Release 4.00 of the of Penthera SDK includes improvements that will reduce the code needed to implement and maintain current and future versions. Code that previously was required is now automatically handled, engine notifications are now available by delegate callbacks, long running methods are clearly indicated and include callbacks to improve asynchronous background processing. This document discusses changes to the iOS client App that may simplify integration with the Penthera SDK.

AppDelegate Changes

Previous releases of Penthera SDK required certain methods of AppDelegate be implemented to handle remote push notifications, download, and communication with the Penthera Backend.

Starting in release 4.00, all of the previously required code for this in AppDelegate is deprecated and can be safely removed. This includes any code that references EventHandler which is now obsolete because Penthera automatically handles wiring-up of code that was previously required in AppDelegate.

If your code still needs to handle remote push notifications, please make sure you implement method didReceiveRemoteNotification:fetchCompletionHandler as the preferred method.

The only suggested code for AppDelegate going forward would be limited to enabling logging as show in the following code fragment:

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {

    VirtuosoLogger.enable = true
    #if DEBUG
    VirtuosoLogger.setLogLevel(.vl_LogVerbose)
    #else
    VirtuosoLogger.setLogLevel(.vl_LogError)
    #endif
    return true
}
```

Notification Managers

The Penthera SDK largely executes asynchronously in the background, downloading and preparing assets for offline playback. Status notifications are provided using NSNotificationCenter notifications. These notifications pass parameters in a dictionary, and must be specifically wired-up by adding listeners to NSNotificationCenter.

We have simplified this abstraction by offering a delegate notification pattern, implement the delegate, register it, and the Engine will invoke your delegate methods with type safe parameters. This eliminates the need to manually add listeners to NSNotificationCenter for the various engine notifications.

The result is clear, concise code. A Notification Manager delegate can be implemented by any view or model object with a minimum number of required delegate methods, and optional methods to suite your needs.

Example:
The following example shows a ViewController that implements the Download Engine delegate for various notifications.

```
class ViewController: UIViewController, VirtuosoDownloadEngineNotificationsDelegate {
    var downloadEngineNotifications: VirtuosoDownloadEngineNotificationManager!

    override func viewDidLoad(){
        super.viewDidLoad()
        downloadEngineNotifications
               = VirtuosoDownloadEngineNotificationManager.init(delegate: self)
    }

    func downloadEngineDidStartDownloadingAsset(_ asset: VirtuosoAsset) {
    }
    func downloadEngineProgressUpdated(for asset: VirtuosoAsset) {
    }
    func downloadEngineProgressUpdatedProcessing(for asset: VirtuosoAsset) {
    }
    func downloadEngineDidFinishDownloadingAsset(_ asset: VirtuosoAsset) {
    }
    func downloadEngineDidEncounterError(for asset: VirtuosoAsset,
                                error: Error?,
                                task: URLSessionTask?,
                                data: Data?,
                                statusCode: NSNumber?) {
    }
    func downloadEngineInternalQueueUpdate() {
    }
    func downloadEngineStartupComplete(_ succeeded: Bool) {
    }
}
```

Many of the delegate methods are optional, check header documentation for more information on when those methods are triggered.

API Changes

Engine Startup -

Engine startup has been changed to execute asynchronously, invoking a closure when startup completes. The change to invoke startup asynchronously was necessary to prevent blocking on the UI MainThread. Startup will make network calls, update the local database, and if the user changes, will delete previously downloaded assets. Invoking this method from the UI thread risks iOS terminating the App when the UI thread is blocked.

To simplify the call, we consolidated all required parameters to be passed using the configuration object pattern. Class VirtuosoEngineConfig defines the configuration properties that can be passed. Create an instance of this object and pass it to VirtuosoDownloadEngine startup.

Finally, it's important to note that VirtuosoDownloadEngine.startup should NOT be repeatedly called when your views are displayed. Invoking this method is time, network, and CPU intensive. Invoke it once you have all of the requisite configuration data and then not again unless the app is restarting.

Previously:

Engine startup took five parameters and executed synchronously.

```
let status = VirtuosoDownloadEngine.instance().startup(withBackplane:<url>,
                                    user: <user>,
                                    externalDeviceID: <externaldeviceid>,
                                    privateKey: <private-key>,
                                    publicKey: <public-key>)
if (status == .vde_EngineStartupSuccess {
        print("start succeeded")
} else {
        print("start failed")
}
```

New 4.00:

Engine startup now takes a configuration object, executes asynchronously and invokes a callback closure when complete.

```
guard let config = VirtuosoEngineConfig(user: <user>,
                                    backplaneUrl: <url>,
                                    publicKey: <publicKey>,
                                    privateKey: <privateKey> ) else { return }

VirtuosoDownloadEngine.instance().startup(config) { (status) in
        if status == .vde_EngineStartupSuccess {
                print("start succeeded")
        } else {
                print("start failed")
        }
}
```

In addition to the callback, the SDK Engine delegate method downloadEngineStartupComplete is called when engine startup completes.

Asset Creation -

Asset creation has been significantly simplified. Previous releases created assets using various different class methods, each of which required up to 14 parameters, including closures needed to add the asset to the download queue before downloading would start.

This has been greatly simplified.

Asset creation now takes a single configuration object (config) that will set reasonable default values. Set the config values you need, invoke Asset init(config:), and the asset is both created and automatically added to the download queue.

Asset creation now takes VirtuosoAssetConfig, which requires minimal parameters, defaults the remaining values and allows you to change as needed.

Assets created using VirtuosoAssetConfig no longer need to provide the closures that were required to add the asset for downloading. That now happens automatically when VirtuosoAssetConfig.autoAddToQueue is set true, the default state.

Previously:

```
// create asset
let _ = VirtuosoAsset.init(remoteURL: "<url>", assetID: "<asset id>",
                    description: "<asset description>",
                    publishDate: nil,
                    expiryDate: nil,
                    expiryAfterDownload: 0,
                    expiryAfterPlay: 0,
                    assetDownloadLimit: -1,
                    enableFastPlay: false,
                    ancillaries: nil,
                    adsProvider: nil,
                    userInfo: nil,
                    onReadyForDownload: { (downloadAsset) in

    guard let asset = downloadAsset else { return }

    // Add asset to download queue
    VirtuosoDownloadEngine.instance().add(toQueue: asset, at: 0, onComplete: nil)

}) { (parsedAsset, parseError) in

}
```

Now:

```
// Create asset configuration object
guard let config = VirtuosoAssetConfig(url: "<url>",
                                       assetID: "<asset id>",
                                       description: "<asset description>",
                                       type: <kVDE_AssetType>) else {
    return // create failed
}

// Create asset and commence downloading.
```

```
let _ = VirtuosoAsset.init(config: config)
```

The Penthera SDK is designed to run as a background service. API's to create and start downloading Assets invoke code paths that make network calls and writes to CoreData to store Asset configuration details.

Best practices for using the Penthera SDK includes invoking many of the methods using background dispatch queues. This is necessary to ensure UI refreshes are smooth and uninterrupted by long running operations like network calls and disk writes.

The following code fragment shows best practice for creating and starting the downloading of a new Asset in Swift using `DispatchQueue`

Example:

```
DispatchQueue.global(qos: .background).async {
    // Create asset configuration object
    guard let config = VirtuosoAssetConfig(url: "<url>",
                                           assetID: "<asset id>",
                                           description: "<asset description>",
                                           type: kVDE_AssetType.vde_AssetTypeHLS) else
    {
        return // create failed
    }

    // Create asset and commence downloading.
    let _ = VirtuosoAsset.init(config: config)
}
```

Performing asset creation using a background thread will ensure that any blocking operations do not effect UI refresh and will result in smooth rendering experience in the Client App.

Background Delegate callbacks -

Engine delegate callbacks can be configured to be invoked on an OperationQueue, by default will be invoked on the UI thread queue. For cases where the Customer would prefer to process delegate calls using a background thread, Penthera provides an initializer that takes OperationQueue for that purpose.

Example:

```
let operationQueue = OperationQueue()
operationQueue.name = "<name of this queue>"
let engineNotifications =
        VirtuosoDownloadEngineNotificationManager.init(delegate: self,
                                                       queue: operationQueue)
```

Delegate methods will now be invoked on the OperationQueue thread.

Playback -

Asset playback checks require blocking calls that will cause UI refresh issues.

In previous releases of the SDK, Customers used instance property (isPlayable) on ViruosoAsset. This call involves blocking calls to the data-base layer that will block UI refreshes while the property value is being resolved. Internally, we have made changes to alleviate the impact of these blocking calls.

This has been further improved with a new class method on VirtuosoAsset. Class method isPlayable will perform the check without blocking the UI thread, and when done will invoke the callback on the UI thread with a Boolean flag indicating the asset is playable (true).

Example:

```
VirtuosoAsset.isPlayable(asset) { (playable) in
    // now executing on main thread
    if playable {
    }
    else {
    }
}
```

Errors -

Error domains for NSErrors returned by the SDK have been updated. This change was made to help Customers categorize Error.code values. Penthera now creates Errors using error domains that are grouped and matched directly to the source of the error as defined by an Enum type in the SDK.

The following table shows the Error Domain, and the Error Codes that will be associated with errors in that domain.

Error Domain	Error Codes
Virtuoso.DownloadErrorCode	kVDE_DownloadErrorCode (enum)
Virtuoso.HttpResponseCode	HttpResponse status codes, i.e. 200-500
Virtuoso.BackplaneStatusCode	kVBP_StatusCode (enum)
Virtuoso.DownloadErrorType	kVDE_DownloadErrorType (enum)
Virtuoso.ErrorCode	kVDE_ErrorCode (enum)
Virtuoso.PlayerError	kVDE_PlayerErrorCode (enum)
Virtuoso.AssetResourceLoader	kVAV_ErrorCode (enum)