready to start using the Download2Go SDK.

To use a copy of the library which defaults to debug logging, which is best during development, use the following implementation statement:

```
implementation ('com.penthera:cnc-android-sdk-debug:3.15.14')
```

Step by step instructions to create all the necessary components to work with the Download2Go SDK can be found in Appendix B.

Permissions used by the SDK

The SDK uses the following permissions:

- **android.permission.INTERNET:** Required to access the network
- **android.permission.ACCESS_NETWORK_STATE:** Required to determine the state of the network, so the SDK can respond to network loss and return appropriately.
- **android.permission.ACCESS_WIFI_STATE:** Required along with network state to react to network changes appropriately.
- **android.permission.CHANGE_WIFI_STATE:** Needed to refresh the wifi connection if/when it gets stale. Known Android issue in older devices. Allows for robust downloading.
- **android.permission.FOREGROUND_SERVICE:** Needed for the downloader service to be able to run as a foreground service since Android API 28. Required for background downloading.
- **android.permission.RECEIVE_BOOT_COMPLETED:** Required to allow the SDK to resume downloads after a device reboot without user interaction.
- **android.permission.READ_EXTERNAL_STORAGE / android.permission.WRITE_EXTERNAL_STORAGE:** Limited to Android SDK API version 18 or less if using default app storage. API versions greater than 18 do not require this permission unless you plan to change the base storage folder to one outside

of the app protected space. The permission is required to write downloads to disk. If you choose to use this permission then you must handle dynamic permission requests.

Penthera also recommends using the **android.permission.QUICKBOOT_POWERON** permission. This permission allows the SDK to restart and resume downloads after a quick boot without user interaction. i.e. This occurs when the device is not fully powered off but all services and applications are killed before it turns back on.

Content Provider and App Identification

The SDK requires a derived Content Provider to be created in order to define the Content Provider Authority for your app. The Authority is used as the primary identifier for the app to ensure that multiple instances of the download service work with the appropriate database for various purposes, including to send out broadcasts and perform local database functions.

The SDK needs details of the Authority of your app's content provider at startup. You provide the SDK this information by including a name/value pair in AndroidManifest.xml:

```
<meta-data android:name="com.penthera.virtuososdk.client.pckg"
           android:value="com.demo.myvirtuosoapp.identifier" />
```

Instantiation

`Virtuoso` has one constructor which takes the current application context. Virtuoso can be instantiated within any component where a handle on the context is available. Generally, each Activity has one instance of Virtuoso, but you can use multiple instances or a singleton if required.

If a singleton instance is going to be used within the application, and stored in an Application derived class, then it is recommended to lazy load the Virtuoso object within the first Activity. The first reason for this is that when the Virtuoso object is created it performs checks on the service and starts integrity checks on any stored content, so this can worsen the performance of app startup if it occurs prior to construction of the first activity. The second, and most important reason, to not create the object within the Application constructor is the multi-process nature of the SDK. It is important to consider that the SDK is designed to use a background download service within a separate process, and that an Application class is created within each process of an application. Therefore, if you create a Virtuoso object within the Application it will result in creation of one client side object within each process, including within the download service. This will result in multiple sets of checks on the service and multiple sets of integrity checks on the content plus a larger memory footprint for the app.

To initialize `Virtuoso` in the `onCreate()` of an Activity:

```
private Virtuoso mVirtuoso;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    mVirtuoso = new Virtuoso(getApplicationContext());
}
```

OnResume

You should call the SDK's `onResume` and `onPause` methods at the respective places within an Activity or application in order to ensure that the client SDK is paused when the UI is backgrounded.

Calling `onResume()` ensures that any service observers are correctly linked to the relative service callbacks. It is in the Activity's `onResume` that you'll register observers with the `Virtuoso` instance:

```
@Override
protected void onResume() {
        super.onResume();
        mVirtuoso.onResume();
        mVirtuoso.addObserver(mBackplaneObserver);
        mVirtuoso.addObserver(mEngineObserver);
        mVirtuoso.addObserver(mQueueObserver);
}
```

onPause unregisters all observers, so you will need to re-register observers in onResume() or an equivalent place in your application.

Beware that the SDK onResume tries to start the service which runs the HTTP proxy for playback, and this service can only be started when the application is in the foreground. If onResume is called at the wrong time and the app is in a background state then an IllegalStateException will be thrown. If the client implementation has any risk of calling onResume at a time when the app is in the background then it is recommended to wrap the call within a try/catch block.

Further to this, on Android 9.0 there is a known issue in the application framework where the Activity onResume method can be called by the OS prior to the app being put into foreground mode. This will result in the IllegalStateException. It is important to wrap the contents of your onResume in a try/catch block to protect against this occurrence. The issue is fixed in later android versions. I can be tracked here: https://issuetracker.google.com/issues/113122354

In the event that an exception is thrown while trying to start the service for playback, the SDK will try again when a playback URI is requested.

OnPause

Call `onPause` to allow the `Virtuoso` instance to unregister any observers it has linked to the background service. Although not necessary, it's good practice to unregister any of your own observers before calling `onPause()`:

```
@Override
protected void onPause() {
        super.onPause();
        mVirtuoso.removeObserver(mEngineObserver);
        mVirtuoso.removeObserver(mBackplaneObserver);
        mVirtuoso.removeObserver(mQueueObserver);
        mVirtuoso.onPause();
}
```

Registering with the Backplane

Before it can perform any meaningful task (download, process events, receive subscription notifications), your app must register with the Download2Go Backplane. For building a proof of concept app, you are welcome to use the shared Penthera-hosted Backplane instance; ask Penthera for the URL and credentials.

Typically you'll want to carry out this initialization within a splash activity.

To register with the Backplane, an app must call `startup`, with the following parameters:

- `aBackplaneUrl <String>`: URL of the Backplane
- aUser <String>: User identifier (which you assign) to be registered on the Backplane. This identifier is limited to 512 characters.
- `aExternalDeviceId <String>`: Optional device ID defined by your app
- `aPublicKey <String>`: Key used to identify the app on the Backplane (provided to you by Penthera)
- `aPrivateKey <String>`: Key used to sign all communications between the SDK and Backplane (provided to you by Penthera)
- `aPushRegistrationObserver <IPushRegistrationObserver>`: Listener to monitor if the registration for push messages through FCM or ADM messaging was successful. Refer to the 'Subscriptions' section, below.

Success/failure of the registration is reported through an `IBackplaneObserver` with a callback type of `BackplaneCallbackType.REGISTER`. (`IBackplaneObserver` is described below).

Once an installed app has successfully registered with the Backplane, it doesn't need to call `startup` again. If the installed app calls startup with the same user and keys then it is a NOOP. If the keys change but not the user then the SDK will revalidate with the backplane. You can check your app's authentication status via `getAuthenticationStatus`:

```
private Virtuoso mVirtuoso;
public void onLoginClick() {

  IBackplane backplane = mVirtuoso.getBackplane();
  if (backplane.getAuthenticationStatus() ==
       Common.AuthenticationStatus.NOT_AUTHENTICATED) {
         // handle user login
         // would need to listen for the registration success through
         // an IBackplaneObserver - see details below

         mVirtuoso.startup("https://backplane.server.com",
                           "APPLICATION_USER",
                           null, // could provide a device identifier here
                           "MY APPLICATION PUBLIC KEY",
                           "MY APPLICATION PRIVATE KEY",
                           new IPushRegistrationObserver(){.....}
                          );
       }
       else { /* proceed to main activity */ }
}
```

NOTE: If after you are authenticated with the backplane, you call startup with a different user then the SDK is reset. I.e.: All media for the old user is deleted and all the settings are reset to the defaults.

Service Observers

The SDK provides several observers for use within your Activity. The observers allow your application to get updates from the SDK and keep your views refreshed. See the SdkDemo source code for example source code, and the Javadocs describe their callbacks.

Observer	Description
`IBackplaneObserver`	Notifies observer when any communication with the Backplane has completed. Provides a callback which informs the client the type of communication that occurred and the result.
`IEngineObserver`	Notifies observer of changes in the download engine status, changes to the settings used by the service, and when assets are deleted or expired.
`IQueueObserver`	Notifies observer on all callbacks which affect the queue or assets within it. Notifies when downloads start and end, a downloaded error occurs, and any queue changes.
`ISubscriptionObserver`	Notifies observer when communication with the Backplane Subscription Service completes.
`ISegmentedAssetFromParserObserver`	Notifies observer of actions when a segmented asset manifest is parsed and the asset created.
`IPushRegistrationObserver`	Notifies observer when Push Messaging registration takes place.

In addition to the Observer interfaces, the SDK provides a base class implementation of the `IEngineObserver, IQueueObserver and ISubscriptionObserver` interfaces. The base implementation of each interface method does nothing.

Interacting with the Background Service

Some of the APIs require a bound connection to the background service. These are provided in the `IService` interface. The `IService` interface obviates the need to maintain a persistent connection; it allows you to bind and unbind from the service as and when needed.

The `IService` interface allows you to:

- pause/resume downloads
- retrieve the current status
- retrieve network throughput data

```
private int mServiceStatus;
private Virtuoso mVirtuoso;
private IService mService;

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ...
    mVirtuoso = new Virtuoso(getApplicationContext());
     mService = mVirtuoso.getService();
}

protected void onResume() {
     super.onResume();
     if(mService.bind())
```

```
        mServiceStatus = mService.getStatus();
}

protected void onPause() {
     super.onPause();
     mService.unbind();
}
```

Setting up Push Notifications

To receive Push Notifications either through Amazon Device Messaging or Firebase Cloud Messaging you will need to add the push notification Receivers and Services to your application.

For supporting FCM you will need to provide the `google-services.json` file obtained from the Firebase console. If supporting ADM then you will need to provide an `api_key.txt` which contains the ADM key supplied in the Amazon developer console.

Instructions for setting up FCM can be found here: Manually add Firebase
Instructions for setting up ADM can be found here: Obtaining ADM Credentials and Store Your API Key as an Asset

TIP : Not all Android devices have GooglePlay Services on the ROM. You may want to check within your app and prompt the user to install it. We suggest you do so within the `onServiceAvailabilityResponse` response of the `IPushRegistrationListener`. See the SDK Demo for details on how to do this, or visit Check for Google Play services

For the SDK to function correctly with FCM and ADM you must use the Push Notification services and receivers provided in the SDK. You will need to declare these in the manifest as shown below. If you need to have access to the token or will be handling messages other than those needed by the SDK then you can subclass the SDK classes (remember to always call `super.<METHOD>` on any methods which are overridden), The SDK Demo application shows how to subclass the SDK classes and declare them in the manifest..

Use the `IPushRegistrationObserver` interface at `startup` to monitor for success/failure when registering with the Push Services..

Setting up DRM

To use the DRM solution provided within the SDK for Widevine DRM, you will need to derive from the LicenseManager class and provide methods to make requests to the server and retrieve the license keys from the results. This is necessary because many Widevine license server implementations wrap the plain keys within JSON or XML response messages which also provide additional information about the license details or expiry. The LicenseServer class has methods to be overridden to specify the URL to request for an asset and to process the response and extract the keys. Details to set this up are provided below.

The SDK can manage licenses for the application and request renewed licenses for offline playback when the device is online and prior to license expiry. Settings are available to disable this automated update behaviour and manually request license updates when the application requires them. If you do not wish the SDK to process DRM at all then it is possible to disable DRM license management and not provide a license manager implementation to the SDK.

Registering Manifest Parser Observer

Manifest parsing runs in the SDK download service, which is isolated in a different process from the main client application. Basic observer interfaces are available within the API that can be registered from the client process using the IAssetManager interface, either during asset creation or after creation and prior to parsing completion. These will provide enough functionality for most applications to be informed when queue time permissions responses are received and when the asset manifest parse completes. If the application is required to alter individual segment URLs prior to the downloader starting, or to alter the asset properties prior to the asset being added to the download queue, then a second interface is available that works within the service process.

Segmented Asset Observer

The segmented asset observer enables the application to be notified when an asset has been created from a manifest. The `ISegmentedAssetFromParser` observer interface can be instantiated and passed either in creation of the asset or via `IAssetManager::addParserObserverForAsset()`. The second approach will only register the observer if the asset parsing is not complete and has not reached the download pending state. The interface has a single method `complete()` which will be called back to indicate the state of the asset when parsing completes, and if it has been added to the download queue. If no other assets were awaiting download then the asset download may have started before this callback occurs.

Queue Permission Observer

The queue permission observer enables the application to be notified when an asset has received a permission response from the Backplane while being added to the queue. If you add a queue permission observer then the asset will have permissions checked at the time of queuing instead of at the time when download will start. This is equivalent to setting “Require Permissions on Queue” in the backplane settings so only add this observer if you intend that style of permissions management. The `IQueue.IQueuedAssetPermissionObserver` interface can be instantiated and passed either in creation of the asset or via `IAssetManager::addPermissionsObserverForAsset()`. The second approach will only register the observer if the asset parsing is not complete and the permissions request has not been processed. The interface has a single method `onQueuedWithAssetPermission()` which will be called back to indicate if the asset has been queued and provide the permissions response. This callback may occur prior to the manifest parse being completed as the asset is queued during the parsing process. If no other assets were awaiting download then the asset download may be starting at the same time that this callback occurs.

Implementing IManifestParserObserver

Modifying individual segment URLs, or modifying other asset parameters before the downloader begins to process the asset, requires a slightly different approach to simply observing when those events occur. To do this requires registering a component with the SDK which will be instantiated from within the download service, within the service process.

There are three steps to registering a manifest parser observer which can modify segment URLs or the asset prior to queuing:

1) Implement the IBackgroundProcessingManager interface. This is a factory interface that will be called from within the SDK to create the manifest parser observer. It is also responsible for other background processes such as subscriptions. Use the `getManifestParserObserver()` method to return an observer. This method will be called separately for each asset that is parsed.
2) Register the background processing manager implementation with the SDK. The SDK will create an instance of this class, if registered, in order to fetch the manifest parser observer. The implementation must be declared in the Android manifest using a meta

data entry with the name “com.penthera.virtuososdk.background.manager.impl” and value matching the class name.

For instance, if the class is defined:

```
package com.yourcompany;

class MyExampleBackgroundProcessingManager implements
IBackgroundProcessingManger{}
```

Then a matching entry is needed in the Android manifest:

```
<meta-data
android:name="com.penthera.virtuososdk.background.manager.impl"
        android:value="com.yourcompany.MyExampleBackgroundProcessing
        Manager" />
```

Note that the class will also need to be protected from Proguard obfuscation.

3) Implement the IManifestParserObserver interface, using `didParseSegment()` to alter the segment URLs or `willAddToQueue()` to alter parameters of the asset such as start window, end window, or download limits.

It is important to consider that implementations of IBackgroundProcessingManager and IManifestParserObserver are running in a separate android process to the application UI and cannot directly share any memory based resources with the main application code. For instance, if you create a singleton in the service process, it will not be the same singleton instance which is in the main application process. Communication between the two will require using Android Inter Process Communication methods, or the component will need to make its own network requests to fetch the necessary information to achieve its purpose. The download service process can be running within a foreground service when the application is not running, so this may also be something to consider during implementation.

Modify AndroidManifest.XML

Model your Android manifest after the following:

```
<!-- Include the amazon namespace if using ADM -->
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  xmlns:amazon="http://schemas.amazon.com/apk/res/android"
 />
      ...
 <!-- This permission ensures no other application can intercept your
ADM messages. -->
<permission
  android:name="com.penthera.harness.permission.RECEIVE_ADM_MESSAGE"
  android:protectionLevel="signature" />

<uses-permission android:name=
 "<YOUR  PACKAGE>.permission.RECEIVE_ADM_MESSAGE" />

<!-- This permission allows your app access to receive push
```

```
notifications from ADM. -->
<uses-permission android:name=
"com.amazon.device.messaging.permission.RECEIVE" />


<application

      ...
      android:networkSecurityConfig="@xml/network_security_config"
>
      ...

    <!-- Add Google play services version -->
   <meta-data android:name=
   "com.google.android.gms.version"
              android:value="@integer/google_play_services_version" />

    <!-- Add client package name for content provider Authority so the
client can use the content provider -->
   <meta-data android:name="com.penthera.virtuososdk.client.pckg"
           android:value="com.yourcompany.app.auth" />

    <!-- Add the license manager class name to use DRM -->
    <meta-data tools:replace="android:value"
android:name="com.penthera.virtuososdk.license.manager.impl"
android:value="com.yourcompany.app.drm.LicenseManager"/>


      ...

<!-- Download Service in separate process -->
<service android:name="com.penthera.virtuososdk.service.VirtuosoService"
  android:process=":service"
  android:label="VirtuosoService">
</service>

<!-- HTTP Proxy service for playback -->
<service
android:name="com.penthera.virtuososdk.service.VirtuosoClientHTTPService"
  android:label="VirtuosoClientHTTPService"
  android:enabled="true">
</service>

<!-- Service Starter -->
<receiver android:name="com.yourcompany.app.YourServiceStarter"
  android:enabled="true"
  android:process=":starter"
  android:directBootAware="true">
  <intent-filter>
     <action android:name="android.intent.action.BOOT_COMPLETED"/>
     <action android:name="android.intent.action.QUICKBOOT_POWERON"/>
     <action android:name="com.htc.intent.action.QUICKBOOT_POWERON"/>
     <action
android:name="virtuoso.intent.action.SERVICE_NOTIFICATION_UPDATE" />
     <action android:name="virtuoso.intent.action.DOWNLOAD_UPDATE"/>
     <action
android:name="virtuoso.intent.action.CONFIGURE_VIRTUOSO_SERVICE_LOGGING"
/>
```

```
      <action android:name="virtuoso.intent.action.BACKPLANE_SYNC_DEVICE"
/>
      <action android:name="android.intent.action.PACKAGE_REMOVED" />
      <action android:name="virtuoso.intent.action.START_SERVICE" />
      <data android:scheme="package"/>
  </intent-filter>
  <intent-filter>
      <action android:name="android.intent.action.BOOT_COMPLETED"/>
      <action android:name="virtuoso.intent.action.START_SERVICE" />
      <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>

</receiver>

<!-- Your Content Provider -->
<provider android:name="com.yourcompany.app.YourContentProvider"
  android:authorities="com.yourcompany.app.auth"
  android:process=":provider"/>

<!-- Notification Receiver -->
<receiver
android:name="com.penthera.sdkdemo.notification.NotificationReceiver"
       android:enabled="true"
       android:process=":receiver"
       android:label="NotificationReceiver">
       <intent-filter>
          <action
android:name="com.yourcompany.app.auth.NOTIFICATION_DOWNLOAD_START"/>
          <action
android:name="com.yourcompany.app.auth.NOTIFICATION_DOWNLOAD_STOPPED"/>
          <action
android:name="com.yourcompany.app.auth.NOTIFICATION_DOWNLOAD_COMPLETE"/>
          <action
android:name="com.yourcompany.app.auth.NOTIFICATION_DOWNLOAD_UPDATE"/>
          <action
android:name="com.yourcompany.app.auth.NOTIFICATION_DOWNLOADS_PAUSED" />
          <action
android:name="com.yourcompany.app.auth.NOTIFICATION_MANIFEST_PARSE_FAILE
D" />
       </intent-filter>
</receiver>


<!-- SDK Component that handles FCM Registration tokens -->
<service
  android:name="com.yourcompany.app.push.FCMInstanceIdService"
  android:exported="false">
  <intent-filter>
     <action android:name="com.google.firebase.INSTANCE_ID_EVENT" />
  </intent-filter>
</service>


<!-- SDK Component that Handles FCM messages -->
<service
  android:name="com.yourcompany.app.push.FCMService">
  <intent-filter>
     <action android:name="com.google.firebase.MESSAGING_EVENT" />
```

```
  </intent-filter>
</service>


<!-- Enable amazon device messaging -->
<amazon:enable-feature
  android:name="com.amazon.device.messaging"
  android:required="false"/>


<!-- SDK Component that Handles ADM messages -->
<service
  android:name="com.yourcompany.app.push.ADMService"
  android:exported="false" />


<!-- SDK Component that Receives ADM client broadcasts -->
<receiver
  android:name="com.yourcompany.app.push.ADMService$DemoADMReceiver"
  android:permission="com.amazon.device.messaging.permission.SEND" >

  <!-- To interact with ADM, your app must listen for the following
intents. -->
  <intent-filter>
     <action
android:name="com.amazon.device.messaging.intent.REGISTRATION" />
     <action android:name="com.amazon.device.messaging.intent.RECEIVE" /
>

     <!-- Replace the name in the category tag with your app's package
name. -->
     <category android:name="com.yourcompany.app" />
  </intent-filter>
</receiver>



   </application>
</manifest>
```

Proguard

It is always recommended to use Proguard/R8 on your release builds, and the SDK includes a packaged proguard rules file to help with this. The SDK is built using a very simple Proguard configuration that does not obfuscate many class names or variables, especially none of those within the public API. This means that crash reports generated with an un-obfuscated test build will contain full original class names. This is done in order to make debugging and bug reporting easy for client developers. The SDK also includes all functionality offered by the Download2Go product in a single library, even though some applications will only use a subset of the features. It is therefore very important to apply a more aggressive ruleset to the release build to improve the final application size.

The rules file shipped with the SDK contains the minimal set of rules to protect the operations of the SDK under Proguard. These rules are primarily based around places where observer classes are registered via the Android manifest or used in broadcasts. The rules file will be included in your build by default if using Gradle, otherwise it can be found within the aar file, named proguard.txt .

Penthera does not explicitly silence any Proguard warnings. It's up to you to add appropriate Proguard configuration for things you don't want to warn about yourself. Some of the dependencies in the Penthera library may produce Proguard warnings and are not errors. You will need to determine any such cases on a per-application basis. Penthera previously shipped a Proguard configuration file which hid some warnings from components, such as OkHttp, which historically caused build problems. It also included a number of clauses to hide warnings from java annotation classes. These have been removed upon request. We do not envisage any impact from these changes on most client builds, but this will need to be checked when updating from versions of the SDK prior to 3.15.12 as the previously shipped warnings may potentially have masked missing clauses in the application proguard file.

Common Functions

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

Each asset creation method contains a parameter to store a string identifier which maps the asset to your catalog. These identifiers are required and must be unique to each asset. The backplane uses them to implement business logic rules, so non-unique identifiers will result in assets being treated as duplicates.

Enqueue an Asset

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

Use this method to enqueue a single file for download:

```
IAssetManager assetManager = mVirtuoso.getAssetManager();
IFile vi = assetManager.createFileAsset (
    "http://some.server.com/media.mp4", // remote URL
    "MY_CATALOG_IDENTIFIER",            // An asset identifier your app can use to
                                       // map this asset to your catalog. Must be
                                    // unique for each tracked asset.
    "video/mp4",                        // asset mime type, for validation
    "{
        // Additional metadata that SDK should store with the asset
        \"title\": \"media title\",
        \"desc\":\"media description\",
        \"img\":\"http://myimage.png\"
     }");

assetManager.getQueue().add(vi);
```

Enqueue a segmented video, specifying a target bitrate

Use these methods (provided through the `IAssetManager` interface) to enqueue a video that's split into multiple segments (e.g. an HLS,HSS or MPEG-DASH video:

- `createHLSSegmentedAsset` or `createHLSSegmentedAssetAsync`
- `createHSSSegmentedAsset` or `createHSSSegmentedAssetAsync`
- `createMPDSegmentedAsset` or `createMPDSegmentedAssetAsync`

You'll provide the URL of the manifest, and a max desired bitrate. The SDK will parse the manifest, identify the highest-quality profile whose bitrate doesn't exceed the specified max bitrate, and then download all the fragments belonging to that profile. The details for the new asset can be provided using an asset builder which is specific to the manifest type.

```
// This observer will receive notification when asset has been created
final ISegmentedAssetFromParserObserver observer =
  new ISegmentedAssetFromParserObserver() {
    @Override
     public void complete(ISegmentedAsset aSegmentedAsset, int aError,
                          boolean addedToQueue) {
       if (addedToQueue) {
           // success
        } else {
           // something went wrong
        }
```

```
   }
};


HLSAssetBuilder hlsAsset = new HLSAssetBuilder();
hlsAsset.assetId("MY_CATALOG_IDENTIFIER")                    // see above
        .manifestUrl("http://some.server.com/manifest.m3u8") // URL of manifest
        .downloadEncryptionKeys(true)           // Download encryption keys?
        .desiredVideoBitrate(1927853)            // max desired bitrate to use
        .assetObserver(observer)                // just-created observer; see above
        .addToQueue(true)                       // Add to download queue?
        .withMetadata("(metadata key:value bindings go here)");  // see above

IAssetManager assetManager = mVirtuoso.getAssetManager();
assetManager.createHLSSegmentedAssetAsync( hlsAsset.build());
```

Supplying Integer.MAX_VALUE for the max bitrate parameter tells the SDK to select the highest-bitrate profile available. Supplying 1 for the max bitrate parameter tells the SDK to select the lowest profile available.

Enqueue a Segmented Asset Manually from the Segments

You can "do-it-yourself" and manually enqueue the individual video fragments:

```
// create list of segment URLs
ArrayList<String> segments = new ArrayList<String>();
segments.add("http://some.server.com/media/low-profile/frag0.ts"):
segments.add("http://some.server.com/media/low-profile/frag1.ts")):
segments.add("http://some.server.com/media/low-profile/frag2.ts")):

// instantiate the HLS segmented asset
IAssetManager assetManager = mVirtuoso.getAssetManager();
ISegmentedAsset hls =
    assetManager.createSegmentedAsset("MY_CATALOG_IDENTIFIER",
                              "(metadata key:value bindings go here)" );

// add segments
hls.addSegments(segments);

// enqueue for download
assetManager.getQueue().add(hls);
```

Access Downloaded/Queued Assets

Retrieve a cursor on the assets previously downloaded and now stored on the device:

```
IAssetProvider downloaded = mAssetManager.getDownloaded()
Cursor c = downloaded.getCursor();
```

Retrieve a cursor on the assets in the download queue:

```
IQueue queue = mAssetManager.getQueue()
Cursor c =  queue.getCursor();
```

Remove an Asset

Delete an asset that is either enqueued or already downloaded:

```
IIdentifier vi = mAssetManager.get(uuidString);
mAssetManager.delete(vi.getId());
```

Flush Queue

Removes all assets from the download queue:

```
mAssetManager.getQueue().flush();
```

Expire an Asset

Manually mark an asset as having expired:

```
IIdentifier vi = mAssetManager.get(uuidString);
mAssetManager.expire(vi.getId());
```

Notes:

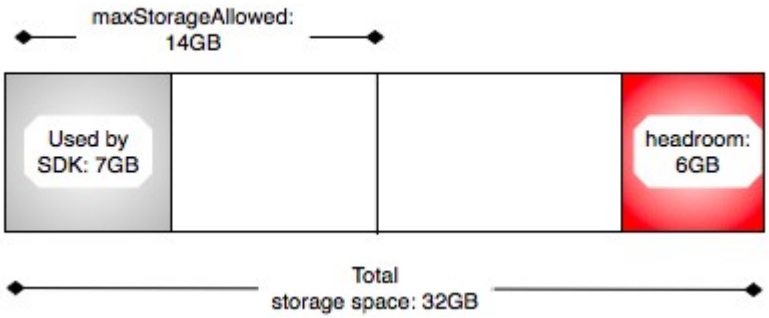
- The SDK automatically removes expired assets from the download queue.
- The SDK provides access to expired assets through the `IAssetManager`.
- When an asset expires, the SDK deletes the data associated with the asset (e.g. the mp4 file or the .ts files), to free disk space. However, the SDK retains the asset's metadata. This allows your app to access information about the expired asset.
- The SDK will automatically track and mark expiry if appropriate metadata has been supplied, such as the expiry timestamp, expiry after download, or expiry after play values.

Configure Download Rules

The SDK obeys several behavioral settings. You can access and configure these settings through the `ISettings` interface. *Notice that the default values are very conservative; for most apps, you'll want to tune these to more aggressive values.*

- **`headroom:`** Storage capacity that the SDK will leave available on the device. If there's less than this amount of free space on the device, the SDK won't download (default: 100MB)

- **`maxStorageAllowed:`** Maximum disk space the SDK will ever use on the device. If the SDK is storing this or more downloaded data on the device, it won't download (default 100MB).



Visualizing maxStorageAllowed and headroom parameters. In this scenario, the device has 32GB of disk space. The Engine will always preserve 6GB of free space on disk (i.e. "headroom"). Currently, the Engine is using 7GB, and may never use more than 14GB total (i.e. "maxStorageAlllowed").

- **`batteryThreshold`**: fractional battery charge level below which SDK suspends downloading. A value of 0 (completely discharged) means "no limit." A value greater than 1 (completely charged) means "only download when charging." (default: 1)

- **`cellularDataQuota`**: MB/month the SDK can download over cellular. A value of 0 means "don't download any bytes over cellular." A negative value indicates an unlimited quota. (default: 0). The SDK divides this number into four and enforces the smaller number on a week-to-week basis.

- **`destinationPath`:** commonly this is an additional relative path added to the SDK root download location. The SDK will store all downloaded content here. By default, the SDK stores all downloads in the the SDK root directory, `/virtuoso/media/`, under appropriate sub-directories. If this destination path is defined to an absolute path then the SDK will try to download all content to that path. The app is responsible for ensuring external write permission is available, the download engine will report back a blocked state if it cannot write due to permissions.

Retrieve a setting:

```
ISettings settings = mVirtuoso.getSettings();
long maxStorage = settings.getMaxStorageAllowed();
```

Override settings:

```
settings.setMaxStorageAllowed(1024)
        .setHeadroom(200)
        .setBatteryThreshold(0.5)
        .save();
```

Reset a setting to the SDK default or to that provided by a Backplane:

```
settings.resetMaxStorageAllowed().save();
```

Retrieve and Persist Widevine Licenses

When downloading a widevine protected MPEG-DASH asset the SDK will attempt to download and persist the license. In order to retrieve the license it needs to request it from the licensing server, the licensing server url may need to be formatted differently depending on the asset. To provide the correct Url for license retrieval you need to provide a License Manager that the SDK can use. The easiest way to do this is to extend the `LicenseManager` class in the `com.penthera.virtuososdk.client.drm` package and override the `getLicenseAcquistionUrl` method.
E.g.:

```
public class DemoLicenseManager extends LicenseManager {
   @Override
   public String getLicenseAcquistionUrl() {
       String license_server_url = "https://proxy.uat.widevine.com/proxy";
        /*
        Here you can examine the mAsset and mAssetId member variables and modify the
        license server url if needed:
        Example:
        String video_id = mAsset != null ? mAsset.getAssetId() :
                            mAssetId != null ? mAssetId : null;
        if(!TextUtils.isEmpty(video_id){
         license_server_url += "?video_id="+video_id + "&provider=widevine_test";
        }
        */
       return license_server_url;
   }
}
```

For the SDK to use your License Manager implementation you need to override a metadata value in the AndroidManifest.
To override metadata you will need to add the tools namespace:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools" … >
```

Now add in the metadata (replace the part in bold with the fully qualified class name of your License Manager implementation):

```
<meta-data tools:replace="android:value"
android:name="com.penthera.virtuososdk.license.manager.impl"
android:value="com.penthera.sdkdemo.drm.DemoLicenseManager"/>
```

Playout with Widevine Persisted License

The SDK provides a Drm Session Manager: `VirtuosoDrmSessionManager` it can be found in the `com.penthera.virtuososdk.client.drm` package. You will need to integrate this with your player implementation. The SdkDemo project shows how to integrate this with ExoPlayer by implementing a wrapper that implements the Exoplayer DrmSessionManager interface. You can find the wrapper in the demo project:
`com.google.android.exoplayer2.drm.DrmSessionManagerWrapper`
The `buildDrmSessionManager` method in `com.penthera.sdkdemo.exoplayer.PlayerActivity` shows how to integrate the wrapper with the ExoPlayer.

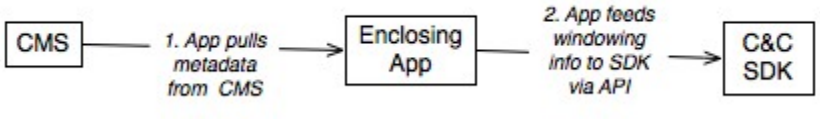
Set Availability Window for an Asset

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

Window Parameter	Description
Start Window (a.k.a. "Publish Date")	The SDK downloads the video as soon as possible, but will not make the video available through any of its APIs until after this date.
End Window (a.k.a. Expiry Date")	The SDK automatically deletes 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 automatically deletes 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 deletes 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.

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

Typically, it's a Content Management System (CMS) which stores the windowing information and communicates it to the enclosing app. The app then feeds this windowing information to the SDK. The data flows as follows:



To set the availability window for a video:

```
asset.setEndWindow(new_long_value);
asset.setStartWindow(new_long_value);
asset.setEad(new_long_value);
asset.setEap(new_long_value);
mAssetManager.update(asset);
```

To get the availability window for a video:

```
IAsset asset = (IAsset) mAssetManager.getAsset(uuidString);
long expiry_after_download = asset.getEad();
long expiry_after_play     = asset.getEap();
long start_window          = asset.getStartWindow();
long end_window            = asset.getEndWindow()
```

Enable Device for Download

The Backplane tracks which devices are enabled for download, and enforces the global "max download-enabled devices per user" parameter, which you may set via the Backplane.

The individual SDK instances "know" their own `download-enabled` status, because the Backplane communicates it to them. An SDK whose download-enabled status is false can enqueue, but will not download enqueued assets.

You can request to change the `download-enabled` status for a device as follows:

```
// create and register a backplane observer - so we know if the request succeeded
IBackplaneObserver mBackplaneObserver = new IBackplaneObserver () {
   @Override
   public void requestComplete(int callbackType, int result) {
        // only checking for download-enablement changes
        if(callbackType == BackplaneCallbackType.DOWNLOAD_ENABLEMENT_CHANGE) {
           switch(result) {
               case BackplaneResult.SUCCESS:
                      // Changed the 'downloaded-enabled' flag
                      break;
               case BackplaneResult.DOWNLOAD_LIMIT_REACHED:
                      // User has already reached quota of devices.
                      break;

               // other failure codes you may want to communicate to user
               case BackplaneResult.DEVICE_NOT_REGISTERED:/*do something*/ break;
               case BackplaneResult.INVALID_CREDENTIALS: /*do something*/ break;
               case BackplaneResult.FAILURE:           /*do something*/ break;
            }
        }
   }
```

```
};

mVirtuoso.addObserver(mBackplaneObserver);

// Relies on having an active network connection and the SDK being authenticated
// with the Backplane.
mVirtuoso.getBackplane().changeDownloadEnablement(true);  // true=enable, false=disable
```

It is also possible to enable / disable download on other devices associated with the user.

Enable / Disable Download on Other Device

Download enablement and disablement can only be done on devices associated with the user's account. The backplane manages the listing of download enabled devices 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.

```
// Relies on having an active network connection and the SDK being authenticated
// with the Backplane.
String anExternalIdToMatch = "AN_EXTERNAL_ID"
IBackplane backplane = mVirtuoso.getBackplane();
backplane.getDevices( new IBackplaneDevicesObserver (){
	@Override
	public void backplaneDevicesComplete(IBackplaneDevice[] aDevices){
		//if there is no active connection then an empty array is returned.
		for(IBackplaneDevice device : aDevices){
			//check the device
			if(anExternalIdToMatch.equals(device.externalId()){
				backplane.changeDownloadEnablement(true,device)
			}
		}
	}
});
```

Using/Configuring Broadcasts

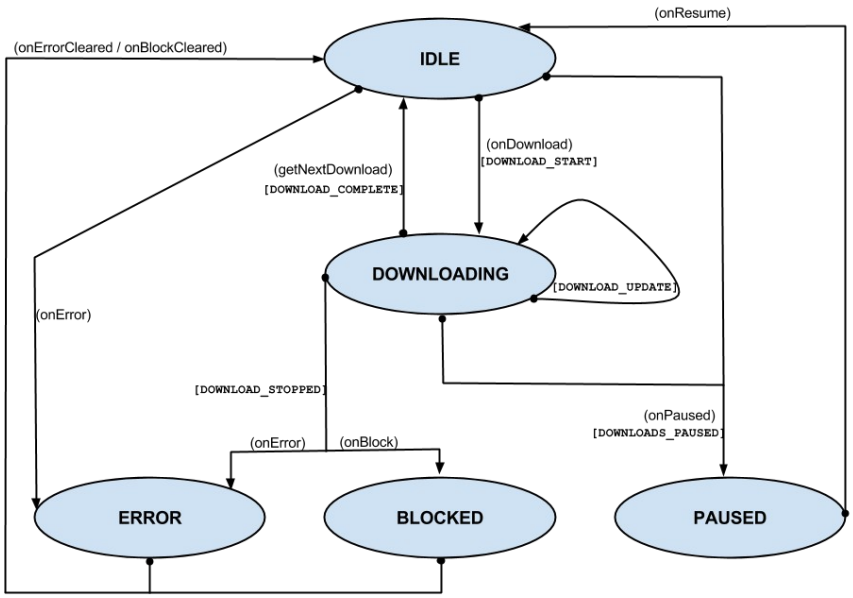
Besides issuing callbacks to the client through observers, the SDK also sends system broadcasts which you can capture with a broadcast receiver. You can use this to update your UI widgets, perform notifications, and track SDK analytics.

Notification Broadcasts

Broadcast	Description
`NOTIFICATION_DOWNLOAD_START`	A downloaded has started. The extras in the `Intent` will contain the number of assets in the queue, the asset that started downloading and the status of the download engine.
`NOTIFICATION_DOWNLOAD_STOPPED`	A download has stopped. The `Intent`'s extras will detail which asset, the number of assets in the queue and the reason for stopping.

NOTIFICATION_DOWNLOAD_COMPLETE	A download has completed. The extras in the `Intent` will detail which asset.
NOTIFICATION_DOWNLOAD_UPDATE	A progress update for a download. The Intent's extras will contain the asset and the number of assets in the queue.
NOTIFICATION_DOWNLOADS_PAUSED	A download was paused. The Intent's extras will contain the asset and the number of assets in the queue.
NOTIFICATION_MANIFEST_PARSE_FAILED	Commonly an observer is passed as an argument when a new asset is queued for download. If the app is backgrounded and cleaned up during the process so the observer cannot be notified then this notification can be used to receive notice that an asset has not been queued. The Intent's extras will contain the asset id for the asset which could not be queued.

The diagram below shows the different download states and the notification broadcasts that are generated in moving between states.



Event-Related Broadcasts

These additional broadcasts are sent to support custom or third-party analytics and directly map to log events recorded in the Backplane. The `IEvent` being reported is always available in the Intent extras at the EXTRA_NOTIFICATION_EVENT key. The event object contains additional details about the event, such as the asset ID of the video it relates to.

Broadcast	Description
EVENT_APP_LAUNCH	A fresh application launch is detected
EVENT_QUEUE_FOR_DOWNLOAD	The user has added a file to the download queue.
EVENT_ASSET_REMOVED_FROM_QUEUE	The SDK has removed a file from the download queue.
EVENT_DOWNLOAD_START	A file began to download.

EVENT_DOWNLOAD_COMPLETE	A file download completed.
EVENT_DOWNLOAD_ERROR	A file has stopped downloading due to too many errors.
EVENT_MAX_ERRORS_RESET	A file previously stopped due to download errors has been reset and will continue downloading.
EVENT_ASSET_DELETED	A file was deleted.
EVENT_ASSET_EXPIRE	An asset was determined to be expired.
EVENT_SYNC_WITH_SERVER	A sync with the Backplane completed.
EVENT_PLAY_START	A player has begun local playback of the asset.
EVENT_STREAM_PLAY_START	A player has begun streamed playback of the asset.
EVENT_PLAY_STOP	A player has stopped local playback of the asset.
EVENT_STREAM_PLAY_STOP	A player has stopped streamed playback of the asset.
EVENT_SUBSCRIBE	The user has subscribed to receive updates to a feed.
EVENT_UNSUBSCRIBE	The user has unsubscribed to receive updates to a feed.
EVENT_RESET	The SDK has handled a remote kill or detected a reinstall of the client app..

Receiving Broadcasts

See SdkDemo for an example of using notifications and events. To receive the correct broadcasts, the receiver must register an intent filter. The action names in the intent filter must be in the following format:

CLIENT_PACKAGE_IDENTIFIER + "." + BROADCAST_NAME

where CLIENT_PACKAGE_IDENTIFIER is the same value as that specified for the `com.penthera.virtuososdk.client.pckg` metadata declared in the AndroidManifest and BROADCAST_NAME is one of the broadcasts in the above table.

For example:

```
<receiver android:name="com.my.app.NotificationReceiver"
       android:enabled="true"
       android:label="NotificationReceiver"
       android:process=:"notification_service">
  <intent-filter>
     <action android:name="com.my.app.auth.NOTIFICATION_DOWNLOAD_START"/>
     <action android:name="com.my.app.auth.NOTIFICATION_DOWNLOAD_COMPLETE"/>
     <action android:name="com.my.app.auth.NOTIFICATION_DOWNLOAD_UPDATE"/>
     <action android:name="com.my.app.auth.EVENT_QUEUE_FOR_DOWNLOAD" />
     <action android:name="com.my.app.auth.EVENT_DOWNLOAD_START" />
     <action android:name="com.my.app.auth.EVENT_DOWNLOAD_COMPLETE" />
     <action android:name="com.my.app.auth.EVENT_PLAY_START" />
     <action android:name="com.my.app.auth.EVENT_PLAY_STOP" />
  </intent-filter>
```

```
</receiver>
```

Configuring Notification Broadcast Frequency

You can configure the frequency at which your app receives notifications regarding downloads through the `ISettings` interface. You can specify the update frequency on a time basis or on a percent complete basis. For Segmented Assets you can also specify updates to occur on a Segment downloaded basis.

```
ISettings settings = mVirtuoso.getSettings();
settings.setProgressUpdateByPercent(1)
         .setProgressUpdateByTime(3000)
         .setProgressUpdatesPerSegment(20)
         .save()
```

Progress Updates By Percent (Integer): min % interval at which a broadcast should be sent out. Set to 100 to disable updates based on % intervals. (default: 1)

Progress Updates By Time (Long): min # of ms that should pass before the SDK sends out a broadcast regarding updates. Set to `Long.MAX_VALUE` to disable updates based on timed intervals. (default: 5000)

Progress Updates Per Segment (Integer): min # of segments that should complete download before the SDK sends out a broadcast. (default: 10)

If multiple values are used in the configuration, the SDK will send out a Broadcast at the next interval.

The SDK guarantees that no update is sent out before the configured intervals. Broadcasts rely on Android delivery, and so you may observe a variation between the times it receives updates and the configured values.

Retrieve a setting: `settings.getProgressUpdateByPercent();`
Reset a setting: `settings.resetProgressUpdateByPercent();`

Configure Debug Logging

The SDK generates log output for debugging purposes which is configured using a custom logger and log level filtering. The SDK logger uses the common log levels of “DEBUG”, “INFO”, “WARN”, “ERROR” that map to the Android log levels of the same name, and a “CRITICAL” level, similar to Android assert that will appear as an error log in the Android logs and cannot be filtered. There is also a VERBOSE log level defined, but this is not accessible in any distributed build. The log level definitions can be found in `Common.LogLevels`.

The default log level for a debug build, which is selected in gradle by appending “-debug” to the dependency name is “DEBUG”, this is also the minimum log level accessible in the build.

The default log level for a release build is “WARN”. This provides enough debug output that problems can be remotely diagnosed if the client application uses a production logging solution. “WARN” is also the minimum log level accessible in a release build. The SDK helper will accept requests to configure lower log levels but the logs will not be generated.

The log level can be changed programmatically within the SDK if the client application wishes to reduce logging during development or disable all logging for a production release. The requested level will be distributed to all processes within the SDK.

The following line can be used to change the ongoing logging level for the SDK, which will be persisted across application executions (where context is a valid android Context):

```
Common.LogLevelHelper.updateLogLevel(Common.LogLevels.LOG_LEVEL_DEBUG,
context);
```

This command can be used for levels DEBUG, INFO, WARN ERROR and OFF. The last of these states will result in all logging other than critical log lines being removed. When the log level changes, a single log entry will indicate that the new level has been set and all further logging will be at the new level. If the client app desires no logging to be sent to the Android logger and ADB logcat then the following line should be called as soon as possible after the SDK is instantiated for the first time:

```
Common.LogLevelHelper.updateLogLevel(Common.LogLevels.LOG_LEVEL_OFF, context);
```

Using Analytics Events

The Backplane tracks and reports on a variety of SDK-related events. For a complete list of events, see the SDK documentation. You may configure the SDK to issue notifications to your application for all or some events using some predefined broadcasts (See section on Broadcast Receivers for general details). An receiver may be registered in the manifest for each analytics event type that is desired, which will result in a broadcast being received by the registered component each time the SDK generates an analytics event of that type. The demo application shows some events being registered in the Android manifest, although they are unused in the notification receiver example.

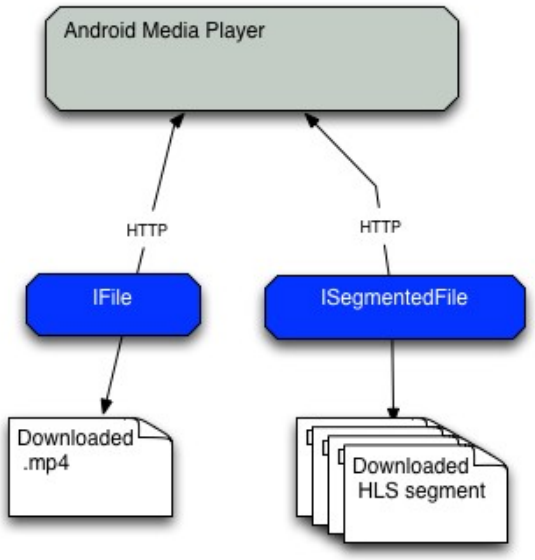
One event that can be manually controlled by the client application is the “app launch”. This event simply records when the application UI was started which caused the SDK to be instantiated. If the application does not manually generate an app launch event with 5 seconds after the SDK is started then the SDK will automatically generate the event itself.

The following can be used to send the “app launch” event to the Backplane manually:

```
Common.Events.addAppLaunchEvent(getApplicationContext());
```

Play a Downloaded Asset

The SDK doesn’t include a video player. It does, however, provide a **playout proxy,** a local HTTP proxy that sits between a media player and the downloaded videos.



The SDK offers two different approaches to video playback, depending on the type of asset. A Segmented Asset is always played through the `VirtuosoClientHTTPService`, and its URL must point to the proxy. A single File can use its path as the URL.

You can retrieve the correct path for a Segmented Asset by using the `getPlaylist` method. For a singleton file, use the `getFilePath` method.

Example of playing back assets:

```
public static void play(Context context, IAsset i) {
   Intent openIntent = new Intent(android.content.Intent.ACTION_VIEW);
   if (i.type() == Common.AssetIdentifierType.FILE_IDENTIFIER) {
      IFile f = (IFile) i;
      File file = new File(f.getFilePath());
      String mimeType = f.mimeType();
      openIntent.setDataAndType(Uri.fromFile(file), mimeType);
   } else if(i.type() == Common.AssetIdentifierType.SEGMENTED_ASSET_IDENTIFIER) {
      ISegmentedAsset f = (ISegmentedAsset) i;
      String mimeType = "video/*";
      URL pl;
      try {
           pl = f.playlist();
           openIntent.setDataAndType(Uri.parse(pl.toString()), mimeType);
      } catch (MalformedURLException e) {
             throw new RuntimeException("Not a playable file");
      }
   } else throw new RuntimeException("Not a playable file");

   // Register a 'play start' event
   Common.Events.addPlayStartEvent(context,i.getAssetId());

   // Play the Asset
   context.startActivity(openIntent);
}
```

You can also retrieve the file path and the playlist through the Cursors provided by the Asset Manager:

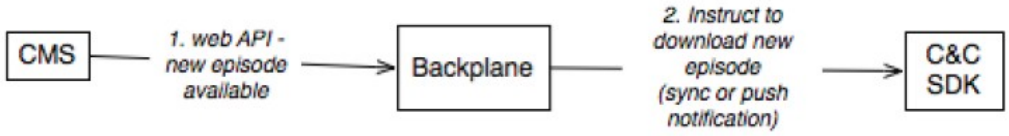
```
// Representation of a playable asset
private class MyAsset {
   private final int mId;
   private final Uri mUri;
   private final String mAssetId;
   private final String mMime;

   MyAsset(int id, Uri uri, String assetId, String mime){
     mId = id;
     mUri = uri;
     mAssetId = assetId;
     mMime = mime;
   }

   void play(Context context){
     // Register a 'play start' event
     Common.Events.addPlayStartEvent(context,mAssetId);
     Intent openIntent = new Intent(android.content.Intent.ACTION_VIEW);
     openIntent.setDataAndType(Uri.parse(pl.toString()), mimeType);
     context.startActivity(openIntent);
   }

}


List<MyAsset> myAssets = new ArrayList<MyAsset>();
Cursor c = null;
try {
    c = mAssetManager.getDownloaded()
                     .getCursor( new String[] {AssetColumns._ID
                                               ,AssetColumns.TYPE
                                               ,AssetColumns.PLAYLIST
                                               ,AssetColumns.ASSET_ID
                                               ,AssetColumns.FILE_PATH
                                               ,AssetColumns.MIME_TYPE
                                                },null,null);
    if( c != null ) {
       while (c.moveToNext()) {
           int type = c.getInt(1);
           Uri uri = null;
           String mimeType = null;
           if(type == Common.AssetIdentifierType.FILE_IDENTIFIER) {
              File file = new File(c.getString(4));
              uri = Uri.fromFile(file);
              mimeType = c.getString(5);
              if(TextUtils.isEmpty(mimeType)) {
                 String ext =
                 android.webkit.MimeTypeMap.getFileExtensionFromUrl(uri.toString());
                 mimeType =
                 android.webkit.MimeTypeMap.getSingleton().getMimeTypeFromExtension(ext)
              }
           }
           else if (type == Common.AssetIdentifierType.SEGMENTED_ASSET_IDENTIFIER) {
              mimeType = "video/*";
              URL pl = new URL(playlist);
              uri = Uri.parse(pl.toString());
           }
           if(uri != null) {
              myAssets.add(new MyAsset(c.getInt(0),uri,c.getString(3),mimeType));
           }
       }
    }
```

```
}
finally { if( c != null && !c.isClosed()) { c.close(); } }
```

Subscriptions

Above we described how you can enqueue a single video (flat file or segmented) for download. In addition, the SDK can subscribe to a **feed** of episodic videos. We'll explain that here.

The Backplane keeps track of which SDK instance is subscribed to which feeds. As new episodes in a feed become known to the Backplane, the Backplane informs each SDK that subscribes to that feed, either through a GCM push notification (if you've set up GCM for your app), or through the normal SDK-Backplane sync. In response, the SDK automatically adds the new episode to its download queue:



Let's go through how this works.

Subscription-Related API Methods

You manage subscriptions via the following methods:

```
/*
subscribe to the feed with the given feedID. Store no more than maxAssets
episodes on the device at any one time. If you're storing maxAssets already,
and a new episode is available for download, then 'canDelete' determines the
behavior. If canDelete==TRUE, then delete the oldest stored episode to make
room for the new one. maxBitRate is the highest bitrate version (profile)
to download.
*/
void subscribe(final String feedUuid, int maxAssets, boolean canDelete, int maxBitRate)

/*
subscribe to the feed with the given feedID. Rely on default values for the other
parameters.
*/
void subscribe(String feedUuid)

/*
unsubscribe from the given feed
*/
void unsubscribe(final String feedUuid)
```

To retrieve results from the calls `unsubscribe()`, `subscribe()` or `subscriptions()`, you must implement an `ISubscriptionObserver`:

```
final ISubscriptionObserver mSubscriptionsObserver = new ISubscriptionObserver() {
  @Override
  public void onSubscribe(final int result, final String uuid) {
    if (result == BackplaneResult.SUCCESS) {
      /* Successfully subscribed to 'uuid' */
    } else { /*  failure; do something here */ }
```

```
  }

  @Override
  public void onUnsubscribe(int result, String uuid) {
    if (result == BackplaneResult.SUCCESS) {
      /* Successfully unsubscribed from 'uuid' */
    } else { /*  failure; do something here */ }
  }

  @Override
  public void onSubscriptions(final int result, final String[] uuids) {
    if (result == BackplaneResult.SUCCESS) {
      /* 'uuids' includes all the feeds the user is subscribed to */
    } else { /*  failure; do something here */ }
  }
};
```

Getting Metadata for a New Episode

When the SDK receives a GCM push notification to download a new episode, the SDK needs to get the metadata for that episode from somewhere. The "missing" metadata may include the remote URL, bitrate/profile to use, asset expiry, title, description, an image, to name a few.

To fill in this missing data, you must implement a class derived from `SubscriptionsService`:

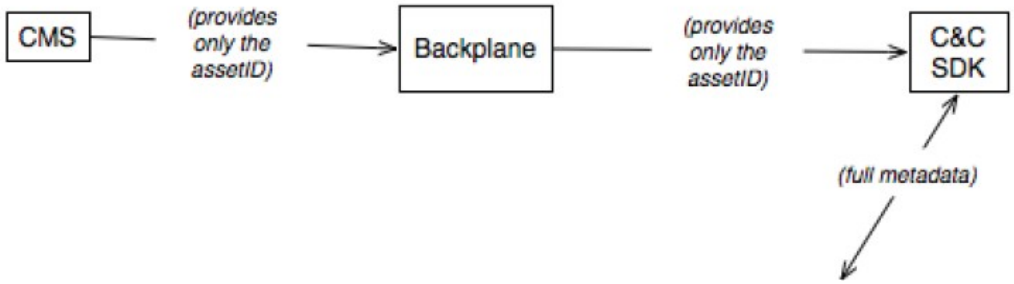
```
public class HarnessSubscriptionsService extends SubscriptionsService {
    @Override
       protected void onHandleIntent(Intent intent) {
           super.onHandleIntent(intent);
       }

    @Override
       public JSONObject processingFeedWithData(String uuid, JSONObject data,
                                                boolean complete) {
              //set values for Max bitrate,Max Episodes, Can Delete rules
              return data;
       }

    @Override
       public JSONObject processingAssetWithData(String uuid, JSONObject data,
                                                boolean complete) {
              // Add Data Here
              return data;
       }

     @Override
       public void processedAsset(IIdentifier aIdentifier) {
              //update asset data if needed. Has not yet been added to the queue
       }

    @Override
       public IVirtuosoIdentifier willAddAssetToQueue(IIdentifier aIdentifier) {
              // Modify asset before it is added to the queue
              return content;
       }
}
```

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

Scenario 1: All required asset metadata is provided from the Backplane

In this scenario, the SDK receives all the mandatory metadata from the Backplane. The SDK will call the `processingAssetWithData` method with `complete=true`. In this case, your app doesn't need to provide any additional data and can simply return the passed-in JSONObject.

Scenario 2: The Backplane doesn't supply all metadata; you need to fetch it manually.



In this case, you must provide the data to the SDK via the `processingAssetWithData` method. In that method, you need to add any required metadata to the supplied JSONObject. For instance:

```
@Override
public JSONObject processingAssetWithData(String uuid,
                                          JSONObject data, boolean complete) {

     ...
     if (!data.has(SubscriptionKey.DOWNLOAD_URL))  {
           data.put(SubscriptionKey.DOWNLOAD_URL, url);
      }
     ...
    return data;
}
```

The `SubscriptionKey` class details the available keys used by the subscription service.

Configuring Subscription Behavior

The SDK provides three ways to configure subscription behavior.

Maximum Bitrate: When the SDK encounters a manifest with multiple profiles, which profile should it download? If this value is Integer.MAX_VALUE, the SDK selects the profile with the highest bitrate. If the value is not Integer.MAX_VALUE, the SDK selects the profile with the highest bit-rate not exceeding this value. If no such profile exists (all profiles are of a higher bit-rate), the SDK will select the profile with the lowest bit-rate. (default: 1)

Maximum Subscribed Assets Per Feed: The SDK can limit the number of episodes N saved in each feed. Once the SDK has downloaded its quota in a feed, it won't download a new episode until one of the existing episodes is deleted. (default: `Integer.MAX_VALUE`)

Auto Delete Old Assets: Determines how the SDK behaves when a new episode is available and the device is already storing its quota for this feed. If YES, the SDK deletes the oldest downloaded file in this feed. If NO, then the SDK won't automatically download the new episode. (default: YES)

You can set these values when subscribing to a feed:

```
public void subscribe(final String feedUuid, final int maxAssets,
                      final boolean canDelete, int maxBitRate)
```

Later, you can update the values with the following APIs:

```
// Global values for all feeds applied through ISettings Interface.
public ISettings setMaxBitrateForSubscriptions(int bitrate)
public ISettings setMaxAssetsForSubscriptions(int max);
public ISettings setCanAutoDeleteForSubscriptions(boolean canDelete);

// Values for a specific feed
public void setFeedMaxBitRate(String feedUuid, int bitrate)
public void setFeedMaxAssets(String feeduuid, int max);
public void setFeedCanDelete(String feeduuid, boolean canDelete);
```

NOTE: Only assets downloaded through a subscription count towards these rules. If you enqueue an episode of a feed manually, that enqueued asset will not be automatically deleted or cause new downloads to be deferred.

Appendix A: How Downloading Works

Here we describe what happens after you enqueue an asset for download. 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.
2. If Virtuoso encounters an error downloading the file, it will try that file two more times before increasing its error count and 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 as well as new files that may have been added.
4. Virtuoso will no longer try to download an asset once its error count reaches 3, unless you reset it (This is the "outer" three). You can reset the error state on an asset by using the `resetErrors` method of the `IQueue` interface.

Here are the error conditions Virtuoso may encounter:

Condition	Description	Result
Server-advertised file size disagrees with expected file size	You provided an expected size for the file and it does not match the Content-Length supplied by the HTTP server.	SDK updates Asset's status to `AssetStatus.DOWNLOAD_FILE_SIZE_MISMATCH` and increments its error count.
Invalid mime type	MIME type advertised by the HTTP server does not match the expected MIME type you supplied.	SDK updates Asset's status to `AssetStatus.DOWNLOAD_FILE_MIME_MISMATCH` and increments its error count
Observed file size disagrees with expected file size	After the download completes, the size of the downloaded file on disk doesn't match the expected size specified when the file was created OR doesn't match the Content-Length supplied by HTTP server.	SDK updates Asset's status to `AssetStatus.DOWNLOAD_FILE_SIZE_MISMATCH` and increments its error count
Network error	Some network issue (HTTP 404,416, etc.) caused the download to fail.	SDK updates Asset's status to `AssetStatus.DOWNLOAD_NETWORK_ERROR` and increments its error count
File system error	The OS couldn't write the file to disk. In most cases, the root cause is a full disk.	SDK updates Asset's status to `AssetStatus.DOWNLOAD_FILE_COPY_ERROR` and increments its error count

The SDK notifies your app of errors that occur during download through the `IQueueObserver` interface. The callback method for receiving details of errored assets is `engineEncounteredErrorDownloadingAsset(IIdentifier aAsset)`. You can determine the type of error by examining the status of the asset (`getDownloadStatus`) sent in the callback.
If you are using a Cursor obtained through the `IQueue` interface then the SDK will notify the `IQueue` Content Uri of the change.

Appendix B: Steps to Create a New App

The quickest way to start developing your own app is to create a simple Android "hello world" project. You can then add in the SDK libraries and modify the manifest accordingly.

1. Create "hello world" app

1. Create a new Android Project
2. Provide the app's name and package; click next
3. Leave the default options alone; click next
4. Choose an icon to use in the app or leave the defaults; click next.
5. Choose the kind of activity to create; click next.
6. Set the name of your activity; click finish

2. Add libraries

Add gradle dependencies for the maven respository and the library as defined at the beginning of this document.

3. Create your ContentProvider

You'll need to add a content provider to your app. This content provider will provide the Virtuoso service with access to the database used for your app's media.

Each app has its own database and is stored in the app's data directory.

Android doesn't permit installation of a content provider it there already exists one on the device with the same name or using the same authority. The SDK provides a framework for content providers so that the Virtuoso service can use a content provider for each application without conflicts.

To create the new content provider:

1. Click on your package and open the context menu to add a new class.
2. Provide a name for your class. To ensure its uniqueness, we recommend naming it with the name of your application followed by the string "`ContentProvider`".
3. Extend your class from the abstract Virtuoso content provider. Click on the "Browse..." button of the Superclass field and select the `VirtuosoSdkContentProvider` as the super class.
4. Click Finish
5. Your generated class should look similar to this:

```
package com.demo.myvirtuosoapp;
import com.penthera.virtuososdk.database.impl.provider.VirtuosoSDKContentProvider;
public class MyVirtuosoAppContentProvider extends VirtuosoSDKContentProvider {
	@Override
	protected String getAuthority() {
		// TODO Auto-generated method stub
		return null;
	}
}
```

Implement the `getAuthority` method and provide a static implementation that sets the Authority to be used for this content provider. The authority string should uniquely identify your content provider on an Android device. We recommend that the authority string consist of:your package name, your application name and the string "`virtuoso.content.provider.`" For example:

```
public class MyVirtuosoAppContentProvider extends VirtuosoSDKContentProvider {
```

```
        static{

        setAuthority("com.demo.myvirtuosoapp.myvirtuosoapp.virtuoso.content.provider");
        }
        @Override
        protected String getAuthority() {
                return "com.demo.myvirtuosoapp.myvirtuosoapp.virtuoso.content.provider";
        }
}
```

4. Update the AndroidManifest.xml

Add the following permissions to the manifest:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
```

5. Add large heap support to the Application

Adding the largeHeap flag to the application in the manifest is recommended to allow for best performance if you are downloading large files:

```
<application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme"
        android:largeHeap="true">
```

6. Add the Virtuoso Service

Add the Background service in the application section of the AndroidManifest:

```
<application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >

<service android:name="com.penthera.virtuososdk.service.VirtuosoService"
                                    android:label="VirtuosoService"
                                    android:process=":vservice" />
.
.
.
</application>
```

7. Add the Client HTTP Service

The HTTP service sits between the device media player and the downloaded files. Its job is to abstract the location of the downloaded files, and to enforce access policies on the files.

For downloaded segmented assets, the HTTP service generates a manifest for the media player.

```
<service android:name="com.penthera.virtuososdk.service.VirtuosoClientHTTPService"
                                    android:label="VirtuosoClientHTTPService"
                                    >
</service>
```

8. Add the Virtuoso Service starter

The Virtuoso service starter ensures the Virtuoso service starts on device boot and keeps running in the background. You'll need to add it to the application section of the AndroidManifest:

```
<receiver android:name="com.penthera.virtuososdk.service.VirtuosoServiceStarter"
               android:enabled="true"
               android:label="VirtuosoServiceStarter"
               android:process=":vservices">
        <intent-filter>
               <action android:name="android.intent.action.BOOT_COMPLETED"/>
              <action android:name="android.intent.action.QUICKBOOT_POWERON"/>
              <action android:name="com.htc.intent.action.QUICKBOOT_POWERON"/>

               <action android:name="virtuoso.intent.action.DOWNLOAD_UPDATE"/>
               <action
        android:name="virtuoso.intent.action.START_VIRTUOSO_SERVICE_LOGGING"/>
               <action android:name="virtuoso.intent.action.BACKPLANE_SYNC_DEVICE" />
                <action android:name="android.intent.action.PACKAGE_REMOVED" />
                 <data android:scheme="package"/>
       </intent-filter>
       <intent-filter>
                 <action android:name="android.intent.action.BOOT_COMPLETED" />
                 <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>

</receiver>
```

9. Add the metadata

This metadata identifies your app with the background service. The meta-data is used when a Virtuoso service needs to access your content provider. The metadata details the authority string used by your provider. The name of the metadata is `com.penthera.virtuososdk.client.pckg`.

```
<meta-data android:name="com.penthera.virtuososdk.client.pckg"
android:value="com.demo.myvirtuosoapp.myvirtuosoapp.virtuoso.content.provider" />
```

10. Add the content provider

Add this to application element of the AndroidManifest. The content provider enables the SDK to access the SQLite database within your app directory. The authority string for the content provider must match that supplied in the metadata element and the authority String being supplied in the content provider class you created.

```
<provider android:name="com.demo.myvirtuosoapp.MyVirtuosoAppContentProvider"
android:authorities="com.demo.myvirtuosoapp.myvirtuosoapp.virtuoso.content.provider"
android:process=":vservicec"/>
```

In your class derived from `VirtuosoSDKContentProvider`:

```
public class MyVirtuosoAppContentProvider extends VirtuosoSDKContentProvider {
    static {
      setAuthority("com.demo.myvirtuosoapp.myvirtuosoapp.virtuoso.content.provider");
       }
       @Override
       protected String getAuthority() {
           return "com.demo.myvirtuosoapp.myvirtuosoapp.virtuoso.content.provider";
       }
```

```
}
```

11. Add the DRM license manager

Optionally, derive a class from LicenseManager and register it with metadata in the Manifest so the SDK can use it to fetch Widevine licenses.

At a minimum, implement the getLicenseAquisitionUrl() for your license manager:

```
public class DemoLicenseManager extends LicenseManager {
   @Override
   public String getLicenseAcquistionUrl() {
       String license_server_url = "https://proxy.uat.widevine.com/proxy";
        /*
        Here you can examine the mAsset and mAssetId member variables and modify the
        license server url if needed:
        Example:
        String video_id = mAsset != null ? mAsset.getAssetId() :
                            mAssetId != null ? mAssetId : null;
        if(!TextUtils.isEmpty(video_id){
         license_server_url += "?video_id="+video_id + "&provider=widevine_test";
        }
        */
       return license_server_url;
   }
}
```

For the SDK to use your License Manager implementation you need to override a metadata value in the AndroidManifest. To override metadata you will need to add the tools namespace.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools" … >

<meta-data tools:replace="android:value"
android:name="com.penthera.virtuososdk.license.manager.impl"
android:value="com.penthera.sdkdemo.drm.DemoLicenseManager"/>
```

You should now be able to build and run your project without errors.

** END OF DOCUMENT **