penthera

Penthera 202: Developing with the Android SDK

Android SDK 3.16.2

July 24, 2020

Introduction to development of an application with the Penthera Android SDK 3.16.2. Covers architecture, accessing the SDK, fundamentals, setup, and basic feature implementation.

Copyright © 2020 Penthera Partners

CONFIDENTIAL - Licensed use only

Table of Contents

Copyright © 2020 Penthera Partners

Welcome to the Penthera SDK

Welcome to the Penthera SDK developer guide. If you have a basic understanding of the Penthera platform, and familiarity with mobile media apps in general, this guide should enable you to integrate our Penthera SDK into a mobile app and make use of its core features. When you are done with this material we have advanced guides to help you enable expanded features which may be useful to your company and your users.

We will start with a brief discussion of typical media ecosystem components, the architecture of the Penthera SDK and how it fits into that media ecosystem, and then proceed with how to integrate our SDK into your app. If you are entirely new to Penthera or to mobile media app ecosystems, you may benefit from reading our Penthera 101: Introduction to Penthera Development document before proceeding here.

Copyright © 2020 Penthera Partners

CONFIDENTIAL - For Licensed Use Only

Fundamentals of the Penthera SDK

The SDK dev guide will demonstrate some basics of using the SDK, including how to run the sample SDK app we provide, how to set up the SDK in your own development project, some fundamentals of how the SDK is configured, how your code can observe the state of the engine and its downloads, as well as some examples of the most common functions your app will use.

Demo code included in this guide and in the SDK demo app demonstrates common ways to use the SDK. This is just a portion of the overall SDK functionality. After you're done looking through the demo code, read our advanced topics and best practices guides, and have a look at the API itself to see what else is available.

Typical Integrations

The Penthera Virtuoso SDK and Penthera Cloud (sometimes referred to as our "backplane") participate within your ecosystem to provide Penthera's Download2Go features and more. Your team includes the Virtuoso SDK within your mobile app, where you integrate the SDK with your application code via local API calls. The Virtuoso SDK handles necessary communications with the Penthera Cloud.

Client applications usually integrate the SDK with other services which your infrastructure provides. These frequently include:

Media Player

The Virtuoso SDK is designed to support integration with your media player of choice. When your app is ready to play an asset which is in our SDK's managed asset download queue, or to play a Penthera FastPlay-enabled streaming asset, we provide easy hooks to retrieve the appropriate reference which your app provides to your media player.

We frequently encounter teams using standard players such as AVPlayer on iOS, ExoPlayer on Android, as well as Bitmovin player and others. The SDK provides wrappers which make it easy to get started with some of the standard players, as well as instructions on how to achieve deep integration with most any player you may choose.

Media Catalog

Your application is responsible for interacting with your media catalog, where it retrieves asset information for display in your user interface. When appropriate, such as to request an asset be downloaded for offline playback, your app uses the asset information to instruct our SDK to manage the download.

Content Distribution Network (CDN)

Virtuoso SDK is designed to transparently support various CDNs. Typically your app code retrieves asset information from your media catalog, then provides asset URLs and other details (e.g., desired bitrates and languages) to our SDK. This information is provided to our SDK when your code requests our download engine to manage the download, and our SDK retrieves the asset from the CDN. For asset types which use a manifest, our SDK retrieves the manifest from the URL you provide, parses it, and downloads the relevant components from the CDN.

User Authorization

Your application is responsible for interacting with your backend to perform any necessary user authentication & authorization. Your application code will register devices to the Penthera Cloud through our SDK, and will startup our SDK with an appropriate user identifier.

In support of various privacy regulations, we have no requirement for you to provide the Penthera SDK or Cloud with any user-identifiable information. Your code may provide our SDK with user and device identifiers which are only associated with real users within your own app code or authentication system. Our only requirement is that user and device identifiers are unique to those users and devices. The user ID you use to startup our SDK may even represent a family unit instead of an individual person.

Copyright © 2020 Penthera Partners

Analytics

The Penthera Cloud collects information about the usage of our SDK in your apps. This information is as anonymous as the user, device and asset IDs you provide to us. We provide some analytic information to you within our backend web interface, and can also provide the raw data to you for further analysis. We periodically look for patterns of anomalies in the backend data which can help us identify potential customer issues.

The Virtuoso SDK also allows you to send custom events from your app to our Cloud in order to augment the standard data we collect for you. Your app code can also retrieve many of our SDK events on the device, if you choose to push those direct from the device to your own analytic engine.

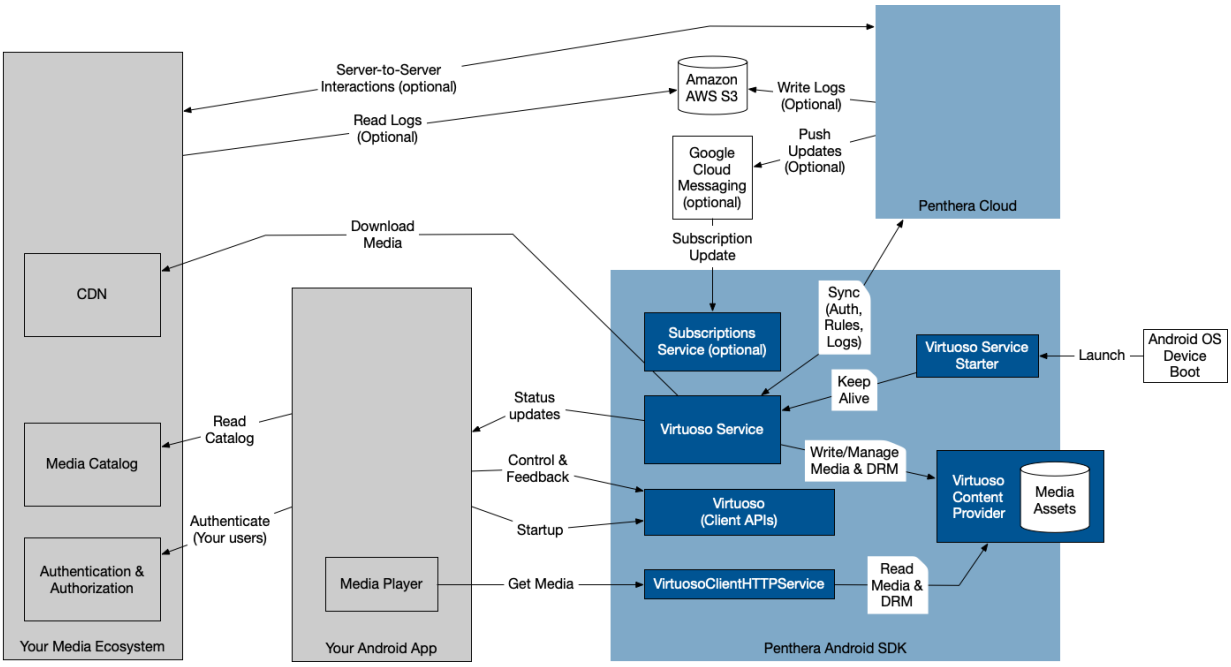
Digital Rights Management (DRM)

The Virtuoso SDK provides out-of-box support for various popular DRM systems, including Apple FairPlay and Google Widevine. The Virtuoso SDK automates the management of persistent/offline DRM keys so that your users' assets are available for offline playback according to rules embedded in the DRM keys and according to business rules you configure in our Cloud. Our SDK will attempt to renew DRM licenses at opportune times during connectivity to ensure that they are as up-to-date as possible

The SDK also provides several layers of APIs for coding to any custom DRM server or key management requirements. Your team has easy access in the SDK to make the most typical modifications required by third-party DRM servers, such as additional request headers, GET/POST parameters, base-64 encoding/decoding, and JSON request/response bodies. For even greater customization the SDK also supports powerful custom low-level access to the DRM interactions.

SDK Architecture: Android

This section provides an overview of the Android SDK architecture.



The SDK consists of seven components (for simplicity, not all components/interconnects are shown in diagram):

- **Virtuoso**. This is your app's main interface with the SDK. Provides interfaces to startup the Penthera SDK, register the user 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 Penthera Cloud services, and sends notifications to the enclosing app.

- **Virtuoso Service Starter**. A small Broadcast Receiver based class that starts the Virtuoso Service. This ensures that a valid Notification is available from the client app to allow the Virtuoso Service run as a foreground service. It also monitors environmental events to ensure the service restarts when required, which helps provide robust behavior.

NOTE

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

- **Virtuoso Content Provider**. Keeps track of events and all information regarding assets known to the SDK.
- **Virtuoso Client HTTP Service**. A proxy which supports playback of segmented assets, e.g. HLS videos.
- **Subscriptions Service**. Handles the synchronization of subscribed feeds, manages when new episodes should be added for download and deletes old episodes.
- **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.

WHY RUN AS SEPARATE PROCESSES?

The benefits of Virtuoso Service, Virtuoso Content Provider and Virtuoso Service Starter as separate processes are:

1. Running the main download service in a separate process ensures that it cannot be affected by any client side code. The service also has its own memory space, which can be important depending on the assets being downloaded as it sometimes uses a lot of memory buffers.
2. Running the content provider in a separate process from the Virtuoso Service or UI ensures that some potentially expensive / blocking database actions are isolated from the download process and user experience. It also ensures that the availability of the content provider does not keep another process in memory when it should not be required.
3. The Service Starter was historically placed in a separate process for two reasons: First to ensure that waking the SDK (using a broadcast receiver) did not require the entire service process to be created and kept alive, and second to ensure that if a crash occurred in the service then the service starter could restart it in the best manner. Originally designed for a much earlier version of Android, the benefits of this architecture are less relevant to more recent android OS versions, and this approach may change in the future.

Supported Mobile OS Versions

The Penthera SDK supports the vast majority of currently shipping mobile devices. At this time the officially supported mobile OS versions are:

- Android API 19 (Android 4.4) and Kindle Fire OS 5+ and greater

Earlier versions than these are not officially supported. If you wish to support earlier OS versions you may want to try earlier versions of our SDK, and current versions of our SDK may sometimes still work with recent unsupported OS versions.

We may attempt to answer questions about unsupported versions, but we cannot prioritize resources to resolve new issues discovered on unsupported versions which are not also present on supported versions.

Users and Devices

To Penthera a “User” is a person, household, or other entity that owns a device. When you call the start-up method on the Virtuoso SDK, your code must supply a UserID.

The SDK uses its own internal logic to assign a unique DeviceID to the device. The SDK uploads this pair (UserID, DeviceID) to the Penthera Cloud, which associates the Device with the User. The Penthera Cloud uses the user and device IDs to enforce business rules such as the “max download-enabled devices per user” setting.

The External Device ID is an optional field we maintain for your convenience. When set by your code, the external device ID will be reported to the Penthera Cloud with device registration and in device logs. You may wish to use this for an ID which matches that device in your own backend systems.

The Penthera Cloud provides a convenient mechanism to use your External Device ID in lieu of Penthera’s internal DeviceID. You can look up a device’s activity on the Penthera Cloud user interface using the External Device ID. You can even perform a “remote-delete” (single asset) or “remote-wipe” (all assets) from the Penthera Cloud using an External Device ID.

NOTE

Downloading becomes available on a device when the SDK is started. If you allow some users to download and others not, based upon their login credentials, it is a good idea to delay calling the SDK startup method until your enclosing app code has verified that the user is a paying customer and/or download-enabled user.

When your app detects that a customer has transitioned from being a download-entitled user to a non-download-entitled user, there are recommended two options. If this transition is permanent, call the unregister method in the SDK to remove the device from the account. This unregister will delete all previous downloads, remove the device from the user account on the Penthera Cloud, and remove the device from your Penthera billing calculations in future months. If the transition is temporary, call the shutdown method on the SDK. Once the SDK is shut down, the user will not be able to play downloaded assets, but the assets will not be deleted from the device so they can be immediately available again once the SDK is started again with their same user. The device will not impact billing if it remains in shutdown status through an entire billing cycle.

CAUTION

On Android there is no way to guarantee the internal device ID will remain the same if the user uninstalls and reinstalls the app.

Asset Identifiers

Upon creation, each asset is assigned an internal identifier for use with 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 allowing your code to provide an external string identifier for the asset, which is used to associate the asset with its ID in your catalog. The external identifier is required, and it must be unique to each asset.

How Downloading Works

The Penthera SDK proceeds through the download queue from top to bottom, attempting to complete each queued asset in order. For multi-segment assets, such as HLS and DASH, the engine will download multiple segments from within the asset simultaneously, but will focus on segments from one top-level asset at a time.

If a recoverable error occurs while downloading a file, the engine will immediately retry that file up to two more times. Immediate retries of the same file in the current downloading asset are known as the "inner retries". If the file continues to fail through each inner retry, the retry count for the asset is incremented by one, and the engine moves on to the next available asset. If a fatal, non-recoverable error occurs, the retry count is immediately set to its max.

After completing a pass through the entire queue, the engine will begin another pass if any items exist which might still be downloaded. These could include any assets whose retry counts were incremented in the previous pass, any assets which had non-permanent permission denials, or any new items added to the download queue. On each pass through the queue the engine will attempt again to finish any assets whose retry count has not reached the maximum number of attempts. These attempts are known as the "outer retries". The SDK may also retry failed assets at other opportune times, such as when the user returns to the app.

NOTICE

The default number of inner and outer retries is three, which we refer to as our "Rule of Threes."

If files of an asset had experienced errors, but on subsequent attempts files on that asset are successfully downloaded, the retry count for the asset is reset to zero. This helps to ensure that transient errors are less likely to cause a permanent asset failure. With many brief transient errors, such as in poor network conditions, the retry count on an asset may rise and fall repeatedly.

TIP

A method exists in the SDK to manually reset the retry count on an asset, which will cause the engine to attempt download of that item again without needing to delete it from the download queue and re-add it. In iOS this is the `clearDownloadRetryCountOnComplete:` method on `VirtuosoAsset`, and in Android this is the `clearRetryCount(assetId)` method on the `IQueue` interface.

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.

Coding with the SDK

This section will provide details on various aspects of coding with the SDK, including adding the SDK to your project, starting up the SDK, observing state, logging, and other topics.

Accessing Penthera SDKs on Github / Archiva

You have access to the Penthera repository at GitHub (https://github.com/penthera) or via Archiva (http://clientbuilds.penthera.com:8081/repository/releases) where you can access the released developer packages.

Download2Go Android SDK

The Android package contains:

- **CnCDemo** A demo app in Java and a demo app in Kotlin, which both run as-is on Android. These contain examples of some of the important SDK functions
- **Android Developer Guide PDFs**
- **java_docs** The java docs for the SDK, which are an important source of information about the SDK, especially some lesser-used features not covered elsewhere in documentation

TIP
The javadocs are an important source of knowledge which is frequently overlooked.

- **libs** A directory containing debug and release versions of our SDK for inclusion in your own project
- **Change List PDF** Release notes for SDK versions
- **README.md** High-level info about Penthera and the SDK

Running the Android SDK Demo

The Penthera Android SDK includes two versions of a reference Android app implementation, one written in Java (SdkDemo) and one in Kotlin (SdkDemoKotlin). 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 in a variety of formats, with and without DRM, all hosted by Penthera on Amazon Web Services (AWS).

To build and run SdkDemo or SdkDemoKotlin:

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. Update the `Config.java` and/or `Config.kt` file with the URL, public key and private key of your Penthera Cloud test or dev account.

NOTE
If you have not already received your test or dev account credentials, please contact <support@penthera.com>.

4. If you intend to test with the optional Push Notification support, modify the provided `google-services.json` and `api_key.txt` files to use Firebase Cloud Messaging and/or Amazon De-

vice Messaging. If you are not using Push Notification, remove the placeholder files and comment out or remove the relevant sections from the Android Manifest and the `gradle` build. These sections are detailed here: Push Notifications: Android.

5. Run or Debug the SdkDemo or SdkDemoKotlin module in an emulator or on a device. Testing on physical devices is preferred.

Congratulations! You now have a video-downloading app up and running. You are ready to develop your own apps with the SDK.

IMPORTANT

The Android demo app 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, you will need to use a different player. Penthera will be happy to help you with an implementation if needed.

Set up the Android SDK

We provide a step-by-step guide to quickly walk you through each required step of adding and configuring the Penthera SDK in an Android app. We also provide a detailed discussion of each portion of the configuration, which we recommend reading and understanding thoroughly.

Step-by-Step Android SDK Configuration Walkthrough

This section contains step by step instructions to create all the necessary components to work with the Penthera Android SDK. If you are interested in detailed explanations and alternatives, these are provided in Android SDK Configuration in Detail [16]. After following this step-by-step process, we recommend you read the remainder of Coding with the SDK [10] and then implement your first app features as described in Implementing Basic Features [44].

To configure the SDK you will perform the following simple steps:

- Add the SDK to an app, using gradle dependencies
- Declare SDK components in your AndroidManifest.xml
- Implement a simple ContentProvider, to provide a datastore for use by the SDK
- Optionally implement a simple class to describe your DRM server to the SDK, if you are using DRM.

Let's get started:

Step 1: Create “hello world” app. (Skip to step 2 if you already have an app)

1. Create a new Android Project.
2. Provide the application 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.

Step 2: Add libraries

In your project's `build.gradle` file you should include a reference to the Penthera maven repository:

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

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

```
implementation ('com.penthera:cnc-android-sdk:3.16.2 ')
```

TIP

During development it is best to use our debug version, which provides debug-level logging.

```
implementation ('com.penthera:cnc-android-sdk-debug:3.16.2 ')
```

Sync the project in your IDE to pick up the changes to your gradle config.

NOTE

AndroidX versus earlier support libraries

As of version 3.15.14 our SDK 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.

Step 3: Update permissions in 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"/>
```

Step 4: Add largeheap support

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">
```

Step 5: Declare the Virtuoso Service

Add the download service in the application section of the Android manifest:

```
<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>
```

Step 6: Declare the internal-use broadcast receiver

Add the receiver for internal-use broadcasts. The SDK will register the messages it needs to listen for, but you register the receiver itself in the manifest.

```
<receiver android:name="com.penthera.virtuososdk.service.VirtuosoService
$ServiceMessageReceiver"
          android:label="VirtuosoServiceMessageReceiver"
          android:process=":vservice"/>
```

IMPORTANT

The `android:process` of the receiver must match with the `android:process` you assigned to the VirtuosoService, so that this receiver will run in the VirtuosoService process.

Step 7: Declare the Client HTTP Service

The HTTP service sits between your app's media player and the downloaded files, providing the local files to the player during playback. For downloaded segmented assets, the HTTP service generates a manifest for the media player. By acting as an intermediary, this service allows the player to function the same as it would with online files, abstracts the location of the downloaded files, and enforces access policies.

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

Step 8: Declare the Service starter

The Virtuoso service starter is a broadcast receiver which 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 Android manifest:

```
<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>
```

Step 9: Implement a Content Provider for the SDK

The content provider enables the SDK to access the database within your app directory. Your app must implement the content provider for the SDK to use. This content provider implementation, together with the authority string and content provider declarations in the next two steps, give the SDK access to the database it will use for your media.

NOTE

Each app has its own database which is stored in the app's data directory. Android does not permit installation of a content provider if one already exists on the device with the same name or using the same authority, so these steps prevent any conflicts by allowing your app to tell the SDK what provider it should use with your application.

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 content provider we ship in the SDK. Click on **Browse...** beside the Superclass field and select `VirtuosoSdkContentProvider` as the superclass.
4. Click **Finish**. Your generated class should look similar to this:

```
package com.demo.myvirtuosoapp;

import
com.penthera.virtuososdk.database.impl.provider.VirtuosoSDKContentProvide
r;

public class MyVirtuosoAppContentProvider extends
VirtuosoSDKContentProvider {
     @Override
     protected String getAuthority() {
         // TODO Auto-generated method stub
         return null;
     }
}
```

Within the class you just created, implement the `getAuthority` method, and also provide a `static` implementation which sets the Authority to be used for this content provider. The same authority string is used in both places, and 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.virtuoso.content.provider");
   }

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

Step 10: Declare the content provider and its authorities

Now that you have implemented a content provider, it needs to be declared within the application element of the Android Manifest. This `provider` element links the class name of your provider (implemented in the previous step) with the authority string needed to access it. The authority string in this `provider` declaration must match the one used within your content provider implementation.

```
<provider
android:name="com.demo.myvirtuosoapp.MyVirtuosoAppContentProvider"

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

Step 11: Declare the authority metadata

This meta-data name-value pair allows the Virtuoso service to look up what authority string it should use to gain access to your content provider. The metadata attribute name must be `com.penthera.virtuososdk.client.pckg` so the Penthera SDK can find it, and the value you provide is the authority string the SDK should use. This must match the authority string you used in the previous steps. Only when the authority string matches in all three places will the SDK be allowed to use the app database.

```
<meta-data android:name="com.penthera.virtuososdk.client.pckg"

android:value="com.demo.myvirtuosoapp.virtuoso.content.provider" />
```

Copyright © 2020 Penthera Partners

OPTIONAL Step 12: Declare a DRM license manager and provide it to the SDK

If you are securing your assets with DRM, the SDK requires you to provide a LicenseManager implementation so it knows how to work with your DRM server. You must implement a subclass from `LicenseManager` and register it with metadata in the Manifest so the SDK can find it.

This example implementation is configured 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 could modify the license server url as needed.
      For example, you could access any mAsset and mAssetId member
variables:

        String video_id = null;
        if (mAsset != null) {
          video_id = mAsset.getAssetId();
        } else if (mAssetId != null) {
          video_id = mAssetId;
        }
        if(!TextUtils.isEmpty(video_id)) {
          license_server_url += "?video_id="+video_id +
"&provider=widevine_test";
        }
    */

    return license_server_url;
  }
}
```

The SDK will look up the class name of your License Manager implementation under the metadata key `com.penthera.virtuososdk.license.manager.impl` in the AndroidManifest. To override this meta-data value you will need to add the `tools` namespace to your manifest xml, and then use the `tools:replace` function to declare the name of your `LicenseManager` subclass.

```
<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"/>

</manifest>
```

Android SDK Configuration in Detail

This section will explore each of the configuration elements of the Penthera SDK in detail, specifically the necessary and optional AndroidManifest.xml elements, as well as some related code examples. We recommend a thorough grasp of the material in this section before integration & debugging with the Penthera SDK in your app.

Accessing the Android SDK with Maven

In your project's `build.gradle` file you should include a reference to the Penthera maven repository:

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

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

```
implementation ('com.penthera:cnc-android-sdk:3.16.2 ')
```

TIP

During development it is best to use our debug version, which provides debug-level logging.

```
implementation ('com.penthera:cnc-android-sdk-debug:3.16.2 ')
```

Sync the project in your IDE to pick up the changes to your gradle config.

NOTE

AndroidX versus earlier support libraries

As of version 3.15.14 our SDK 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.

App Permissions: Android

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.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.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.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 ver-

sions 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: Android

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.m
yvirtuosoapp.identifier" />
```

Foreground Download Service: Android

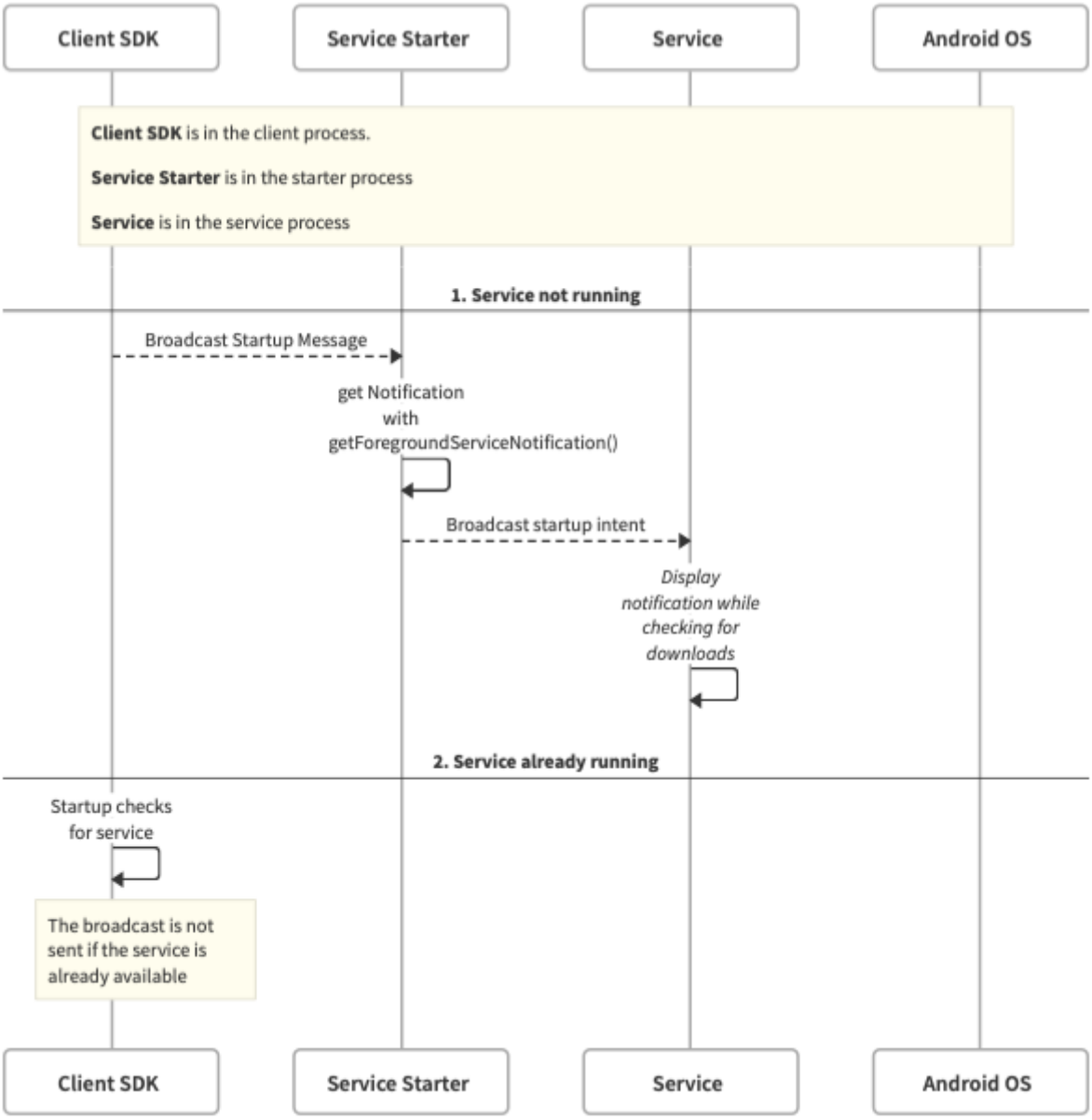
The download capability of the Penthera SDK runs as a *foreground service* process in Android. It is required to be a foreground service so that it may continue downloads, communications with the Penthera Cloud, or expiry processing while the UI portion of the app is in the background.

Because the download capability runs in its own process, separate from the main app process, there are important implications to understand about the download service process notifications. One implication is that an initial OS-level notification must be presented to the user before the foreground service is allowed to run. Another implication is that subsequent notifications may cross process boundaries, which is worth consideration within your app architecture and design.

Notification Before Service Startup

Android requires that a notification be displayed to the user for any foreground service being launched, otherwise the OS will quickly kill the foreground service. Thus to ensure the download service will stay running as a foreground service the Penthera SDK requires that a notification is created before it will start the download service. Startup of the SDK download service is performed by the SDK's Service Starter component, including enforcing that the required notification exists. The Service Starter will then provide the notification to the download service, which calls the Android OS API to present the notification to the user. This satisfies the OS requirement, which allows the download process to continue running as a foreground service even when the main app is backgrounded.

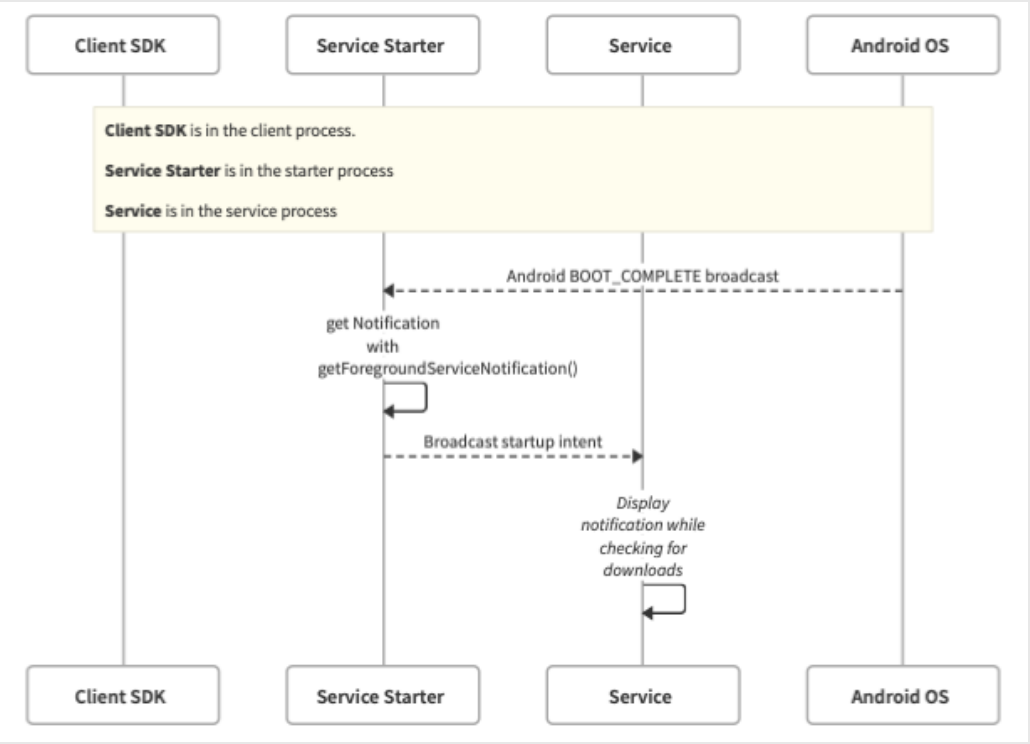
The Service Starter will ensure the existence of the notification every time the SDK tries to start the service, or whenever it checks that the download service is running, by calling your implementation overriding `getForegroundServiceNotification`. This may occur at device startup, during app startup, during processing of push messaging, or while processing expiry while the app is not open.



SDK startup ensures Service is running and user notification exists

NOTE

With proper configuration of the Service Starter in your AndroidManifest.xml, booting the Android device will message the Service Starter, which will start the download service to ensure downloads continue even after a device reboot. For an example see the Service Starter configuration in Step 8: Declare the Service starter [13]



Android OS boot should start the SDK Service

It is important that your implementation of `getForegroundServiceNotification` always returns a valid notification when called. Each request the SDK makes to that method includes an Intent which details the action taking place, and optionally also an asset. On initial startup the Intent may be Null. Your implementation of the Service Starter may return the same notification instance each time, if you choose, but should never return null.

CAUTION

The Service Starter passes the notification to the Service across a process boundary. The Android OS imposes limits on the size of data crossing the process boundary, so be cautious with the size of any images used in this startup notification.

NOTICE

The requirement that the notification related to foreground service startup originate within the app process is unique to that one notification, and is not required of any other notifications used by or generated by the Penthera SDK.

Copyright © 2020 Penthera Partners

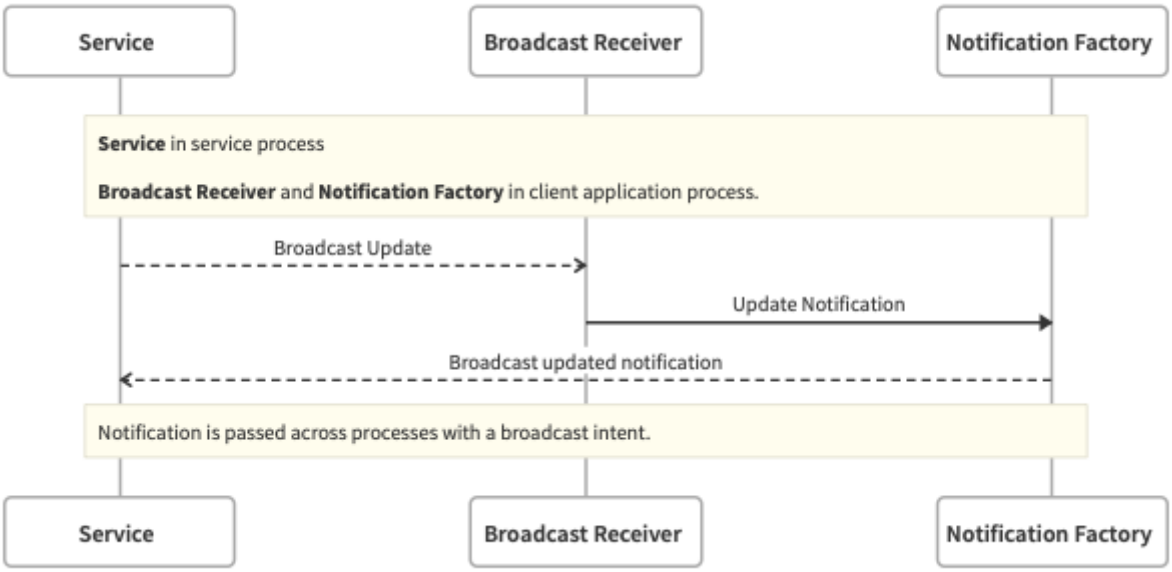
Status Update Notifications From The Service

As the download service makes progress, your user can be notified via updates to the original notification, even when the the main app UI is in the background or stopped. This allows your user to be notified when their download(s) complete, or with other status updates. There are two ways to achieve this with the Penthera SDK, and both options allow your code to define what appears in the notification. To choose between the two approaches you will want to consider process boundaries as well as what you would like to show the user. Option one makes it easy to produce a highly customized notification, but comes with some interprocess concerns and constraints. Option two can be simpler, but using it to generate highly customized notifications comes at a cost to memory efficiency.

Because the user notification is updated by the Service, both options involve your code receiving a status update from the Service, converting that into a notification intent, and returning that notification intent to the Service so it can be displayed. The significant difference between the options is *where this code resides and executes*. Option one relies on the Service Starter to provide the notification to the Service, just as it did with the initial notification. With option two you implement a provider interface which executes within the Service itself to convert status updates into a notification intent. Notice that option one involves crossing process boundaries, while option two runs entirely within the download service process. Let us consider the implications.

User Updates: Option One

With option one your AndroidManifest.xml registers a `BroadcastReciever` to receive certain status updates from the Service in the form of broadcast intents. When the Service generates a status update broadcast for which your receiver is registered, the broadcast is delivered to your manifest-declared receiver. A manifest-declared receiver runs in your main app process, so if your app is not already running when the broadcast occurs, Android will launch an app process to receive the broadcast. Because the receiver is running in your main app process, it has access to everything your main app has access to, including catalog images and metadata. Your broadcast receiver can build an attractive and sophisticated user notification `Intent`, possibly even including a thumbnail poster image for the asset. The Intent is then passed to the `updateNotification` method of your Service Starter class, which in turn passes the notification Intent to the Service, and the Service delivers it to the OS for display to the user.



User updates via Service notifications, Option One

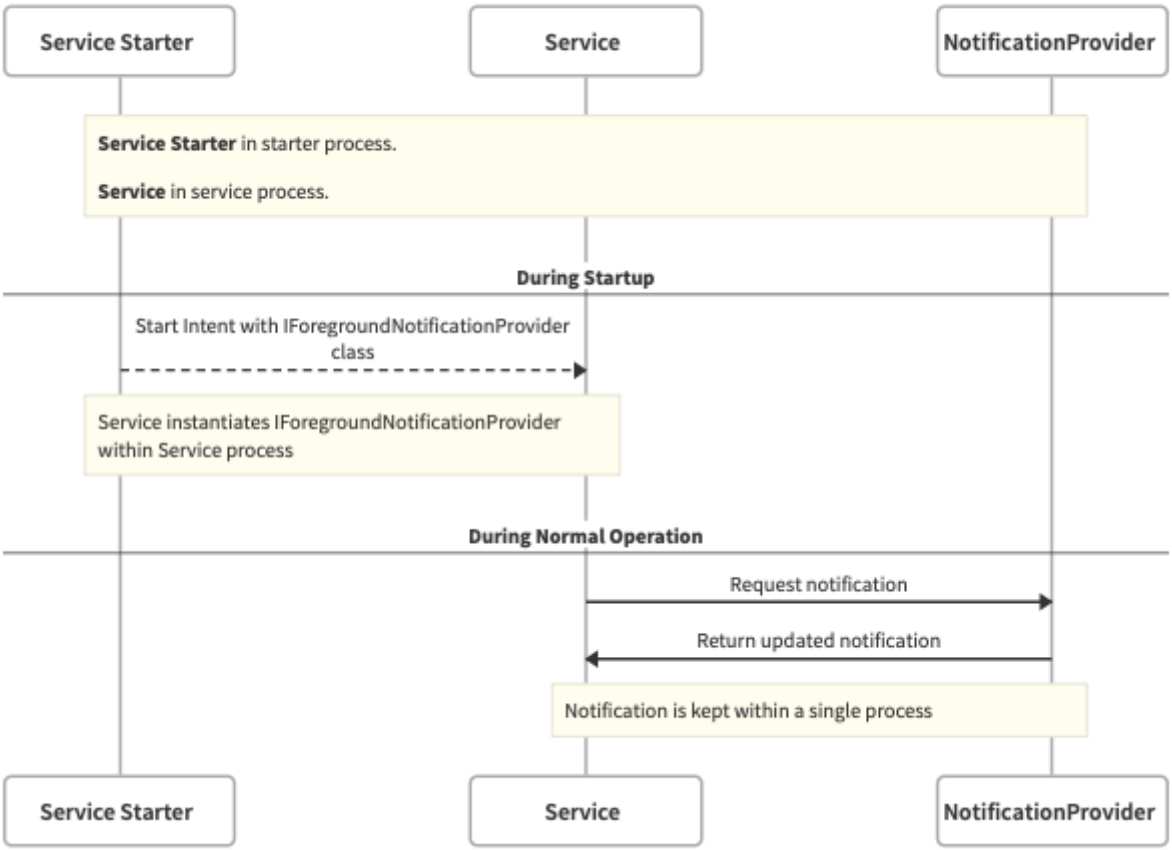
Notice that in this option your notification Intents are passed from the Service Starter to the Service, which means the notification crosses over a process boundary. This is readily achievable because notifications implement the Parcelable interface, however the important thing to note is that *there are limits to the size of objects passed across the process boundary.* This limits the maximum size of any images

you include in the notifications. Note also that if your app was not already running, an app process is launched by the OS to handle the broadcast, which means an application class or any dependencies such as a catalog which are used while handling the broadcast will be instantiated even if the user had previously quit the app.

NOTE
While the general trend in Android app design is to eliminate manifest-declared broadcast receivers in favor of context-declared receivers, the use case of a long-lived service whose status updates should be received and converted to user notifications *even if the app is not in foreground* is one of the special cases where manifest-declared receivers are still useful.

User Updates: Option Two

With option two you do not need to register a `BroadcastReceiver` to participate in the process. Instead, you will implement our `IForegroundNotificationProvider` interface to process Service status updates Intents into Notifications. Your IForegroundNotificationProvider implementation will determine which status updates are converted to user notifications, and it will perform the conversion. You configure the SDK to use your IForegroundNotificationProvider implementation for processing Service status updates by returning your implementation's Class from the `getForegroundServiceNotificationProvider` method of your Service Starter. This method overrides the base `VirtuosoServiceStarter` implementation, which otherwise returns null.



User updates via Service notifications, Option Two

Notice that with option two, user notifications are generated *within the Service* using your implementation of `IForegroundNotificationProvider` . This means that the user notifications do not get passed across any process boundaries, thus they *will not experience any size limits*, and they can be lazy-loaded in the Service rather than constructed elsewhere, serialized, passed, and de-serialized in the Service. This approach is faster and more efficient than passing broadcasts across process boundaries. However, your provider will be instantiated within the Service process, *along with any dependencies it creates*. It is important to bear this in mind as generating complex user notification Intents within the Service can result in more classes being instantiated in the Service process, duplicating more of those which are also used in the main app process, which can increase the overall memory footprint for the application.

NOTE

Even though with Option Two you are not required to create a manifest-declared broadcast receiver for the purpose of creating user notifications of background download progress, you may still wish to create a broadcast receiver to receive status updates from the download service for other reasons. For example you may wish to receive broadcast updates in your app to collect messages for a custom analytics engine. However, if you have taken the approach of Option Two, your app code has the option of using manifest-declared or the preferred context-declared receivers for UI support.

Note also that for the purposes of updating your UI elements with download progress, it is recommended to use the associated content provider or the SDK's queue observer rather than receiving broadcast messages for that purpose.

Broadcast Receivers: Android

The Penthera SDK utilizes broadcast messages in several ways. The SDK uses broadcasts internally to pass events between its download process and its components in the main app process. The SDK receives broadcasts from the Android system to support some of its functions, such as ensuring downloads resume after a device reboot. Depending on the design option you selected in the section Foreground Download Service: Android [18] you may have a broadcast receiver declared which updates your user with notifications during background downloads. The SDK also declares a number of events in its public interface which your app can receive as broadcasts if useful.

The following receivers may be required or otherwise present in your manifest:

Internal Use Receiver

Internal-use messages are not for use by your app code. They are generated and handled within SDK code, and the only requirement from you as the app developer is to ensure that the AndroidManifest.xml contains the appropriate receiver declaration so the SDK's internal-use receiver runs within the SDK's Service process (foreground service). The manifest should declare the receiver as shown in Step 6: Declare the internal-use broadcast receiver [13].

Service Starter Receiver

The SDK uses a Service Starter implementation (derived from the `VirtuosoServiceStarter` class) to receive broadcasts from the Android system, such as when the device is booted or when the app is removed from the device, and also to receive some internal-use SDK broadcasts. In the SDK Demo's `AndroidManifest.xml` you will find the Service Starter declared to receive its required broadcasts, such as `android.intent.action.BOOT_COMPLETED`, `android.intent.action.PACKAGE_REMOVED`, and `virtuoso.intent.action.START_SERVICE`, among others. Ensure that your app's

manifest declares the Service Starter receiver as shown in our example, as failure to receive the necessary events will impede some functions of the SDK.

User-Notification Receiver

You may have chosen the design option in the section Foreground Download Service: Android [18] which requires a broadcast receiver in order to update your user with notifications during background downloads. See that section of the documentation for further details.

App-use Receiver

Most of the broadcast messages exposed by the SDK for potential use in your app are related to events your code can also receive via observers (see Using Service Observers: Android [32]) or through associated content providers. In most cases the observers and content providers are the *preferred* mechanism, as the broadcast messages are not intended to be used for updating the application UI. One potential exception is to receive these SDK broadcast messages so that your app can pass events to a third-party logging or analytics system. Whatever the reason, for your app to make use of the SDK's public broadcasts the appropriate receivers must be declared either in context or within the application manifest. See Using Broadcast Receivers: Android [36] for details of what broadcasts are available, and how to receive them.

NOTE
Remember that since Android API 26, implicit android broadcasts are no longer permitted within applications.

WorkManager Time-based Scheduling: Android

The SDK uses the Android WorkManager for scheduling timed actions such as expiration. It is declared using the `androidx.work` package.

The SDK starts the WorkManager 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 customization 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.

IMPORTANT
The default WorkManagerInitializer is disabled by the manifest of the SDK to avoid unintentional changes to the default implementation flags without consideration. If you have a need to use WorkManager in your app, please coordinate with Penthera so we can help ensure a configuration that works for your app as well as for the Penthera SDK.

Custom Work Manager

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. Content providers are initialized in an order which may be specified in a manifest, or otherwise is determined by the operating system. If your content provider starts the work manager before the SDK work manager initializer runs, then the SDK's work manager initializer will use your already-running Work Manager instead of starting one. If your application requires a workmanager initialization with different parameters than the SDK work manager initializer would use, then your work manager Content Provider must be defined and the `initOrder` attributes set to ensure that your work manager starts before the SDK work manager initializer runs.

Lines as below in the application manifest can be used to set the SDK `initOrder` to the value ofyour choice, in this case, 2:

```
<provider
       android:name="com.penthera.virtuososdk.data.imple.provider.VirtuosoWorkManagerInitializer"
       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"
/>
```

Allow Playback of Segmented Assets via localhost on Android API 28+

When assets downloaded to the device are played back, they are delivered to the player via a localhost connection (at 127.0.0.1) from an HTTP proxy server run by the SDK. This allows the player to treat the offline content exactly the same as it would from an online URL. The proxy delivers content over unsecured HTTP, because delivering content locally to the player over HTTPS would use additional processing and battery resources, and would require the player to validate a self certification. The proxy is designed to only deliver content to localhost, so it cannot be used to stream the assets outside of the device.

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

In your Android Manifest:

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

Security Config XML (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>
```

Instantiating the Virtuoso Engine on Android

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 interacting with the Penthera SDK has one instance of Virtuoso, but you can use multiple instances within an activity, and your application can even hold a singleton reference for use everywhere, if required.

If a singleton instance is going to be used within the application, and stored in an Application derived class, then it is highly 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 which runs in a separate process, and that an Application class is created within each process of an application. Therefore, if you create the Virtuoso object within the Application it will result in creation of a Virtuoso instance within each process, including *within the download service process itself*. This is unnecessary, and 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());
}
```

Now that you know how to instantiate the Virtuoso object, read Engine Startup: Android [26] to learn how to use it to log your user's device into the Penthera Cloud. Also ensure you understand the implications of Moving To and From Background [28].

Engine Startup: Android

Before the Virtuoso engine can perform any meaningful task (download files, process events, receive subscription notifications), your app must cause the engine to register with the Penthera Cloud. This registration is also known as "startup" of the engine. Typically you will want to carry out this registration within a splash activity (if your users all have access to Download2Go features) or in a login activity (if you only provide Download2Go features to logged-in users, or to users of a certain tier).

NOTE

After you license the Download2Go SDK, you will be provided with a URL and keys specific to your Penthera Cloud instances (development/staging, and production). These URLs and keys will be used within your app to identify your app to the backplane when starting the engine. For building a proof of concept app, you are welcome to use a shared Penthera-hosted cloud instance; ask Penthera for the URL and credentials to that instance.

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

- `aPentheraCloudUrl <String>`: URL of your Penthera Cloud instance for development or production.
- `aUser <String>`: Identifier (which you assign) to register the user on the Penthera Cloud. This identifier is limited to 512 characters. It is common practice that the identifier provided to Penthera is not traceable to your user's actual identity or to their account with your company, except by cross-reference within your own identity management system.
- `aExternalDeviceId <String>`: Optional device ID defined by your app. Useful for associating each user device in Penthera Cloud with a device record in your own identity management system or your own cloud user experience.
- `aPublicKey <String>`: Key used to identify your app on the Penthera Cloud (provided to you by Penthera, and also available within your administrative web console on the Penthera Cloud)
- `aPrivateKey <String>`: Key used to sign all communications between the SDK and Penthera Cloud (provided to you by Penthera, and also available within your administrative web console on the Penthera Cloud)
- `aPushRegistrationObserver <IPushRegistrationObserver>`: Optional listener to monitor if registration for push messages through FCM or ADM messaging was successful. Refer to Push Notifications: Android for more details.

NOTE

After authenticating a user with the backplane, the SDK will reset if you call startup again on the device with a different user. All media for the old user is deleted and all the settings are reset to their defaults.

TIP

It can be beneficial to store your URL and keys somewhere securely in the cloud, and retrieve them at runtime for use within your app. This would enable you to change them without an app release if you ever desired, and can help keep them more secure.

An example of startup within a login activity:

```
private Virtuoso mVirtuoso;

public void handleSuccessfulLogin() {

  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 */ }
}
```

DEVICE STARTUP & SYNCING MAY IMPACT BILLING

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

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

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

Moving To and From Background

Once the Virtuoso engine is started, our SDK code running in your main app process instantiates internal observers which listen to various state updates from the download engine's foreground service process. When your main app process moves to the background it is undesirable for those internal observers to continue running. In order for the SDK to disconnect those observers it needs your app code to tell the SDK when your app goes to background and when it has returned to foreground. Calling the Virtuoso SDK `onResume` and `onPause` methods will ensure that internal UI-specific functions of the SDK are paused when your app UI is backgrounded and restored when your app UI is brought to the foreground again. Downloads and other non-UI features of the SDK continue even though the `onPause` is called.

Whether you have per-Activity instances of Virtuoso, or an app-wide singleton instance, you should ensure that your app code calls the SDK's `onPause` method on those instances before moving into the

background, and calls the SDK's `onResume` method on any instances when the SDK is needed again in the foreground. The simplest way to achieve this is within any activity which accesses a Virtuoso instance, call the Penthera SDK methods `onResume` and `onPause` from the similarly-named Android Activity lifecycle methods, *onResume* and *onPause*., or within methods annotated `@OnLifecycleEvent(Lifecycle.Event.ON_RESUME)` and `@OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)`.

onResume

Calling `onResume()` on the Virtuoso instance when your app is in the foreground ensures that the SDK's internal observers are correctly linked to the relevant services. In the onResume implementation of your Activity, you should call our onResume() method. If you are using any custom activity-specific observers these should also be registered with the Virtuoso instance at this time:

```
@Override
protected void onResume() {
        super.onResume();
        mVirtuoso.onResume();
        mVirtuoso.addObserver(myCustomBackplaneObserver);
        mVirtuoso.addObserver(myCustomEngineObserver);
        mVirtuoso.addObserver(myCustomQueueObserver);
}
```

NOTE

With AndroidX you would take similar steps within methods marked:

```
@OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
```

CALLING ONRESUME FROM THE BACKGROUND THROWS AN EXCEPTION

The SDK `onResume` method attempts to start the HTTP proxy service so it is ready for playback, but the HTTP proxy service can only be started successfully when the application is in the foreground. An `IllegalStateException` will be thrown if `onResume` is called when the app is in a background state. Recent versions of the Penthera SDK protect against this exception crashing your app or otherwise bubbling up into your app code. In the event that such an exception is thrown within the SDK while trying to start the HTTP proxy service for playback, the SDK will try again when a playback URI is requested.

Furthermore, there is a known issue in the application framework of Android 9.0 where the Activity `onResume` method can be called by the OS prior to the app being put into foreground mode. Left unchecked this would result in the `IllegalStateException`. As stated above, recent versions of the Penthera SDK protect against this issue, and the underlying issue is itself fixed in later android versions. You can read more about the Android issue here: https://issuetracker.google.com/issues/113122354

Be aware that if any of your `onResume` code is not safe to run in the background, it is recommended practice to wrap that code with a try/catch block if your app will run on Android 9.0 or if your app implementation presents any other risk of calling the SDK `onResume` when the app is in the background.

onPause

Call `onPause` on the Virtuoso instance when your app or Activity is about to move to the background, so the SDK can unregister any internal observers it has linked to services. Although not strictly necessary, it is good practice to unregister any of your own observers before calling `onPause()`:

```
@Override
protected void onPause() {
       super.onPause();
       mVirtuoso.removeObserver(myCustomEngineObserver);
       mVirtuoso.removeObserver(myCustomBackplaneObserver);
       mVirtuoso.removeObserver(myCustomQueueObserver);
       mVirtuoso.onPause();
}
```

NOTE

With AndroidX you would take similar steps within methods marked:

```
@OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
```

ONPAUSE() UNREGISTERS YOUR OBSERVERS

Calling `removeObserver(...)` on a Virtuoso instance is the recommended way to unregister a custom observer you create, but calling the `onPause` method on a Virtuoso instance will also unregister any custom queue, engine, backplane or subscription observers you may have registered on that instance.

Even if you choose to rely on the Virtuoso `onPause` method instead of making explicit calls to `removeObserver(...)` for each of your observers, any custom observers used in your Activity must be added to the Virtuoso instance with a call to `addObserver(...)` in the Activity's *onResume* implementation to ensure they are attached each time your user opens the Activity.

State Management: Android

Pausing or resuming all downloads can be performed through method calls on the `IService` interface. This is also a source for the general state of the Virtuoso Engine, and for information about the connection to the Penthera Cloud.

Beyond these top-level engine states available upon request from IService, there are two ways to receive messages about a variety of state changes and events in the Penthera SDK. Observers are typically the preferred method, and the other is receiving Android Broadcasts.

Familiarize yourself with the two approaches, with what events are available through each, and select the appropriate mechanisms for your use cases. A blend of approaches may be appropriate, rather than using only one or the other.

IService Interface: Android

Instances of the `IService` interface can be created using the `getService()` method on your `Virtuoso` instance. An instance of the `IService` interface allows you to:

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

IMPORTANT

The `IService` is **not** a singleton. Each call to the `getService()` method will return a *new instance*. Note that in Kotlin the call is shortened to `.service` which looks like a property accessor, but will still return a new instance each time it is called.

To perform `IService` functions the SDK must first establish a bound connection to the download service process from within the main app process where your code is running. This connection is made when your code calls the `bind()` method of `IService`. The bind call is required before other `IService` methods will function. You should also call `unbind()` when you are done with the `IService`, `such as in the onPause method of an Activity`.

```
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();
}
```

The `getStatus()` method of `IService` provides an `int` which corresponds to values in `Common.EngineStatus`. Check the javadoc for complete codes, but these `EngineStatus` values include:

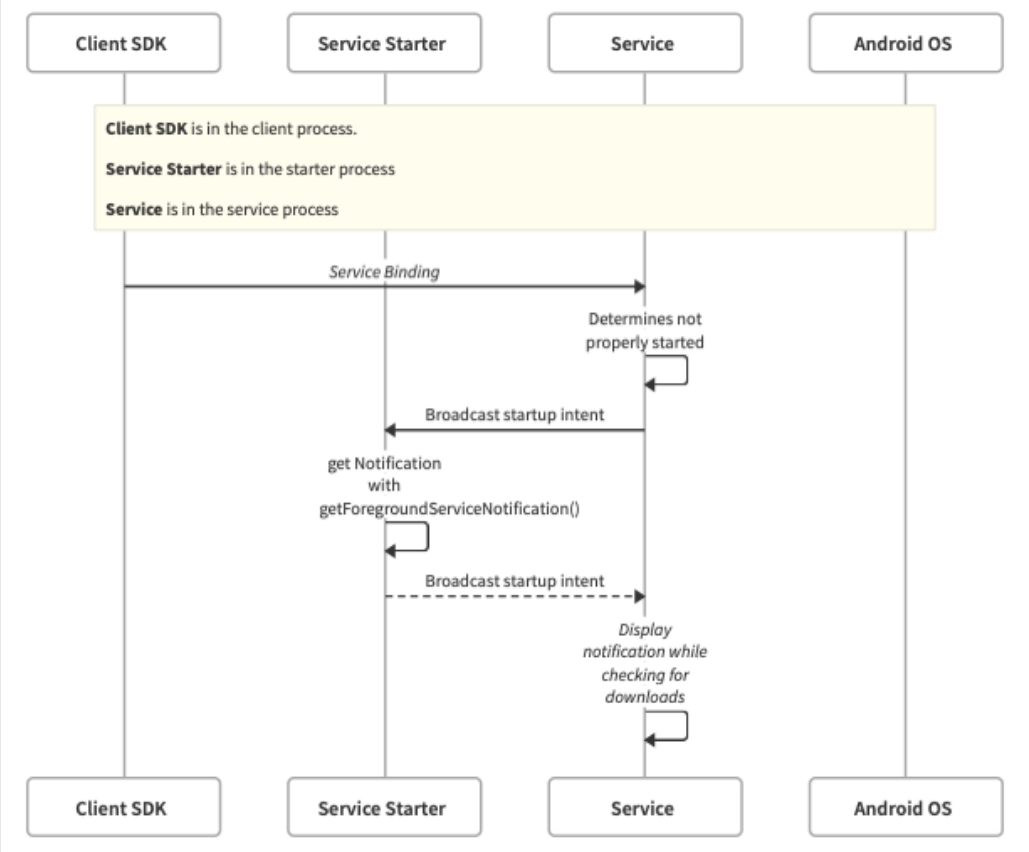
- Common.EngineStatus.IDLE
- Common.EngineStatus.DOWNLOADING
- Common.EngineStatus.PAUSED
- Common.EngineStatus.BLOCKED
- Common.EngineStatus.ERROR
- Common.EngineStatus.DISABLED

Copyright © 2020 Penthera Partners

- Common.EngineStatus.AUTH_FAILURE

NOTICE

Notice there is no engine status of "not running." When the SDK `bind()` method is called on `IService`, it will start the foreground Service process if it is not already running. In extreme operating conditions the Android OS might terminate the Service, so if it has not already been restored the SDK restores the Service when it is needed by your main app process.



IService binding ensures the Service is running

Using Service Observers: Android

The SDK provides several observers for use within your Activity and app. The observers allow your application to get updates from the SDK, which among other things helps keep your views refreshed.

Observer	Description
`IBackplaneObserver`	Observer notified when any communication with the Penthera Cloud (the "Backplane") has completed. Provides a callback method which receives the type of communication, the result code, and any error string.
`IEngineObserver`	Observer notified of changes in the download engine status, changes to the settings used by the service, and when assets are deleted or expired.
`IQueueObserver`	Observer notified of changes for the queue or assets within it. Notifies when downloads start and end, when a download error occurs, and any queue changes.
`ISubscriptionObserver`	Observer notified when communication with the Subscription Service of the Penthera Cloud completes.

Observer	Description
`ISegmentedAssetFromParserObserver`	Observer notified of actions when a segmented asset manifest is parsed and the asset created.
`IPushRegistrationObserver`	Observer notified when Push Messaging registration takes place.

In the following sections we will give more detail about three core observers you are likely to implement: `IBackplaneObserver`, `IEngineObserver`, and `IQueueObserver`.

Your implementations of any of these three observers can be registered with the Virtuoso object through the `addObserver(...)` method, and removed with the `removeObserver(...)` method.

IMPORTANT

When using observers in your Activity, it is important to ensure your observers are not left registered with the Virtuoso instance when the user leaves the Activity. The best practice is to make any `addObserver(...)` calls in your Activity `onResume()` implementation, and remove each observer in your Activity `onPause()` implementation with a call to the Virtuoso object's `removeObserver(...)` method.

ONPAUSE() UNREGISTERS YOUR OBSERVERS

Calling `removeObserver(...)` on a Virtuoso instance is the recommended way to unregister a custom observer you create, but calling the `onPause` method on a Virtuoso instance will also unregister any custom queue, engine, backplane or subscription observers you may have registered on that instance.

Even if you choose to rely on the Virtuoso `onPause` method instead of making explicit calls to `removeObserver(...)` for each of your observers, any custom observers used in your Activity must be added to the Virtuoso instance with a call to `addObserver(...)` in the Activity's *onResume* implementation to ensure they are attached each time your user opens the Activity.

Details of `IPushRegistrationObserver` are covered further in Push Notifications: Android, `ISubscriptionObserver` in Subscriptions: Android, and `ISegmentedAssetFromParserObserver` in Observing Manifest Parsing Results with ISegmentedAssetFromParserObserver: Android.

Also, see the SdkDemo source code for example uses, and it is often useful to read the description of these observers in the javadocs included with the SDK.

TIP

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, which allows your observers to extend from the base implementation and override only the methods you require.

IEngineObserver: Android

The Virtuoso engine notifies any registered `IEngineObserver` of changes in the download engine status, changes to the settings used by the service, and when assets are deleted or expired.

An `IEngineObserver` can be registered with the Virtuoso object through the `addObserver(...)` method and removed with `removeObserver(...)`.

Review the Virtuoso SDK javadoc for the `Observers` class to find complete documentation covering the `IEngineObserver` interface. The status codes passed to each observer method are explained in the javadoc of the `Common` class.

Examples of the IEngineObserver methods and status types include:

`engineStatusChanged(int status)` with status values such as Common.EngineStatus.DOWNLOADING, Common.EngineStatus.IDLE, Common.EngineStatus.PAUSED, Common.EngineStatus.BLOCKED, and more.

`settingChanged(int flags)` with flag values such as Common.SettingFlag.SETTING_MAX_STORAGE, Common.SettingFlag.SETTING_HEADROOM, Common.SettingFlag.SETTING_BATTERY_THRESHOLD, and many more.

`settingsError(int flags)` with flag values such as Common.EngineBlockedReason.DISK_FULL, Common.EngineBlockedReason.POWER, Common.EngineBlockedReason.NETWORK, Common.EngineBlockedReason.PERMISSIONS, and more.

`backplaneSettingChanged(int flags)` with flag values such as Common.BackplaneSettingFlag.SETTING_DEVICE_DOWNLOAD_ENABLED, Common.BackplaneSettingFlag.SETTING_MAX_DOWNLOADS_PER_ACCOUNT, Common.BackplaneSettingFlag.SETTING_DEVICE_DOWNLOAD_ENABLED, and more.

```
assetDeleted(String assetUUID, String assetID)

assetExpired(IIdentifier asset)

assetLicenseRetrieved(IIdentifier asset, boolean aSuccess)

engineDidNotStart(String exceptionStringReason)
```

IQueueObserver: Android

The Virtuoso engine notifies any registered `IQueueObserver` of changes for the queue or assets within it, for example, when downloads start and end, when a download error occurs, and any queue changes.

An `IQueueObserver` can be registered with the Virtuoso object through the `addObserver(...)` method and removed with `removeObserver(...)`.

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.

Review the Virtuoso SDK javadoc for the Observers class to find complete documentation covering the `IQueueObserver` interface. Most of the methods receive a reference to the relevant Asset, in the form of the `IIdentifier` interface implemented by `Asset`.

Examples of the methods of `IQueueObserver` include:

`engineStartedDownloadingAsset(IIdentifier asset)` Useful for updating the UI to reflect this asset is being downloaded.

PENTHERA ANDROID SDK CAN NOTIFY DOWNLOAD START MULTIPLE TIMES FOR SAME DOWNLOAD

Anytime the download of an asset in queue stops and restarts, the SDK will send another `DOWNLOAD_START` broadcast and call `engineStartedDownloadingAsset(IIdentifier asset)` method on any registered `IQueueObserver`. To distinguish whether a given broadcast or observer notification is a ***NEW*** download versus the resume of a previous download, your code should look at the value of `getCurrentSize()` on the asset in the notification. If that returns zero, it is a new download. If it returns a non-zero value, the notification is for a resume of a previous download. Recall that for broadcasts the asset will be contained in the extras on the notification.

Note also that due to required differences in download engines, the Penthera iOS SDK behaves differently. On iOS the download start notification will only occur the first time download begins for an item in queue, and not when downloads resume.

`enginePerformedProgressUpdateDuringDownload(IIdentifier asset)` With this method your code can be notified of download progress, which can be retrieved from the provided Asset and displayed in the user interface.

`engineCompletedDownloadingAsset(IIdentifier asset)`

`engineCompletedDownloadingSegment(IIdentifier asset)`. Typically not useful for UI updates, but could be useful for logging or debugging.

`engineUpdatedQueue()` occurs with any change to the queue, such as an item added, removed, or reordered.

`engineEncounteredErrorParsingAsset(String assetId)`

`engineEncounteredErrorDownloadingAsset(IIdentifier asset)` When this is called, the Asset status will reflect the most recent error, which can be retrieved from the getDownloadStatus() method on the provided Asset. Here are some of the error conditions Virtuoso may attach to the Asset:

Error Condition	Description	Results
Invalid mime type	MIME type advertised by the HTTP server does not match the expected MIME type you supplied. **This is usually a very important category of error for customers to track**. Often a MIME type of `text/html` results when the server returns an error page. Otherwise, this result may indicate your server is returning the correct file, but with a different MIME than you expected. An example is when developers expecting closed caption to be a form of text file are surprised to find some of their manifests use mp4 files for closed caption data, which then arrive unexpectedly as video MIME types.	SDK updates Asset's status to `AssetStatus.DOWNLOAD_FILE_MIME_MISMATCH` and increments its error count.

Error Condition	Description	Results
Observed file size disagrees with expected file size	After the download completes, the size of the downloaded file on disk does not 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

IBackplaneObserver: Android

The Virtuoso engine notifies any registered `IBackplaneObserver` when the SDK has completed interactions with the Penthera Cloud (the Backplane).

An `IBackplaneObserver` can be registered with the Virtuoso object through the `addObserver(...)` method and removed with `removeObserver(...)`.

These communications with the backplane are initiated within our SDK, and their result will be handled by SDK code, so there is no direct requirement for you to observe backplane interactions in this way. You may still wish to observe at this low level for debug logging, to feed a custom analytic engine, or as an option to make certain UI updates.

Receiving this callback might be useful when download enablement has been changed, a device was registered or unregistered, a device name was changed, or if during development you want to log when a backplane sync completed.

Review the Observers javadoc for documentation covering the `IBackplaneObserver` interface. The callback method parameters contain an `int Common.BackplaneCallbackType`, an `int Common.BackplaneResult`, and a String error message if appropriate. See the javadoc of the Virtuoso SDK `Common` class for descriptions of the list of possible `BackplaneCallbackTypes` (e.g., REGISTER and NAME_CHANGE), and for the possible `BackplaneResult` values (e.g., SUCCESS, FAILURE, and DEVICE_NOT_REGISTERED).

Using Broadcast Receivers: Android

Besides sending events to your app code through observers and callbacks, the SDK also sends system broadcasts. You may optionally capture these system broadcasts with a broadcast receiver for use in your app code, but there is no functional requirement for you to do so. Observers and content providers are preferred over broadcast messages for the purpose of updating your UI. The most common reason customer apps register to receive these broadcast messages is if they are logging the SDK behaviors in a third-party logging or analytics system.

NOTE

This section covers broadcast receivers from the perspective of how your code might receive our SDK broadcasts, if desired. If you are interested in learning more about how and why the Penthera SDK uses various broadcast receivers for internal purposes, and how you would configure your application for the Penthera SDK to function properly, see Broadcast Receivers: Android [23]

Receiving Broadcasts

As is customary with Android broadcast receivers, in order to receive the desired broadcast messages you will implement a broadcast receiver and register an intent filter. The receiver must be declared either in the application manifest, or in context within code.

To receive any of the broadcasts sent by the Penthera SDK, the action names in the declared receiver's intent filter must be in the following format:

```
CLIENT_PACKAGE_IDENTIFIER + “.” + BROADCAST_NAME
```

where `CLIENT_PACKAGE_IDENTIFIER` is the same value specified for the `com.penthera.virtuososdk.client.pckg` metadata as declared in the `AndroidManifest`, and `BROADCAST_NAME` is one of the broadcasts declared by the SDK in `Common.java`. Examples of the available broadcasts from the Penthera SDK can be found in the javadocs, and in the discussions Broadcasts of Download State: Android [37] and Broadcasts of Analytic Events [39].

Below is an example of a broadcast receiver declared in the Android manifest, registered to receive various download and playback events. In this example, the value of `com.penthera.virtuososdk.client.pckg` would have been declared elsewhere in the manifest to be "com.my.app.auth":

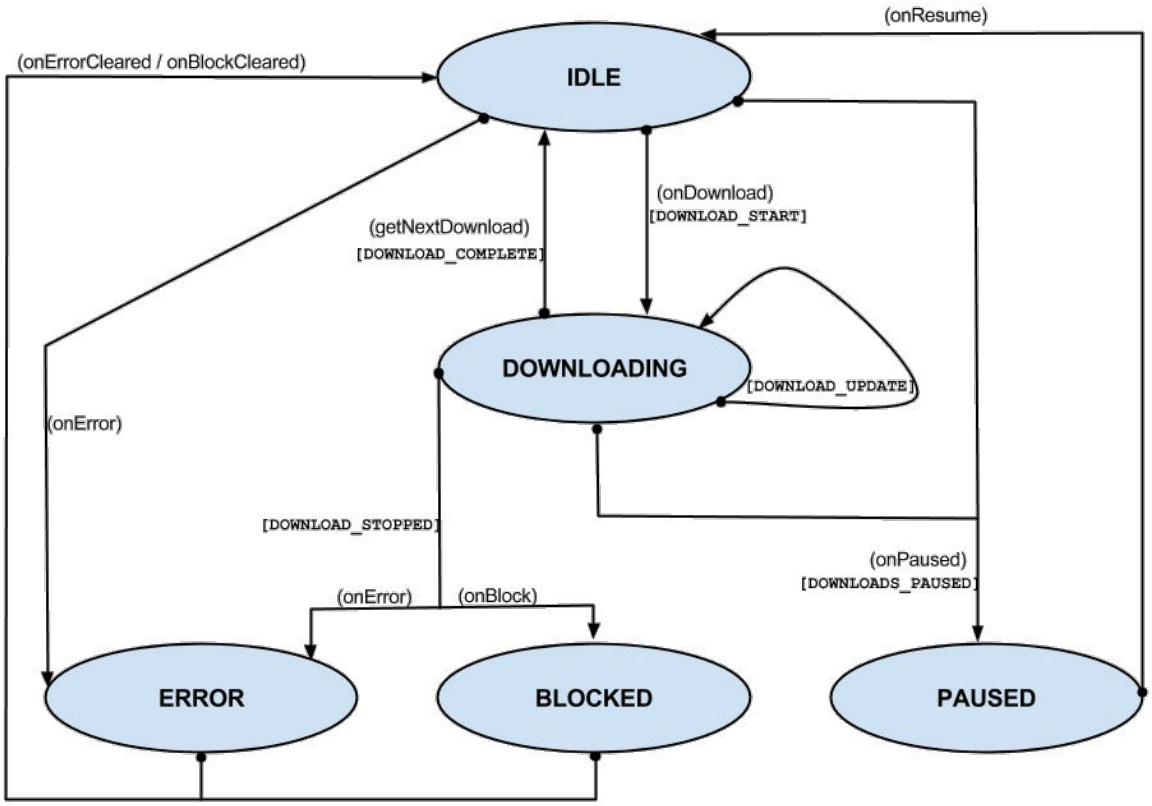
```
  <receiver android:name="com.my.app.NotificationReceiver"
            android:enabled="true"
            android:label="NotificationReceiver"
            android:process=:"notification_service">
     <intent-filter>
       <action android:name="com.my.app.auth.NOTIFICATION_DOWNLOAD_START"/>
       <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>
```

TIP

See the demo application for an expanded example of using notifications and events.

Broadcasts of Download State: Android

The diagram and table below show the different download states and the broadcasts that are generated when moving between states. The javadoc for the `Common` class in the SDK documents the entire list of possible download status broadcasts.



Download Broadcasts

Broadcast	Description
NOTIFICATION_DOWNLOAD_START	A downloaded has started, or resumed after being paused. The extras in the broadcast's 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 extras in the broadcast's Intent 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 broadcast's Intent will detail which asset.
NOTIFICATION_DOWNLOAD_UPDATE	A progress update for a download. The extras in the broadcast's Intent will contain the asset and the number of assets in the queue.
NOTIFICATION_DOWNLOADS_PAUSED	A download was paused. The extras in the broadcast's Intent will contain the asset and the number of assets in the queue.
NOTIFICATION_MANIFEST_PARSE_FAILED	When a new asset is queued for download by your app code, it is common practice to provide a manifest parsing observer to that method call (passed as an argument). If your app is backgrounded and cleaned up while the manifest is parsing, the observer cannot be notified. If this is important to your app implementation, this broadcast can be received as an alternative notice that an asset has not been queued. The extras on the broadcast Intent will contain the asset id for the asset which could not be queued.

PENTHERA ANDROID SDK CAN NOTIFY DOWNLOAD START MULTIPLE TIMES FOR SAME DOWNLOAD

Anytime the download of an asset in queue stops and restarts, the SDK will send another `DOWNLOAD_START` broadcast and call `engineStartedDownloadingAsset(IIdentifier asset)` method on any registered `IQueueObserver`. To distinguish whether a given broadcast or observer notification is a ***NEW*** download versus the resume of a previous download, your code should look at the value of `getCurrentSize()` on the asset in the notification. If that returns zero, it is a new download. If it returns a non-zero value, the notification is for a resume of a previous download. Recall that for broadcasts the asset will be contained in the extras on the notification.

Note also that due to required differences in download engines, the Penthera iOS SDK behaves differently. On iOS the download start notification will only occur the first time download begins for an item in queue, and not when downloads resume.

Broadcasts of Analytic Events

The Penthera Cloud logs a variety of SDK-related events which you can receive in server-side reports, and perform analytics on them. You can even send custom events to our SDK which it will log for you in our Penthera Cloud, where they will be available along with our standard events.

Alternatively, you may configure your mobile app to receive broadcast events for some or all of our SDK's predefined broadcasts (see Broadcast Receivers: Android [23] for contextual information). Your broadcast receiver may be registered in the manifest or in context. The demo application provides an example of registering to receive a few events in the Android manifest (although in the demo code they are unused in the broadcast receiver which receives them).

TIP

The timing of one event, the "app launch" event, can be influenced by the client application. This event simply represents when the application UI was launched. If the application has not manually generated an app launch event by 5 seconds after the SDK is started, the SDK will automatically generate the event itself.

The following code can be used if you wish to manually induce the creation of the "app launch" event, so that it occurs at a particular time and place in your app code:

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

These broadcasts listed below are sent to support custom or third-party analytics, and they directly map to log events recorded in the Penthera Cloud. See the javadoc for the SDK's `Common` class for complete information. In each broadcast, the `IEvent` being reported is always available in the extras of the broadcast's Intent, at the `EXTRA_NOTIFICATION_EVENT` key. The event object may contain 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.

Broadcast	Description
EVENT_DOWNLOAD_START	A file began to download, or resumed a download which was previously stopped .
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~~	This event appears in `Common.java`, but is not used by the SDK.
EVENT_PLAY_STOP	A player has stopped local playback of the asset.
~~EVENT_STREAM_PLAY_STOP~~	This event appears in `Common.java`, but is not used by the SDK.
EVENT_PLAYBACK_INITIATED	The player initiated the *first* playback of the asset from the SDK proxy.
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..

PENTHERA ANDROID SDK CAN NOTIFY DOWNLOAD START MULTIPLE TIMES FOR SAME DOWNLOAD

Anytime the download of an asset in queue stops and restarts, the SDK will send another `DOWNLOAD_START` broadcast and call `engineStartedDownloadingAsset(IIdentifier asset)` method on any registered `IQueueObserver`. To distinguish whether a given broadcast or observer notification is a ***NEW*** download versus the resume of a previous download, your code should look at the value of `getCurrentSize()` on the asset in the notification. If that returns zero, it is a new download. If it returns a non-zero value, the notification is for a resume of a previous download. Recall that for broadcasts the asset will be contained in the extras on the notification.

Note also that due to required differences in download engines, the Penthera iOS SDK behaves differently. On iOS the download start notification will only occur the first time download begins for an item in queue, and not when downloads resume.

Configuring Frequency of Download Progress Updates

You can configure the frequency at which your app receives notifications regarding downloads through the `ISettings` interface. You may specify the update frequency on a time basis or on a percent complete basis. For Segmented Assets you may also specify update frequency based on how many segments have been downloaded.

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

Progress Updates By Percent (Integer): Minimum percentage 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): Minimum number of milliseconds that should pass before the SDK sends out a broadcast regarding progress updates. Set to `Long.MAX_VALUE` to disable updates based on timed intervals. (default: 5000)

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

If multiple values are provided in the configuration, the SDK will not send out an update until the combined settings are all met (i.e., at the latest opportunity).

The SDK guarantees updates are not sent out early, that is, no updates will occur within the configured intervals but updates may sometimes arrive later than configured. Whether your code receives the download progress updates via observer or broadcast receiver, the updates still rely internally on broadcasts. Because broadcasts depend on Android delivery, you may observe some variation between the times your code receives updates and the values you have configured.

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

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

HTTP Persistent Cookie Management

The SDK installs a cookie manager within the Service process to persist cookies between executions. This ensures that any cookies delivered during manifest responses will be correctly added to requests during segment download. In most cases this will not interfere with any use of cookies in the main application process, as Android does not share the cookie manager between processes. In particular, DRM licenses and advertising reporting are executed via the SDK within the main application process, not the SDK service process, so the SDK cookie manager will not manage relevant cookies on these types of HTTP requests.

IMPORTANT

If your application requires a custom cookie manager, and inserts this within the application class, then your custom cookie manager may clash with the version used by the SDK in its Service process. In that case the SDK cookie manager may be disabled by entering the following meta data into the Android manifest:

```
<!-- Disable Penthera Cookie Manager -->
<meta-data
android:name="com.penthera.virtuososdk.persistentcookiemanager.disabled"
            android:value="true" />
```

R8/Proguard

It is always recommended to use R8/ProGuard 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 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 shrink ruleset to your 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 regis-

tered 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 is up to you to add appropriate Proguard configuration for things you don't want to warn about. 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 anticipate 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.

CAUTION

The Penthera Android SDK is built using the R8 toolchain. This can cause warnings and minor issues if the client app is still being built with ProGuard. See our Penthera 312: Known Issues doc for details and resolutions.

Disabling R8 has been deprecated by the Android toolchain. We recommend updating from ProGuard at the earliest opportunity.

Configure Debug Logging: Android

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`.

With `gradle` you can build against our debug SDK build by appending `-debug` to the dependency name for our SDK. The default log level for a debug build is DEBUG. This is also the minimum log level accessible in our SDK 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 lower level outputs will not be generated by the release SDK.

The log level can be changed programmatically within the SDK if the client application wishes to reduce logging during development, or to 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);
```

Implementing Basic Features

Once you have learned the fundamentals of the Penthera SDK, and installed & configured the SDK in an app, this section will step you through the code necessary to implement the most common features in your app.

Enable/Disable Device & Engine for Downloads: Android

The Penthera Cloud tracks which devices are enabled for download, and enforces the global `max download-enabled devices per user` parameter, which you may set via the Penthera Cloud web admin UI. The Penthera Cloud also offers an option to set the default state of new client devices to enabled or disabled.

The individual SDK instances “know” their own download-enabled status, because the Penthera Cloud communicates it to them. An SDK whose download-enabled status is `false` can add assets to queue, but will not download the queued assets.

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

```
// create and register a backplane observer to 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);

// Now we request the actual enablement change.  This relies on having an
active
// network connection and the SDK being authenticated with the Backplane.

mVirtuoso.getBackplane().changeDownloadEnablement(true); // true=enable,
false=disable
```

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

Queue an Asset for Download: Android

Queueing an asset for download varies depending on the type of asset. In each case the `IAssetManager` instance is used, but the methods vary. The `IAssetManager` instance is retrieved from your Virtuoso instance.

Queue a single file (for example, an mp4)

The `IAssetManager` contains a method to create a single-file asset, and then the file is simply added to the queue 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",                        // Expected 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);
```

Queue a segmented video, specifying a target bitrate

For segmented, manifest-based videos, there are various parameters to define (e.g., download URL and bitrate settings) so the Penthera SDK can download the desired files of the asset. The SDK will download and parse the manifest, identify the highest-quality profile whose bitrate does not exceed the specified max bitrate, and then download all the fragments belonging to that profile.

The details for the desired asset can be provided using an asset parameter builder which is specific to the manifest type (e.g. an HLS or MPEG-DASH). One of the builder parameters indicates whether the SDK should automatically add the asset to the download queue when it completes parsing the manifest, or allow your code to decide later if and when the asset should be added to the download queue. The builder produces a configured parameters instance which can then be passed to one of our synchronous or asynchronous methods on `IAssetManager` to create the Asset instance.

The synchronous create methods should never be called from the UI thread. With the asynchronous create methods you may implement an `ISegmentedAssetFromParserObserver` to receive the outcome of the parsing. Within that observer you may add the parsed asset to the download queue if you did not configure that to occur automatically.

The synchronous and asynchronous methods of the `IAssetManager` interface to create/queue a multi-segment video are:

- `createHLSSegmentedAsset` or `createHLSSegmentedAssetAsync`
- `createMPDSegmentedAsset` or `createMPDSegmentedAssetAsync`

See the javadocs to explore various other options, especially javadocs for `IAssetManager`, `HLSAssetBuilder`, and `MPDAssetBuilder`.

The following example code demonstrates creating an `ISegmentedAssetFromParserObserver`, using a builder to create the desired parameters instance, and passing that parameters instance to the create method of `IAssetManager`. Note that "addToQueue" is configured to occur automatically, so it is not necessary to manually add the parsed asset to the download queue in this observer implementation.

```
// 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")                       // your
unique identifier
       .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.

Pause/Resume Download: Android

The Penthera Android SDK provides the ability to pause and resume individual assets, using `pauseDownload(...)` and `resumeDownload(...)` methods on the `IAssetManager` object.

An example of how this could be used in a catalog item detail Activity to handle pausing download of the current detail view's asset follows:

```
IAsset currentAsset;
IAssetManager mAssetManager;

// handlers for UI pause and resume buttons
private void handlePause() {
    if (currentAsset != null && currentAsset.getDownloadStatus() !=
AssetStatus.DOWNLOAD_PAUSED)
    {
        mAssetManager.pauseDownload(currentAsset);
    }
}

private void handleResume() {
    if (currentAsset != null && currentAsset.getDownloadStatus() ==
AssetStatus.DOWNLOAD_PAUSED)
    {
        mAssetManager.resumeDownload(currentAsset);
    }
}
```

Similar methods are available which pause/resume based on the asset ID instead of the asset instance. To provide pause buttons for each item in your download queue view, pass the asset or ID corresponding to the row the user wishes to pause or resume.

The SDK also provides methods on `IService` to pause/resume **all** downloads. This is useful for providing your user with a UI button to pause/resume all downloads, but it is important to note that unlike pause/resume of individual assets (which is synchronous and immediately updates the status of the asset), your code will require an observer to confirm completion of the asynchronous `IService` pause/resume status change. An example of handling a pause/resume toggle in your UI for all downloads follows:

```
IService mConnectedService;

private void handlePauseResume(MenuItem item) {

if(getString(R.string.pause).equalsIgnoreCase(item.getTitle().toString())){
        try {
            if( mConnectedService != null )
                mConnectedService.pauseDownloads();
        } catch (ServiceException e) {
            e.printStackTrace();
        }
    }
    else
if(getString(R.string.resume).equalsIgnoreCase(item.getTitle().toString())){
        try {
            if( mConnectedService != null )
                mConnectedService.resumeDownloads();
        } catch (ServiceException e) {
            e.printStackTrace();
        }
    }
}
```

We discussed the "master switch" in Enable/Disable Device & Engine for Downloads: Android [44], which is not the recommended method for providing your user with a pause/resume all asset button.

The master switch might be useful if you want to programmatically disable downloads such as during a delinquent account payment status for the user.

Cancel a Download: Android

Canceling a download is the same as deleting the asset from the device, even if the asset has not yet completed download.

The SDK provides a simple method to delete an asset. This works the same whether the item is waiting to download, still being downloaded, or has completed download. Call one of the `delete(...)` methods on `IAssetManager`, providing either the asset instance or its asset ID. The `IAssetManager` also provides a `deleteAll()` method.

```
IAssetManager mAssetManager;

private void handleDelete(IAsset anAsset)
{
    mAssetManager.delete(anAsset);
}
```

List Assets: Android

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

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

To retrieve a cursor on the assets in the download queue:

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

Delete an Asset: Android

The SDK provides a simple method to delete an asset. This works the same whether the item is waiting to download, still being downloaded, or has completed download. Call one of the `delete(...)` methods on `IAssetManager`, providing either the asset instance or its asset ID. The `IAssetManager` also provides a `deleteAll()` method.

```
IAssetManager mAssetManager;

private void handleDelete(IAsset anAsset)
{
    mAssetManager.delete(anAsset);
}
```

Another example which also demonstrates retrieving the asset reference (as its base class `IIdentifier`) by asset UUID, then passing the asset ID to the delete method:

```
IIdentifier anAsset = mAssetManager.get(uuidString);
int anAssetId = anAsset.getId();

mAssetManager.delete(anAssetId);
```

Delete All Downloads: Android

There are two different concepts which people may think of as "deleting all downloads." If you want to remove (or "flush") all assets from the download queue *which are not yet downloaded*, call the `flush()` method of the `IQueue` interface:

```
mVirtuoso.getAssetManager().getQueue().flush();
```

Alternatively, the `IAssetManager` has a `deleteAll()` method which deletes all queued *and previously-downloaded* assets.

Play Downloaded Content: Android

The SDK does not include a video player. It does, however, provide the mechanism for your media player of choice to access the assets managed by the SDK. For both manifest-based and file-based assets the SDK provides a playout proxy, a local HTTP proxy server, which sits between your media player and the downloaded assets. The SDK provides a URL for each downloaded asset, which your code passes to the player.

You can retrieve the correct reference to any asset by calling the `getPlaylist()` method on the `IAsset` instance. The URL returned by `getPlaylist()` will point the player to the local proxy, at a port owned by the SDK's `VirtuosoClientHTTPService`. Playback of manifest assets results in your player retrieving both the manifest and its referenced asset files from the SDK's local proxy server.

CAUTION

Downloaded manifest-based assets will not play correctly without accessing them through the SDK proxy. While it may be technically feasible for your app code to access downloaded *file-based* assets through their local file storage paths, *we strongly advise against this approach*. This would invalidate certain metrics we collect on your behalf, and would introduce other problems and inconsistencies. Access downloaded assets for playback using the URL provided by `getPlaylist()` on `IAsset`, even for file-based assets.

Example of playing back assets, where `openIntent` contains the player and also holds the URI and MIME type for the asset to be played:

```
public static void play(Context context, IAsset asset)
{
   Intent openIntent = new Intent(android.content.Intent.ACTION_VIEW);
   try {
      URL assetURL = asset.getPlaylist();
      String mimeType = "video/*";
      openIntent.setDataAndType(Uri.parse(assetURL.toString()), mimeType);
   } catch (MalformedURLException e) {
      throw new RuntimeException("Not a playable file");
   }
   // Register a 'play start' event
   Common.Events.addPlayStartEvent(context,asset.getAssetId());
   // Play the Asset
   context.startActivity(openIntent);
}
```

Unregister Device: Android

The Penthera SDK provides the ability to unregister the local or remote devices from the user's account. Unregistering a device will disassociate it from the user's Penthera Cloud account, delete all of the downloaded assets from the device, and shutdown the Penthera SDK on that device. You will also hear this function described as a "wipe" or "remote wipe." This occurs immediately on the local device, or upon the next Penthera Cloud sync from the remote device.

TIP

Restoring the use of SDK functions on an unregistered device requires starting up the SDK again on that device. Previously downloaded assets will have already been removed.

You may choose for your app UI to present a list of the user's devices, and allow the user to manage the device list. See the javadoc for the `IBackplane` interface for the full set of methods, including retrieving the list of user devices, changing a device nickname, enabling/disabling the SDK on a device, unregistering a device from the user account, and more.

If your code wishes to unregister the current device, simply call `unregister()`:

```
mVirtuoso.getBackplane().unregister();
```

If you wish to unregister one of the other devices on the user's account, once your user has selected the desired `IBackplaneDevice` instance from the list of devices, call `unregisterDevice(...)` with the other device reference:

```
mVirtuoso.getBackplane().unregisterDevice(otherDevice);
```

Note that an unregister may not succeed, or may be delayed, based on factors such as connectivity of the current and/or remote device. If you wish to observe the result of a device unregister, either local or remote, you can do so with an `IBackplaneObserver`. See the javadoc for `IBackplane` and the documentation IBackplaneObserver: Android [36] for details.

DEVICES: DISABLED NOT DELETED

If devices you have remote wiped disappear from the device list in API calls, then all is well. Our backplane does not ever completely delete devices. It just disables them and disassociates them from whatever user ID they were last registered with when you call unregister/remote wipe. We filter out such devices from the basic web and device API call but those devices will still appear in the Penthera Cloud web admin GUI at present. So if you remote wipe a device and it vanishes from the API call device list but not the Penthera Cloud GUI, then all is working as expected.

Troubleshooting

The various SDK functions will log to your application log. If you are unable to determine the cause of issues on your own, contact Penthera support for assistance via email to support@penthera.com.

To help resolve issues, we will usually need a description of the scenario, any code snippets you may be able to provide, and ideally a complete *debug-level* log file from the application session in which you experience the issue.

For issues related to communications off the device, such as errors during download, or failures of methods which rely on communications with the Penthera Cloud, our support will also ask for a log of the network communications captured with the Charles Proxy or other web debugging application. The Charles app can be found at https://www.charlesproxy.com. A Charles log of the full session starting at app launch is usually more helpful than a log of a partial session. Charles capture configuration can be confusing, but Penthera Support can help achieve the ideal setup.

Copyright © 2020 Penthera Partners

What Next?

The following publications may be of interest:

Penthera 101: Introduction to Penthera Development

Penthera 201: Developing with the iOS SDK

Penthera 202: Developing with the Android SDK [3]

Penthera 203: Best Practices

Penthera 301: iOS SDK Beyond the Basics

Penthera 302: Android SDK Beyond the Basics

Penthera 310: DRM Deep Dive

Penthera 311: Server-to-Server with the Backplane API

Penthera 312: Known Issues

Copyright © 2020 Penthera Partners

CONFIDENTIAL - For Licensed Use Only