

Penthera 302: Android SDK Beyond the Basics

Android SDK 4.0

May 12, 2021

Advanced development documentation for Penthera Android SDK 4.0. Covers intermediate and advanced topics, as well as extended SDK features.

Copyright © 2020 Penthera Partners

CONFIDENTIAL - Licensed Use Only

Table of Contents

Intermediate Topics	3
Configure Device-level Download Behaviors: Android	3
Notification of Account & Asset Permissions Violations in Android	5
Download Ancillary Files: Android	6
Enable/Disable Other Devices: Android	7
Push Notifications: Android	8
IPushRegistrationObserver: Android	12
Fixing Firebase Cloud Messaging errors: Android	12
Availability Windows for Assets: Android	13
Mark an Asset as Expired: Android	15
Error handling for MIME types	16
MIME type error behavior	16
Configuring MIME type settings	17
Advanced Topics	18
Upgrading the SDK: Android	18
Database Versioning: Android	18
Upgrading to Penthera 4.0 Android SDK	18
Observing and Interacting with Asset Creation & Queuing	20
Observing Manifest Parsing Results with ISegmentedAssetFromParserObserver:	
Android	20
Observing Queue Additions with IQueuedAssetPermissionObserver: Android	21
Observing or Modifying Manifest Parsing with IManifestParserObserver: Android	21
Android occasionally changes the proxy port	24
Using Cursors to Access Assets	24
DRM Setup: Android	27
Retrieve and Persist Widevine Licenses: Android	27
Playout with Widevine Persisted License: Android	29
Errors with DRM License Durations on Android 8	30
Extended Features	32
FastPlay: Android	32
Auto-download	33
Implement the IPlaylistAssetProvider	34
Register a Playlist	35
Grow a Playlist	36
Advertising Support: Android	36
What Next?	38

Intermediate Topics

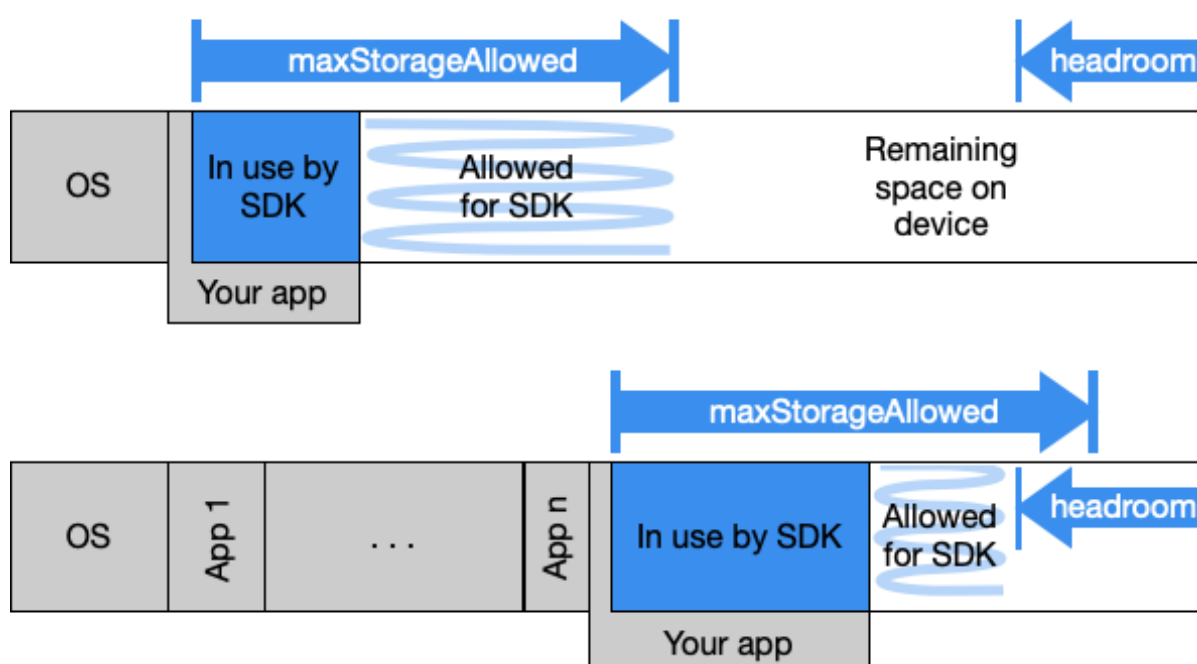
Configure Device-level Download Behaviors: Android

The SDK obeys several device-level 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.

Configuration Parameters

- `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 will not download (default: 1GB)
- `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 5GB).

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



- `batteryThreshold` Defines the 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: 0.5, meaning 50%)
- `cellularDataQuota` How many 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: 100MB). The SDK divides this number into four and enforces the smaller number on a week-to-week basis.
- `destinationPath` 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. Note that by default this path is

within the app private file space on the first "external" file system (which in many cases is not actually "external" storage, but is called so by Android). 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.



NOTE

It can be difficult for the SDK to estimate file sizes without knowing the file sizes ahead of time, especially when content is not encoded with a constant bitrate. The initial estimated file size for videos is based on the selected bitrate to download and the duration of the asset. Once download starts, the SDK will look at the existing downloaded segments and create an "average segment size" for each segment type (audio, video, etc.) and then recalculate the estimated total based on the number of segments multiplied by the average downloaded segment size. Naturally, with this approach, the estimated size will become more accurate over time.



LOW BATTERY IMPACTS VARIOUS FEATURES

The SDK battery threshold value in `ISettings` impacts the SDK download behavior, but device battery level impacts other SDK behaviors which are not controllable by the SDK's `ISettings` value. Various functions of the SDK are implemented as Android OS Workmanager tasks which are configured not to run whenever "battery is low". The battery level which Android OS Workmanager considers "low" is commonly set to 20% and is not under control by the SDK. Thus, independent of the SDK `ISettings` value you may see the following SDK functions delayed whenever the OS determines the device is in a low-battery state:

- Execution of Expiry actions
For assets with SDK expiry values the SDK will still prevent playback of expired assets, but the automatic removal of the expired asset files from disk may be delayed if device battery is low.
- Automatic refresh of Ads
- Timed refreshes of DRM (refreshes requested manually by your code will still occur)

Retrieve a parameter setting

Current values for settings can be retrieved from the `ISettings` instance:

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

Override parameter settings

The `ISettings` instance also allows to override defaults. Remember to call `save()` to cause setting changes to persist.

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

Reset a setting to the SDK default or to that provided by your Penthera Cloud configuration

`ISettings` provides reset methods to restore settings to SDK defaults. For example:

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

Notification of Account & Asset Permissions Violations in Android

The *preferred mechanism* for receiving notification about issues occurring during the download of an asset is the `IQueueObserver`, as discussed in [IQueueObserver: Android](#). If for any reason you cannot use the `IQueueObserver` approach, similar information can be found by registering to receive the related Broadcasts.

To use the less-preferred mechanism of Broadcasts to receive and act upon notification when some of the typical account-wide and asset-specific download permissions have been violated:

1. Ensure that your registered Broadcast receiver is configured in `AndroidManifest.xml` with an intent-filter allowing the `NOTIFICATION_DOWNLOAD_STOPPED` action. See [Using Broadcast Receivers: Android](#) for more details.

```
com.penthera.virtuososdk.provider.sdkdemo.NOTIFICATION_DOWNLOAD_STOPPED
```

2. In your Broadcast receiver, when it receives the notification intent, look into the action to determine it is related to "download stopped":

```
String myAction = intent.getAction();  
if  
(myAction.endsWith(Common.Notifications.INTENT_NOTIFICATION_DOWNLOAD_STOPPED))  
{  
    //handle download stopped by reason (see sample below)  
}
```

3. Once the action is determined to be `INTENT_NOTIFICATION_DOWNLOAD_STOPPED`, take appropriate steps based on the `DownloadStopReason` contained in the intent extras:

```

Bundle myExtras = intent.getExtras();
if (null != myExtras && myExtras.containsKey(
Common.Notifications.EXTRA_NOTIFICATION_DOWNLOAD_STOP_REASON ))
{
    //extras should contain reason codes
    int stopReason =
myExtras.getInt(Common.Notifications.EXTRA_NOTIFICATION_DOWNLOAD_STOP_REASON);
    switch (stopReason)
    {
        case
Common.Notifications.DownloadStopReason.ERROR_DOWNLOAD_DENIED_ACCOUNT:
            //violated the Max Downloads per Account Rule
            break;

        case
Common.Notifications.DownloadStopReason.ERROR_DOWNLOAD_DENIED_ASSETS:
            //violated the Max Downloads per Asset Rule
            break;

        case
Common.Notifications.DownloadStopReason.BLOCKED_DOWNLOAD_DENIED_COPIES:
            //violated the Max Copies of Asset rule
            break;

        case
Common.Notifications.DownloadStopReason.DENIED_MAX_DEVICE_DOWNLOADS:
            //violated the Max Permitted Downloads on Device Rule
            break;
    }
}

```

**NOTE**

You may also be interested in receiving `Common.Notifications.INTENT_NOTIFICATION_DOWNLOADS_PAUSED` events.

Download Ancillary Files: Android

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

Below is an example where a DASH asset is defined, including providing URLs, descriptions, and tags for both thumbnail and poster images. When the asset builder's `.build()` method is called, the SDK will manage retrieving the related ancillary files along with the manifest and the audio and video files appropriate to the desired bitrates.

```
MPDAssetBuilder mpdAsset = new MPDAssetBuilder()
    .assetObserver(observer)
    .manifestUrl(new URL(url))
    .desiredAudioBitrate(0)
    .desiredVideoBitrate(0)
    .addToQueue(true)
    .assetId(remoteId);

mpdAsset.withAncillaryFiles(
    Arrays.asList(
        new AncillaryFile[]{

            new AncillaryFile(new URL("http://yourserver.s3.amazonaws.com/
somepath/0001thumb.jpg"), "My Movie Thumbnail Description", new String[]
{"tag1", "tag2", "thumbnail"}),

            new AncillaryFile(new URL("http://yourserver.s3.amazonaws.com/
somepath/0001poster.jpg"), "My Movie Poster Description", new String[]
{"tag1", "tag2", "poster"})
        }
    )
);

assetManager.createMPDSegmentedAssetAsync(mpdAsset.build());
```

Once the SDK has completed download of the asset, the ancillary files can be accessed from the asset instance. You could access all the ancillaries at once with `getAncillaryFiles(context)`, but notice in the example below that the tags used when declaring the ancillary files can be used to easily retrieve a specific ancillary file. Tags could be used to distinguish thumbnails from poster images, to mark ancillary images in various different languages, or for any other purpose.

```
List<IIdentifier> ls = assetManager.get(myAssetId);
ISegmentedAsset myDashAsset = (ISegmentedAsset) ls.get(0);

List<AncillaryFile> thumbnails =
myDashAsset.getAncillaryFilesForTag(context, "thumbnail");
AncillaryFile myThumb = thumbnails.get(0);

File myThumbImage = new File(myThumb.localPath);
String myThumbText = myThumb.description;
```

Enable/Disable Other Devices: Android

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

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

```
String anExternalIdToMatch = "AN_EXTERNAL_ID"; //device we wish to change

IBackplane backplane = mVirtuoso.getBackplane();
backplane.getDevices( new IBackplaneDevicesObserver () {
    @Override
    public void backplaneDevicesComplete(IBackplaneDevice[] aDevices) {
        //an empty array is received if there is no active connection or
        the user is not authenticated to the Penthera Cloud
        for(IBackplaneDevice device : aDevices) {
            //check the device
            if(anExternalIdToMatch.equals(device.externalId()) {
                backplane.changeDownloadEnablement(true,device)
            }
        }
    }
});
```

**TIP**

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

Push Notifications: Android

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

The Penthera SDK provides the default classes necessary to integrate Penthera functions using either Amazon Device Messaging (ADM) or Firebase Cloud Messaging (FCM). To leverage push notifications with the Penthera SDK in your app, you will need to:

- complete the generic install of either FCM or ADM into your project,
- configure your manifest to reference the provided Penthera SDK components to support FCM or ADM, and
- enter your messaging info into the Penthera Cloud admin, so that our cloud can send to your app.

If you need to receive push messages from other sources into your app, for purposes outside of the Penthera SDK, you may subclass our provided components to handle your other messages elsewhere in your app. This is the simplest approach for an app that does not already use push. If you are adding our SDK into an app where you already implement your own push notifications, you may instead wish to use a method on our SDK API which will inform the SDK of the push message, and our SDK will determine if it should handle the message or not. That method will return a boolean to indicate if the message was intended for the SDK and has been consumed, allowing your app to perform normal processing with other messages.

**NOTE**

Because the Penthera SDK ships with classes supporting ADM, if you are not including the third-party Amazon Device Messaging library in your project, your compiler may warn of missing class dependencies. These may look like the following:

```
com.penthera.virtuososdk.client.subscriptions.ADMReceiver:
can't find superclass or interface
com.amazon.device.messaging.ADMMessageReceiver
com.penthera.virtuososdk.client.subscriptions.ADMService: can't
find superclass or interface
com.amazon.device.messaging.ADMMessageHandlerBasecom.penthera.vi
rtuososdk.client.subscriptions.ADMReceiver: can't find
referenced class
com.amazon.device.messaging.ADMMessageReceivercom.penthera.virtu
ososdk.client.subscriptions.ADMReceiver: can't find referenced
class
com.amazon.device.messaging.ADMMessageHandlerBasecom.penthera.vi
rtuososdk.client.subscriptions.ADMService: can't find
referenced class
com.amazon.device.messaging.ADMMessageHandlerBasecom.penthera.vi
rtuososdk.subscriptions.PushTokenManager: can't find referenced
class
```

Code minimization by ProGuard or R8 will remove the unused files, but to suppress related warnings you can use the following `dontwarn` directives in your `proguard` config:

```
-dontwarn
com.penthera.virtuososdk.client.subscriptions.ADMService
-dontwarn
com.penthera.virtuososdk.client.subscriptions.ADMReceiver
```

To configure your app to use a messaging provider, do the following:

Complete the generic install of your messaging provider

To receive push messages either through Amazon Device Messaging (ADM) or Firebase Cloud Messaging (FCM) you need to add the appropriate package dependency, and configure your application to enable push notification receivers and services. For supporting FCM you need to provide the `google-services.json` file obtained from the Firebase console. If using 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](#). For ADM you will wind up with manifest entries similar to the following:

```

    <!-- This permission allows your app access to receive push
notifications from ADM. -->
    <uses-permission
android:name="com.amazon.device.messaging.permission.RECEIVE" />

    <!-- This permission enables ADM -->
    <amazon:enable-feature
        android:name="com.amazon.device.messaging"
        android:required="false" />

    <!-- This permission ensures that no other application can intercept
your ADM messages. -->
    <permission
        android:name="com.yourdomain.yourapp.permission.RECEIVE_ADM_MESSAGE"
        android:protectionLevel="signature" />
    <uses-permission
android:name="com.yourdomain.yourapp.permission.RECEIVE_ADM_MESSAGE"

```

**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 GooglePlay Services](#)

Configure your manifest for the provided Penthera SDK messaging components

For the SDK to function correctly with FCM or 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.

For FCM you will add a service, similar to the following:

```

<service
android:name="com.penthera.virtuososdk.client.subscriptions.FcmMessagingService">
    <intent-filter>
        <action android:name="com.google.firebase.MESSAGING_EVENT" />
    </intent-filter>
</service>

```

For ADM you will add a service and receiver, similar to the following:

```
<service
    android:name="com.penthera.virtuososdk.client.subscriptions.ADMService"
    android:exported="false" />

<receiver
    android:name="com.penthera.virtuososdk.client.subscriptions.ADMReceiver"
    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
        package name. -->
        <category android:name="com.yourdomain.yourapp" />
    </intent-filter>
</receiver>
```

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 the `super` implementation of any methods which are overridden. See `DemoFCMService` and `DemoADMService` in the SDK Demo application for an example of how to subclass the SDK classes. The SDK Demo also shows how to declare them in the manifest, with your custom components registered instead of the SDK default implementations.

Configure the Penthera Cloud with your messaging credentials

Using the URL and login credentials provided by Penthera Support, log into your Penthera Cloud admin console to add your messaging credentials to your cloud application config. It is acceptable to provide the credentials even if they are also being used by another messaging provider in your infrastructure, though with FCM you do have the option of using separate keys targeting different services in the same app.



PERFORMANCE WITH PUSH NOTIFICATIONS

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

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

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

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

The Penthera Android SDK guarantees at least one sync every 24 hours, as long as the device comes online during that time. If no sync has occurred within the past 24 hours, the SDK will attempt to sync when the network state changes (the device comes online), or when the app is brought to the foreground.

Generally, push notices are posted to the device fairly quickly by Firebase after the notice is sent from the Penthera Cloud.

IPushRegistrationObserver: Android

Implement the `IPushRegistrationObserver` interface provided in the Penthera SDK to monitor for success/failure when registering a device with push messaging services. This interface provides the callback method `onServiceAvailabilityResponse(int pushService, int errorCode)`, where the `pushService` will be one of those listed in `Common.PushService` in the Penthera SDK. If using GCM/FCM, the `errorCode` will be one of those from `GooglePlayServices com.google.android.gms.common.ConnectionResult`, or if using ADM the `errorCode` will be from `Common.AdmPushServiceAvailability` in the Penthera SDK.

Fixing Firebase Cloud Messaging errors: Android

If any of the following errors are showing in your logs, there is a configuration issue with your Firebase Cloud Messaging (FCM):

- Received error: Received push with missing authorization on platform
- Received error: Received push with invalid authorization on platform
- Application does not have permission to send to this device.
- Original Message: Error: mismatched sender ID


**NOTICE**

GCM has been deprecated by Google and is being phased out. To support "legacy systems", the FCM setup supports a "Legacy Token" format. Which essentially behaves exactly the same way as GCM did, but the below screens may look a little different now than they did before the change.


Check the FCM Server API Key

If it appears that your push notifications are no longer being received on Android devices, it is possible that you are using the **Android API Key** rather than the **Server API Key** to identify your project in our Penthera Cloud.


To fix this issue, create a new key:

1. Log into the [Firebase](#) site.
2. Select your project.
3. From the Settings button , choose **Project Settings**.
4. Choose **Cloud Messaging** to view your Server Keys. Either of these keys can be used in our system.
5. Ensure that this is the Key used in configuring the Cloud Messaging services in the Penthera Cloud dashboard. The Server Key should go in the **GCM** field.

Check the Sender ID

1. Log into the [Firebase](#) site.
2. Click the Settings button  and choose **Project Settings**.
3. Choose **Cloud Messaging** to view your Sender ID. your project.

Double check the Package Name, if set for your project.

1. Select the **Settings** button  and choose **Settings** from the menu.
2. If you have a **Package Name** configured, it will be displayed.

Other common things to check

- If you see the error *Received error: Received push with missing authorization on platform*, check that Cloud Messaging is enabled in the console.
- Background data needs to be enabled on devices. Otherwise pushes cannot send to Android devices until they have active WiFi connection. To enable background data for each version of Android OS, please consult Android documentation.
- Notifications are not enabled for the application itself. Notifications are enabled or disabled in the *Manage Applications* screen. Navigate there for each Android OS by selecting *Show Notifications*.
- If you are testing a production build, the *inProduction flag* in your code should be set to **true**. Otherwise, it should be set to **false**.
- The *Mismatched Sender ID* error happens when your Sender ID is incorrect. Ensure you're using your project number, not ID.

Availability Windows for Assets: Android

If you are using DRM, your DRM configuration will typically control the availability of the asset for playback. The Penthera SDK provides "Availability Windows" for assets in cases where customers are not using DRM, or in some cases where additional control features may be desired in addition to the DRM.

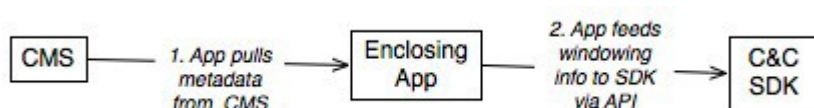
The Availability Window governs when the video is actually available to the end user, even if its files have already been downloaded onto the device by the SDK.

The Penthera SDK Availability Window enforces several windowing parameters, which may be set on a per-asset basis. You may use one or more of the parameters at a time, or none at all:

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 with the SDK when the video is played.

The Penthera Cloud stores a global default value for EAP and EAD. You may set these values from the Penthera Cloud admin web console. If set, the Penthera Cloud transmits these default values to all SDK instances. Settings received from the Penthera Cloud may be overridden by code on the device.

Typically, a Content Management System (CMS) stores 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 any of the availability window parameters 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 an asset:

```

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();
  
```



TIP

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 meta-data. This allows your app to access information about the expired asset, such as to indicate expired assets in your user interface. The SDK provides access to these expired assets through the `IAssetManager`.

**NOTE**

The SDK automatically removes expired assets from the download queue.

**NOTE**

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

**NOTE**

The Penthera SDK has an internal secure clock implementation that is entirely independent of the local clock time. The secure clock time is based on a combination of NTP calls and Penthera Cloud time stamping for "ground truth time," combined with tracked differentials of the system "up-time clock". All expiry calculations are done against this internal secure clock, so that expiry is independent of the local clock settings. The secure clock maintained by the Penthera SDK provides a public API call to determine the secure clock date/time.

**LOW BATTERY IMPACTS VARIOUS FEATURES**

The SDK battery threshold value in `ISettings` impacts the SDK download behavior, but device battery level impacts other SDK behaviors which are not controllable by the SDK's `ISettings` value. Various functions of the SDK are implemented as Android OS Workmanager tasks which are configured not to run whenever "battery is low". The battery level which Android OS Workmanager considers "low" is commonly set to 20% and is not under control by the SDK. Thus, independent of the SDK `ISettings` value you may see the following SDK functions delayed whenever the OS determines the device is in a low-battery state:

- Execution of Expiry actions
For assets with SDK expiry values the SDK will still prevent playback of expired assets, but the automatic removal of the expired asset files from disk may be delayed if device battery is low.
- Automatic refresh of Ads
- Timed refreshes of DRM (refreshes requested manually by your code will still occur)

Mark an Asset as Expired: Android

In addition to the availability window information, expiry after download, and expiry after playback, your code also has the ability to expire an asset at any time. To manually mark an asset as having expired:

```
IIentifier myAsset = mAssetManager.get(uuidString);  
mAssetManager.expire(myAsset.getId());
```

**NOTE**

There is no requirement to manually expire an asset. The SDK will automatically track and mark expiry if appropriate metadata has been supplied in the Penthera Cloud settings or on the asset instance at creation or after.

Error handling for MIME types

The SDK checks the reported MIME content type of each file that is downloaded and will warn the developer or block the download if it determines that the content being downloaded does not appear to be appropriate for the expected content. The main purpose of this check is to intelligently catch cases where a proxy might intercept a request and return HTML content, for instance in the case of a sign-in page for a roaming network. Without detecting such occurrences there is a risk that the SDK would download and store an HTML page in place of a video segment, reporting the download as successful and subsequently delivering the page to a player.

The SDK checks the content type based upon the files purpose in a segmented asset, and automatically accepts default MIME types for video, audio, and text segments. If you are delivering content with content types that the SDK does not automatically accept then it may be necessary to configure those types within the MIME type settings. The download behavior changes between debug and production versions of the SDK to help developers identify where MIME types are not recognized. The debug version will stop the download and mark an asset as blocked upon encountering any unexpected content types in order to highlight issues during the development process, whereas the release version is more lenient of issues and will only block the download for the most questionable MIME types.

MIME type error behavior

During development, the debug SDK version applies strict adherence to the content type rules in order to help the developers ensure that any exceptions to the default types have been properly configured. Any content type that does not meet the configured or built-in rules will cause the download to fail. The asset download status will be set to indicate the download finished with an error caused by MIME type mismatch and an analytics event will be generated for the download error.

In production these rules are loosened to prevent new content types from causing assets downloads to fail unnecessarily, while serious potential failures such as receiving HTML or XML text types will still cause an error. The SDK will not raise an error for any content-type failure other than those indicating a text response. Instead it will log the issue to Logcat at ERROR level and generate an analytics event for download warning. If text MIME types are received for video or audio segments then the asset download status will be set to indicate a MIME type mismatch and analytics event generated for download error.

Download warnings can be monitored in the analytics output once the application is deployed to catch any issues which may occur in the future where mime types delivered by content management systems might change.

**IMPORTANT**

Receiving text MIME types such as text/html and text/xml for an audio or video segment will always cause the SDK to raise an error and stop the download.

Configuring MIME type settings

Additional MIME types can be managed in the SDK settings. They are specified by asset type (HLS or DASH) and segment types including video, audio, and text. The `MimeTypeSettings` object can be fetched from the `ISettings` interface. Please note that it must be applied back to the settings using the `setMimeTypes()` method for changes to take effect. The `MimeTypeSettings` class provides methods to get and set a collection of accepted MIME types for each asset type and segment type.

Example of adding "video/mp4" as an accepted MIME type for text segment types in DASH assets:

```
List<String> ccMimes = Collections.singletonList("video/mp4");
MimeTypeSettings mimes =
mVirtuoso.getSettings().getMimeTypeSettings();
mimes.setMimeTypesForFileAndSegment(MimeTypeSettings.ManifestType.DASH,
MimeTypeSettings.SegmentType.TEXT, ccMimes);
mVirtuoso.getSettings().setMimeTypeSettings(mimes);
```



NOTE

The default MIME types accepted by the SDK are pre-configured internally and do not need to be managed via the MIME type settings API. They include:

For all uses: application/binary, binary/octet-stream, application/octet-stream

Video/Audio: audio/*, video/*

Text (closed captions and subtitles): text/*, application/ttml*, application/mp4*

Ancillary files / Licenses / DASH init segments: all types accepted



NOTE

Prior to Penthera Android SDK 4.x, changes to `ISettings` values required calling `.save()` to commit them permanently. This is no longer necessary as of SDK 4.0.

Advanced Topics

Upgrading the SDK: Android

Database Versioning: Android

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 already had a more recent version of the database, 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.

Upgrading to Penthera 4.0 Android SDK

If you are upgrading to the 4.0 Penthera Android SDK from an earlier Penthera SDK, it is important to note that significant changes have been made with the goal of simplification and modernization. You will want to read this entire section to understand implications in your project from the following major SDK changes:

- Simplified integration and extended compatibility for various players has been achieved with compatibility classes. These are delivered as supporting dependencies, including various ExoPlayer versions as well as BitMovin player. Read further to learn how to leverage the integration libraries.
- Simplified AndroidManifest.xml by removing the need for boilerplate configurations of our download service and broadcast receiver. Read further to learn what can and should be removed from your manifest, or how to maintain legacy configuration support if desired.
- Simplified AndroidManifest.xml by removing the need for boilerplate configuration of our content provider. Read further to learn what can and should be removed from your manifest, or how to maintain legacy configuration support if desired.
- Eliminated the need for the Service Starter function to exist in a separate process. The legacy approach still works, but read further to learn how to simplify your processes configuration.
- Added support for AndroidX Component Architecture, including use of a ViewModel architecture and the exposure of SDK data as LiveData objects which are Lifecycle aware. Read further to learn how to include and use this approach in your app.

Player Compatibility Components

Add the new Penthera gradle dependency for the player integration component which is appropriate to your version of ExoPlayer, or for the Bitmovin player. You may optionally continue using the existing solution approach which supports ExoPlayer up to version 11, but to do so you will still need to reference the new support library we provide for ExoPlayer 11 because the previous ExoPlayer support classes have been removed from the core SDK and placed into that library. See [Playout with Widevine Persisted License: Android \[29\]](#) for further details which are useful whether your app uses DRM or not.

For ExoPlayer 12 and later use the new dependency, which significantly simplifies integrating support for DRM managed by the Penthera SDK. This integration uses the MediaItem configuration pattern as

well as the ExoPlayer playlist for scheduling multiple assets to play. This also helps with advertising support. The new player integration approach can be seen in the example projects included with the SDK distribution.

Simplified Service Declarations

The SDK no longer requires several of its legacy AndroidManifest.xml configuration blocks. In most cases you should remove the following from your manifest:

- Remove the `service` tag block which defines `com.penthera.virtuososdk.service.VirtuosoService`
- Remove the `receiver` tag block which defines the internal-use broadcast receiver `com.penthera.virtuososdk.service.VirtuosoService$ServiceMessageReceiver`
- Remove the `service` tag block which defines `com.penthera.virtuososdk.service.VirtuosoClientHTTPService`
- Remove the `uses-permission` tags which define the following permissions, unless your app uses them otherwise:

```
<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" />
```

If for any reason you need to retain your existing manifest definitions or control the process names, please contact Penthera Support for assistance to ensure your configuration properly overrides ours.

Content Provider Boilerplate

The 4.0 SDK provides a simplified content provider implementation. For apps migrating from pre-4.0 you may keep your content provider configuration as-is, or migrate to the new approach to simplify your project. If you wish to migrate to the new approach, you may:

- Remove your subclass of `VirtuosoSDKContentProvider` from the project
- Remove the `provider` block from `AndroidManifest.xml` which defines your content provider
- Use the new 4.0 approach to the content provider, which generates the content provider and configuration for you via gradle. This is covered in detail by [Set up the Android SDK](#) in our basic android dev guide for the 4.0 SDK. The high level approach is:
 - Ensure our gradle repository is referenced in your gradle settings
 - Add a reference to our SDK configuration plugin to your gradle plugins block

When configured with this new approach, the content provider class is generated for you, and the appropriate `AndroidManifest.xml` configuration is provided automatically as well.

Service Starter Process Simplification

The class in your project which derives from the Penthera SDK Service Starter (from `VirtuosoServiceStarter`) can now run in the main process rather than its own process. To make this change, modify the `android:process` tag of your `receiver` block in `AndroidManifest.xml` where you define your Service Starter.

AndroidX Component Architecture

The Penthera 4.0 SDK provides LiveData objects which are Lifecycle aware and can be integrated into your ViewModel classes. To use this approach you will add the new Penthera gradle dependency which contains the AndroidX extensions. In your code you will then create a LiveData factory object associated with the lifecycle owner, and Fetch LiveData objects from that factory to observe in your application. This is covered in detail by [Set up the Android SDK](#) in our basic android dev guide for the 4.0 SDK.

Observing and Interacting with Asset Creation & Queuing

The SDK provides a few observer interfaces which can be implemented to observe and interact with various steps in creating and queuing manifest assets. These include the following:

ISegmentedAssetFromParserObserver

If this observer is provided in the request to create an asset, it receives a notification when the `ISegmentedAsset` has been created from a parsed manifest. The observer call provides a reference to the asset after parsing was attempted, an indication of any parsing error, and a flag indicating if the asset was already added to the download queue.

This is the easiest place to confirm that asset parsing occurred successfully, and provides an opportunity to handle any error. You might use this as an opportunity to modify the asset details, but it is important to note that for some asset configurations (e.g., FastPlay enabled assets) and for assets already added to the queue, downloading may have already begun. To ensure asset modifications occur before any download is attempted, consider using `IManifestParserObserver` instead.

For further details see [Observing Manifest Parsing Results with ISegmentedAssetFromParserObserver: Android \[20\]](#).



DOWNLOAD MAY BEGIN BEFORE PARSING COMPLETES

As of the Penthera Android SDK 3.15.14 and iOS SDK 4.0, the engine can begin to download segmented asset files before the manifest parsing is complete. This allows downloads to begin sooner, but may result in parsing and downloading notifications arriving in a different order than expected in the past.

IQueuedAssetPermissionObserver

If this observer is provided in the request to create an asset or in the request to add an asset to the queue, it receives a notification when the asset has been added to the queue. This provides a reference to the asset after queueing was attempted, an indication whether the asset was successfully added to the queue, an indication whether download is permitted (at this time), and any relevant permission code. Note that when you provide this observer, the initial permission check attempt will occur when the asset is added to queue, regardless of the `IBackplaneSettings.requirePermissionWhenQueued` value. See [Observing Queue Additions with IQueuedAssetPermissionObserver: Android \[21\]](#) for further details.

IManifestParserObserver

Unlike other parsing and queue observers, this observer is configured in your app manifest, and runs in the SDK's download engine Service instead of in your main app process. This observer is notified after each segment is parsed, and again just before the asset is added to the download queue. This allows the observer to make modifications to either the asset or its segments as each segment is parsed from the manifest, or to modify the asset just before it is added to queue. This is the preferred location to modify the asset or segments, such as if you have a requirement to manually add custom tokens to segment URLs before download. See [Observing or Modifying Manifest Parsing with IManifestParserObserver: Android \[21\]](#) for further details.

Observing Manifest Parsing Results with ISegmentedAssetFromParserObserver: Android

The `ISegmentedAssetFromParserObserver` is passed into various constructors for segmented Asset instances in the SDK, and receives a callback when the parsing of that particular asset manifest is completed.

The observer interface has a single method `complete` which will be passed a reference to the `ISegmentedAsset`, an error code if necessary (or a 0, if no error), and a boolean indicating whether the asset was added to the download queue.

If the asset indicates it was added to the download queue, and no other assets were awaiting download, then the asset download may have started before this callback occurs. Download may have also begun if the asset was marked to enable FastPlay, even if it was not added to queue.

Depending on your UI design, an `ISegmentedAssetFromParserObserver` may be intended for use in the current Activity, or for a more global app purpose. If it is for an app-wide purpose, you may want to hold your observer outside the Activity where the asset was constructed. If instead the `ISegmentedAssetFromParserObserver` created during asset construction was designed only for updating the current Activity, then you may have dispensed with it if the user has navigated away from that Activity. In either case you have the option of later creating a new observer if needed. If the user navigates into an activity where you want to instantiate a new `ISegmentedAssetFromParserObserver` for a previously-created asset, you may create a new observer instance and pass it to the `addParserObserverForAsset` method of the `IAssetManager`. Note that the `IAssetManager` method will only register the observer if the asset parsing is not already complete and the asset has not reached the download pending state.



DOWNLOAD MAY BEGIN BEFORE PARSING COMPLETES

As of the Penthera Android SDK 3.15.14 and iOS SDK 4.0, the engine can begin to download segmented asset files before the manifest parsing is complete. This allows downloads to begin sooner, but may result in parsing and downloading notifications arriving in a different order than expected in the past.

Observing Queue Additions with `IQueuedAssetPermissionObserver`: Android

The `IQueue.IQueuedAssetPermissionObserver` enables the application to be notified when an asset has been added to the queue, and includes a permission response from the Penthera Cloud if that permission check occurs while being added to the queue.

The observer interface has a single method `onQueuedWithAssetPermission` which will be called to indicate if the asset has been queued, indicate whether download is permitted, provide a reference to the asset, and provide the permissions result code. This callback may occur before the manifest parse is completed, because some asset configurations cause the asset to be queued during the parsing process. If the asset is FastPlay enabled, or if no other assets in queue were awaiting download, then the asset download may be starting or in progress at the same time that this callback occurs.

The `IQueue.IQueuedAssetPermissionObserver` interface can be instantiated and passed either at the time of creation of the asset or via `addPermissionsObserverForAsset()` on `IAssetManager`. See the various `AssetBuilders` in the SDK for methods to add the observer when creating the asset.

Since the `IAssetManager` approach of adding this observer may occur well after creation of the asset, that approach will only register the observer if the asset parsing is not yet complete and the permissions request has not yet been processed. Due to that potential, it is recommended to provide this observer to the asset creation methods.

If you add this observer then the asset will have its permissions checked at the time of queuing instead of when download starts. This is equivalent to setting `requirePermissionsOnQueue` in the `IBackplaneSettings`, so do not use this observer if you want to defer permission checks to when the download begins. In either case, the SDK checks permissions again when download completes.

Observing or Modifying Manifest Parsing with `IManifestParserObserver`: Android

Manifest parsing occurs in the SDK's download Service, which is isolated in a different process from the main client application. The `IManifestParserObserver` is ideal if your application is required to al-

ter individual segment URLs prior to asset download, or to alter the asset properties prior to the asset being added to the download queue. Otherwise, the `IManifestParserObserver` is likely unnecessary, and the other observers you could provide during asset creation will provide enough functionality.

Implementing `IManifestParserObserver`

To provide an `IManifestParserObserver` to the SDK's download Service requires you implement the observer, and also implement and register a factory component in your manifest which the SDK which will use to instantiate the observer from within the download service.

1. Create your implementation of the `IManifestParserObserver` interface, using its `didParseSegment()` method to alter the segment URLs or `willAddToQueue()` method to alter parameters of the asset such as start window, end window, or download limits. Remember this class will execute in the SDK's Service thread, not in your main app thread, so be aware of any implications this might have for your implementation.

```
package com.yourcompany;

public class MyManifestParserObserver implements IManifestParserObserver
{
    @Override
    public String didParseSegment(ISegmentedAsset asset, ISegment
segment)
    {
        // If your assets require you add to or change the download URL
        // from the manifest prior to downloading it, then
        // you would use this method to return the modified URL.
        return segment.getRemotePath()+"&magicToken=123456";
    }

    @Override
    public void willAddToQueue(ISegmentedAsset aSegmentedAsset,
IAssetManager assetManager, Context context)
    {
        //You could modify the asset here and save it via the
IAssetManager.
        //See the SDK Demo for an example.
    }
}
```



DOWNLOAD MAY BEGIN BEFORE PARSING COMPLETES

As of the Penthera Android SDK 3.15.14 and iOS SDK 4.0, the engine can begin to download segmented asset files before the manifest parsing is complete. This allows downloads to begin sooner, but may result in parsing and downloading notifications arriving in a different order than expected in the past.

2. Next, implement the `IBackgroundProcessingManager` interface, which contains a factory method that will be called from within the SDK to instantiate your manifest parser observer at appropriate times. Implement the `getManifestParserObserver()` method such that it returns an instance of your observer. This method will be called separately for each asset that is parsed.

```

package com.yourcompany;

class MyExampleBackgroundProcessingManager implements
IBackgroundProcessingManager {
    @Override
    public IManifestParserObserver getManifestParserObserver() {
        return new MyManifestParserObserver();
    }

    @Override
    public IClientSideAdsParserProvider getClientSideAdsParserProvider()
    {
        return null;
    }

    @Override
    public ISubscriptionsProvider getSubscriptionsProvider() {
        return null;
    }

    @Override
    public IPlaylistAssetProvider getPlaylistProvider() {
        return null;
    }
}

```

**NOTE**

Notice that the `IBackgroundProcessingManager` is also used in other optional background processing, such as subscriptions and ads. If you are not using these other features, simply return null from unused methods.

3. Register your `IBackgroundProcessingManager` 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 your implementation's class name.

```

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

```

**IMPORTANT**

Because it is referenced in the manifest, your `IBackgroundProcessingManager` implementation will need to have its class name protected from ProGuard/R8 obfuscation. Our SDK is distributed with a ProGuard rule in place to protect the name of any class implementing the interface, which should be honored provided you included our SDK into your project automatically via gradle.



IMPORTANT

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.

Android occasionally changes the proxy port

On very rare occasion the port will have changed upon restart of the HTTP proxy. There is a very remote edge case tied to this occurrence, due to Android lifecycle timing. If the port changes and the player resumes AFTER the proxy service has restarted, your code might miss the notification and still attempt to restart playback on the old port. It is very difficult to trigger this error in testing, as the server nearly always restarts on the same port it was using prior, but it is theoretically possible.

If you get a playback error, storing the current playback position and restarting the player at that position with a newly requested manifest will correct for that problem. In nearly all cases however, the service will resume on the same port it was using prior, and playback will continue normally.

Using Cursors to Access Assets

You can also retrieve the file path or asset playlist through the Cursors provided by the Asset Manager.



TIP

If you get the playback URL from the cursor then it does not contain the full URL in the same way that the corresponding method on `IAsset` would do. The URL retrieve from the cursor only contains the relative path. You need to append it to the proxy url, which can be fetched from the base `Virtuoso` object with `getHTTPProxyBaseUri()`



TIP

The asset manager class itself provides a cursor over all assets, including those which are expired.

Here is an example of cursor use:


```
// 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(); }
}
```

DRM Setup: Android

The Penthera SDK provides out-of-box support for basic Widevine DRM, together with a mechanism to easily provide custom support for specialized DRM server requirements as necessary. With a few easy steps, the SDK will not only manage retrieval of DRM for you, it will also manage the lifecycle of keys for various assets, including refreshing the keys at appropriate intervals to ensure the best likelihood that keys are valid for offline playback when your user desires it. At playback, the SDK provides a DRM session manager which you will integrate with your player of choice, so that SDK-managed keys are provided to the player whenever necessary.

The `LicenseManager` class provided with the SDK implements most of the SDK `ILicenseManager` interface for you, and in basic cases you need only to override the `getLicenseAcquisitionUrl` method to provide the URL for your DRM server endpoint. A DRM server may also have special formatting or token requirements in the request and/or response, so various methods of the `LicenseManager` implementation can be overridden to provide increasing levels of customization. For the most extreme cases you could even create an entirely custom `ILicenseManager` implementation.

If specialized request headers are needed for your DRM server, override the implementation of the `getKeyRequestProperties` method to return a map of header key-value pairs. If even more specialized key request properties are required, you can specialize the URL returned from `getLicenseAcquisitionUrl` on a per-asset basis, or at the most intense levels of customization you could override `executeKeyRequest` and `executeProvisionRequest` methods to access the underlying Android `KeyRequest` and `ProvisionRequest` instances. Implementations of any of these `LicenseManager` methods can have access to the `Asset` within the `LicenseManager` to build appropriate request headers, query parameters or request bodies.

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. Overriding `LicenseManager` methods `executeKeyRequest` and `executeProvisionRequest` will allow you to highly customize the request and/or retrieve license keys and metadata from specialized responses.

It is recommended to allow the SDK to manage licenses for your application, and request renewed licenses for offline playback when the device is online and prior to license expiry. The method `setAutomaticDrmLicenseRenewalEnabled` on `ISettings` makes it possible to disable this automated update behavior if you need to manually request license updates within your application. If you do not need the SDK to process DRM at all then it is possible to not provide a license manager implementation to the SDK. Failure to configure a license manager will cause some warnings to the log file, but the SDK will then proceed without DRM.

Retrieve and Persist Widevine Licenses: Android

When downloading a Widevine-protected MPEG-DASH or HLS 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 an implementation of the `ILicenseManager` interface to the SDK. The easiest way to do this is to extend the `LicenseManager` class in the `com.penthera.virtuososdk.client.drm` package and override the `getLicenseAcquisitionUrl` method.

Implement your LicenseManager subclass

```
public class YourLicenseManager extends LicenseManager {
    @Override
    public String getLicenseAcquistionUrl() {
        String license_server_url = "https://proxy.uat.widevine.com/proxy";

        /*
         Here you can examine the mAsset or mAssetId member
         variables of the base LicenseManager implementation,
         and modify the license server url as needed:
        */
        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;
    }
}
```

Register your LicenseManager subclass for use by the SDK

For the SDK to use your License Manager implementation you need to override a metadata value in the AndroidManifest.

To override metadata you will first need to add the tools namespace to your android manifest:

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

Now add in the metadata to point the SDK to your license manager implementation (replace the value the fully qualified class name of your implementation):

```
<meta-data tools:replace="android:value"
    android:name="com.penthera.virtuososdk.license.manager.impl"
    android:value="com.yourdomain.yourapp.YourLicenseManager" />
```



LOW BATTERY IMPACTS VARIOUS FEATURES

The SDK battery threshold value in `ISettings` impacts the SDK download behavior, but device battery level impacts other SDK behaviors which are not controllable by the SDK's `ISettings` value. Various functions of the SDK are implemented as Android OS Workmanager tasks which are configured not to run whenever "battery is low". The battery level which Android OS Workmanager considers "low" is commonly set to 20% and is not under control by the SDK. Thus, independent of the SDK `ISettings` value you may see the following SDK functions delayed whenever the OS determines the device is in a low-battery state:

- Execution of Expiry actions
For assets with SDK expiry values the SDK will still prevent playback of expired assets, but the automatic removal of the expired asset files from disk may be delayed if device battery is low.
- Automatic refresh of Ads
- Timed refreshes of DRM (refreshes requested manually by your code will still occur)

Playout with Widevine Persisted License: Android

The ExoPlayer DRM implementation can vary substantially between ExoPlayer versions. The SDK provides support for multiple versions of ExoPlayer through additional libraries, allowing you to include the appropriate library in your app.

These new version-specific support packages include an `ExoplayerUtils` class which can use version specific code to initialize ExoPlayer for playback of Virtuoso SDK assets. They also contain version-specific DRM implementations (which can replace code in existing projects that was previously based on SDK demo code). The support packages include the class `ExoplayerUtils` which has methods that streamline creating DRM, `MediaSource`, `MediaItem` (ExoPlayer2.12), and fully-initialized ExoPlayer instances.



TIP

If you are migrating to the Penthera 4.0 Android SDK and had previously implemented a `DrmSessionManager` based on SDK Demo code from a pre-4.0 Penthera Android SDK, you may in most cases replace your implementation with the supported implementation of `DrmSessionManager` from the appropriate ExoPlayer support library.

The ability of the 4.x SDK to easily support multiple versions of ExoPlayer required changes to the Virtuoso SDK core DRM handling. Thus if you do not choose to migrate to the new support libraries, you will need to update your existing DRM related code as some Penthera SDK DRM classes have changed while others have been moved to new support packages.

ExoPlayer 2.11.x

Implementing apps will need to add the new Penthera Android ExoPlayer 2.11 support library manually, or by adding to their build.gradle dependency configuration:

```
implementation "com.penthera:cnc-exoplayer-2_11-support:4.0.0"
```

The debug library can be referenced in development as `com.penthera:cnc-exoplayer-2_11-support-debug:4.0.0`

ExoPlayer 2.12.x

Implementing apps will need to add the new Penthera Android ExoPlayer 2.12 support library manually, or by adding to their build.gradle dependency configuration:

```
implementation "com.penthera:cnc-exoplayer-2_12-support:4.0.0"
```

The debug library equivalent is `com.penthera:cnc-exoplayer-2_12-support-debug:4.0.0`

Legacy ExoPlayer and other players

If you need help integrating with older versions of ExoPlayer or with other players, please contact Penthera Support.

Migrating from Penthera implementations prior to Penthera 4.0 Android SDK

If you have existing code supporting DRM with a pre-4.0 version of the Penthera SDK, but your implementation did not require any customizations to the example DRM code we provided, then your class can be replaced entirely with our new support library implementation. The new library implementation handles all offline DRM with no need for the previous classes which were based on our demo code, so the new library class can be used as a wholesale replacement in your app code. The new class you would use depends on the version of ExoPlayer you use, either `com.penthera.virtuososdk.support.exoplayer211.ExoplayerDrmSessionManager` for ExoPlayer 2.11.x or `com.penthera.virtuososdk.support.exoplayer212.ExoplayerDrmSessionManager` for ExoPlayer 2.12.x.

Errors with DRM License Durations on Android 8

There is a known issue with Android 8.x devices which can cause the "license duration remaining" to be reported from the Android MediaDrm component as 0. On buggy devices the duration will report as zero even immediately after the offline license has been retrieved, when there should be ample duration left for playback, such that opening the license in the DRM framework and starting a player should otherwise result in fully functional playback.

Because the SDK is properly adhering to the erroneous value being reported by Android 8, if the SDK is being used for key management then by default it will stop playback in this case because it interprets the license to be expired. The Penthera SDK now provides a manifest meta-data option which can be declared true to request the SDK to bypass the checks in the circumstance of Android 8.x and license duration == 0, but where playback duration is greater than 0. The manifest option can be set as in this example:

```
<!-- Enable Penthera DRM fix for Android O -->
<meta-data
    android:name="com.penthera.virtuososdk.client.drm.android_o_license_allowed"
    android:value="true" />
```

**NOTE**

Note that on the subset of Android 8.x devices which exhibit this DRM bug, some versions of popular players (e.g., ExoPlayer and others) and/or their default DRM session components may *ignore* the zero license duration reported by Android, and may allow playback when technically they should prevent it. Depending on the vendor and version, you may find varying results from their player and default DRM session. This can be confusing to you as the app developer, because our SDK will *accurately* report a problem *even though such keys may still enable playback on some versions of popular players*.

The Penthera SDK relies in part on a valid remaining license duration to determine when to refresh DRM keys for maximum offline playback capability. Unless you choose to enable the workaround we have provided, on these buggy Android 8.x devices the Penthera SDK will continue to accurately report that the provided DRM key has no usable license duration remaining, and our SDK will prevent offline playback. On the problematic devices there is no way to receive a license which is technically valid for playback. If you need to support the problematic Android 8.x devices in your market, you will want to enable our provided workaround. With the workaround enabled, our SDK will substitute "*playback duration remaining*" for the missing "*license duration remaining*," and will make its DRM refresh decisions based on that value.

Extended Features

FastPlay: Android

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

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

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

For example:

```
MPDAssetBuilder mpdAsset = new MPDAssetBuilder()
    .manifestUrl(myManifestURL)
    .desiredVideoBitrate(1927853)
    .desiredAudioBitrate(0)
    .assetId(remoteId)
    .usesFastPlay(true)
    .addToQueue(false);

IAssetManager assetManager = mVirtuoso.getAssetManager();
assetManager.createMPDSegmentedAssetAsync(mpdAsset.build());
```

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

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

```
if (mpdAsset.fastPlayReady()) {
    URL fastplayAssetURL = mpdAsset.getPlaylist();
    //initiate playback using the fastplayAssetURL
} else {
    //fastplay not ready, so release that reference
    mpdAsset = null;
    //now initiate playback with the online manifest URL instead
}
```


That is all that is required. The `fastPlayReady()` test is sufficient to know whether FastPlay preparations are complete when your user presses a button to start playback.

Auto-download

The Auto-download feature allows you to define any set of content as a playlist, and as the user finishes with one item in the playlist, another item will automatically be downloaded for them. Using Auto-download is an easy way to keep desirable content available on your users' devices.

You might engage Auto-download when your user chooses to download any episodic content, or provide the option after they have watched an episode of a series. You may provide an opt-in or opt-out approach. However you integrate the feature with your user experience, it becomes easy for the user to start following a series of content.

For episodic seasons, workout videos, news segments, movie trilogies, and other serial content, the next item defaults to the subsequent episode. You can even append dynamically to a playlist as new episodes are released, so Auto-download works for active series as well as completed seasons & content. Regardless, it is easy to provide a simple toggle in your UI to enable and disable Auto-download.

While there are various configuration parameters to tweak Auto-download behaviors, our default settings cover most needs. The Auto-download feature does not require server-side integration with the Penthera Cloud. The required code is minimal and is only in the client app. If you have episodic content defined in your catalog you already have all the necessary metadata.

All your client app needs to do is:

- Register lists of Asset IDs with the `PlaylistManager` in our client SDK
- Implement the `IPlaylistAssetProvider` interface to allow the SDK to resolve each asset ID into a `VirtuosoPlaylistAssetItem` when it is time to download the next asset in the playlist
- Implement the `getPlaylistProvider()` method of an `IBackgroundProcessingManager` and register the `IBackgroundProcessingManager` with the SDK via `AndroidManifest.xml`, which is how the SDK accesses your `IPlaylistAssetProvider` implementation

Once a playlist is defined in the app, when an asset contained in the playlist is watched and then deleted, the next available asset *which is not already on the local device* is queued for download. When Auto-download is enabled, each watched & deleted asset in a playlist will queue another unless one of the following occurs:

- the user has reached the end of the playlist,
- the user cancels the next asset while it is downloading,
- the user deletes the asset without watching it,
- your app code calls our "delete-all" method (see note one below),
- the asset expires without being played (see note two below), or
- the user manually disables the Auto-download feature in your UI (if you give that option)

This leads to an intuitive experience for the user. In fact, if the user manually downloads more than one asset from the same active playlist, the SDK will seek to keep that many episodes of the playlist available locally.

If the user has watched the last item in a playlist, when you add a new item to that playlist the new episode will immediately queue for download, so the user can follow an active series with no further effort!

**NOTE**

Note one: In a future release the design will simplify such that even a delete-all can trigger Auto-download.

**NOTE**

Note two: Any expiry *without playing* the asset will never trigger an Auto-download. When a played asset reaches its expiry-after-*play* deadline, Auto-download can download an item from a playlist.

However, in this first release of the Auto-download feature if a played asset happens to reach its expiry-after-**download** deadline before its expiry-after-play deadline, Auto-download does **not** engage. In a future release the design will simplify slightly such that *any* expiry of a played asset can trigger Auto-download.

Implement the IPlaylistAssetProvider

The `getAssetParamsForAssetId(...)` method of `IPlaylistAssetProvider` allows the SDK to translate the desired episode Asset ID into the `VirtuosoPlaylistAssetItem` which will be used to instantiate the `IAsset`.

You will implement the provider method, implement a trivial `getPlaylistProvider` method of an `IBackgroundProcessingManager`, and register the `IBackgroundProcessingManager` within your android manifest. Once those pieces are in place in your app, your delegate code will be called to create assets in the playlist whenever appropriate.

First, the `IPlaylistAssetProvider`:

```

class MyPlaylistProvider : IPlaylistAssetProvider{
    override fun getAssetParamsForAssetId(assetId: String?):
VirtuosoPlaylistAssetItem {
        //this method is called by the SDK to request the asset parameters
        for an asset ID that is associated with a playlist
        //this is called when preparing to add a new item to the download
        queue for a playlist
        var ret :VirtuosoPlaylistAssetItem
        val assetInfo = MainActivity.ASSET_MAP[assetId];
        assetInfo?.let{
            val params = HLSAssetBuilder().apply {
                assetId(assetId)
                manifestUrl(URL(it.second))
                addToQueue(true)
                desiredVideoBitrate(Int.MAX_VALUE)
                withMetadata(it.first)//Here we supply a title.
            }.build()
            return
VirtuosoPlaylistAssetItem(Common.PlaylistDownloadOption.DOWNLOAD,params)
        }
        return
VirtuosoPlaylistAssetItem(Common.PlaylistDownloadOption.TRY_AGAIN_LATER,
null)
    }

    override fun didProcessAssetForPlaylist(asset: IIdentifier?,
playlistName: String?): IIdentifier {
        //this method notifies the app that an asset has been downloaded
        for a playlist
        return asset!!;
    }
}

```

Next, the IBackgroundProcessingManager:

```

class MyBackgroundProcessingManager : IBackgroundProcessingManager {

    ...

    override fun getPlaylistProvider(): IPlaylistAssetProvider {
        return MyPlaylistProvider()
    }
}

```

and finally, the Android manifest:

```

<meta-data
    android:name="com.penthera.virtuososdk.background.manager.impl"
    android:value="com.myco.myapp.MyBackgroundProcessingManager" />

```

Register a Playlist

Now that the SDK can use your IPlaylistAssetProvider to create assets whenever needed, you simply need to declare playlists when appropriate. To register a playlist with the SDK:

```

val plConfig = PlaylistConfigBuilder().apply{
    withName("MY_PLAYLIST")
    requirePlayback(true)           //true is also the default
    considerAssetHistory(false)    //false is also the default
    searchFromBeginning(false)    //false is also the default
}.build()

//create playlist with list of asset ids
virtuoso.assetManager.playlistManager.create(plConfig,
mutableListOf("assetId1", "assetId2", "assetId3"))

```

Once the playlist is registered, if the user downloads, watches and deletes episode one, the SDK will automatically download episode two, and so on.

Grow a Playlist

To add an episode to an existing playlist:

```

val playlist = virtuoso.assetManager.playlistManager.find("MY_PLAYLIST")

playlist.append("assetId4")

virtuoso.assetManager.playlistManager.append(playlist)

```



TIP

For active seasons of episodic content, use the append feature when a new episode is available. If the user is caught up on the existing episodes, the new episode will automatically download to their device!

Advertising Support: Android

Beta support for client- or server-side advertising definitions is present in the SDK, but is disabled by default. This prevents unnecessary setup and processing for customers who are not using this feature. The following meta-data statements can be added to the manifest to enable the feature:

```

<!-- Enable Penthera Ad support -->
<meta-data android:name="com.penthera.virtuososdk.adsupport.enabled"
    android:value="true" />

```



LOW BATTERY IMPACTS VARIOUS FEATURES

The SDK battery threshold value in `ISettings` impacts the SDK download behavior, but device battery level impacts other SDK behaviors which are not controllable by the SDK's `ISettings` value. Various functions of the SDK are implemented as Android OS Workmanager tasks which are configured not to run whenever "battery is low". The battery level which Android OS Workmanager considers "low" is commonly set to 20% and is not under control by the SDK. Thus, independent of the SDK `ISettings` value you may see the following SDK functions delayed whenever the OS determines the device is in a low-battery state:

- Execution of Expiry actions
For assets with SDK expiry values the SDK will still prevent playback of expired assets, but the automatic removal of the expired asset files from disk may be delayed if device battery is low.
- Automatic refresh of Ads
- Timed refreshes of DRM (refreshes requested manually by your code will still occur)

What Next?

Browse through the example projects in the Tutorials directory of the SDK distributions. The ReadMe files explain what capabilities are covered by each example. The examples are available in various programming languages, and can easily be built & run in your IDE. When running the examples, use the demo/development keys and URL we have provided you for the Penthera Cloud.

In the SDK distribution, look for additional API details in the code-level documentation (javadoc for Android, header docs for iOS).

The following publications may also be of interest. These can be read online in the Penthera ZenDesk instance, and are also available as PDFs.

201: Developing with the iOS SDK ([PDF](#), Penthera [online support](#))

202: Developing with the Android SDK ([PDF](#), Penthera [online support](#))

301: iOS SDK Beyond the Basics ([PDF](#), Penthera [online support](#))

302: Android SDK Beyond the Basics ([PDF](#), Penthera [online support](#))

203: Best Practices (Penthera [online support](#))

312: Known Issues (Penthera [online support](#))