



Predictive Content Delivery SDK iOS Integration Guide



[1 Introduction](#)

[2 Getting Started](#)

[2.1 Requirements and Dependencies](#)

[2.2 Installing the iOS SDK](#)

[3 Integrating with your iOS Application](#)

[3.1 Including the SDK](#)

[3.2 Initialization](#)

[3.3 Registration](#)

[3.4 SDK Delegate](#)

[4 Accessing Content](#)

[4.1 Data Download](#)

[4.1.1 Prepositioned Content](#)

[4.1.2 User/Client initiated content download](#)

[4.2 Working with Data Sets](#)

[4.3 Content Items](#)

[4.3.1 Accessing Content from Data Sets](#)

[4.3.2 Accessing Content Individually](#)

[4.3.2.1 Using uniqueId](#)

[4.3.2.2 Using content_unique_id](#)

[4.4 Playing HLS Video Content](#)

[4.5 Playing Non-HLS Content](#)

[4.6 Content Item States](#)

[4.7 Start/Cancel Content Downloads](#)

[4.8 Start Downloading a Specific Item](#)

[4.9 Pause/Resume Downloading a Specific Item](#)

[4.10 Track Progress of a Downloading Video](#)

[4.11 Content Sources](#)

[4.12 Content Categories](#)

[4.13 Listening for Data Set Changes](#)

[4.14 Data Set Filters](#)

[4.15 Delete/Lock/Save an Item](#)

[4.16 Record the Video Consumption by the user](#)

[4.17 Modify File Download Request](#)

[4.18 Accessing Subtitles and Thumbnails](#)

[4.19 Storing additional information on each item](#)

[5 SDK Settings](#)

[5.1 Network Preference](#)

[5.2 Item Download Behavior](#)



[5.3 Auto Purge](#)

[Maximum Number of Concurrent Downloads](#)

[6 Download Policy](#)

[7 Content Migration](#)

[8 Appendix - Requirements and Dependencies](#)

[8.1 Background Execution](#)

[8.2 Remote Notifications](#)

[8.3 App Transport Security](#)

[8.4 Call force fill on registered SDK](#)

[8.5 Avoid server response delay for initiating download](#)



1 Introduction

The Predictive Content Delivery (PCD) SDK prepositions content based on user preferences and policies set up between the client and server. Prepositioning operates on a push basis, so there is no need for the developer to initiate downloads. Downloads also continue whether the app is active in the foreground, running in the background, or entirely closed. The SDK acts as a data cache, providing access to content along with its current download state. Configuring your project to support these features is covered below.

2 Getting Started

The basic steps for SDK setup are initialization followed by registration. Content begins prepositioning onto the device after registration. Your app is notified via delegate callbacks as files change status. Cached contents are accessible individually or as data sets.

2.1 Requirements and Dependencies

The PCD SDK requires iOS 8 or higher.

A PCD SDK license key is required for registration.

The application's product name (**Project** → choose target → **Build Settings** → **Packaging** → **Product Name**) must match the name provided on the PCD Web portal SDK license page. The portal field for this is "iOS Application ID."

In order to preposition content, the app must enable Background Execution and Remote Notifications. See the appendix for guidance.

2.2 Installing the iOS SDK

1. Download the PCD SDK and unzip it in a folder under your project, e.g. `~/myproject/pcd_sdk`.
2. Locate the `VocSdk.framework` under `~/myproject/pcd_sdk/iphoneos/` and copy it into your project folder, e.g. `~/myproject/VocSdk.framework`. This makes the framework available before the first build so you can add it to your project. On each build afterward it will be copied here by the `copy_vocsdk` build phase script.
3. Add the framework to your Xcode project.
 - a. Open your project in Xcode.
 - b. Open the *File* menu.
 - c. Click *Add Files to <project>*.
 - d. Choose `~/myproject/VocSdk.framework`
4. Link the SDK to your project.
 - a. Open project settings by clicking the project name in the *Project navigator*.
 - b. Click the *General* tab.
 - c. Under *Embedded Binaries*, click + and choose `VocSdk.framework`.
 - d. Click *Add*.

5. The right framework for your platform -- simulator or device -- must be copied into place before building your app. This requires a build phase to be added to your project settings.
 - a. In **Project Settings**, choose your build target then click on **Build Phases**.
 - b. Click the “+” to add a new build phase, and choose “New Run Script Phase.”
 - c. Drag this build phase to position it after Target Dependencies.
 - d. Optionally single-click its name and rename it to copy_vocsdk.
 - e. Click the arrow to expand the copy_vocsdk row.
 - f. Leave the default shell setting of /bin/sh.
 - g. Add the following text to the script area. The “copy_vocsdk.sh” path is relative to your .xcodproj file and may need modification depending on where you unzipped it (pcd_sdk in this example).

```
./pcd_sdk/copy_vocsdk.sh "${PROJECT_DIR}/pcd_sdk"
```

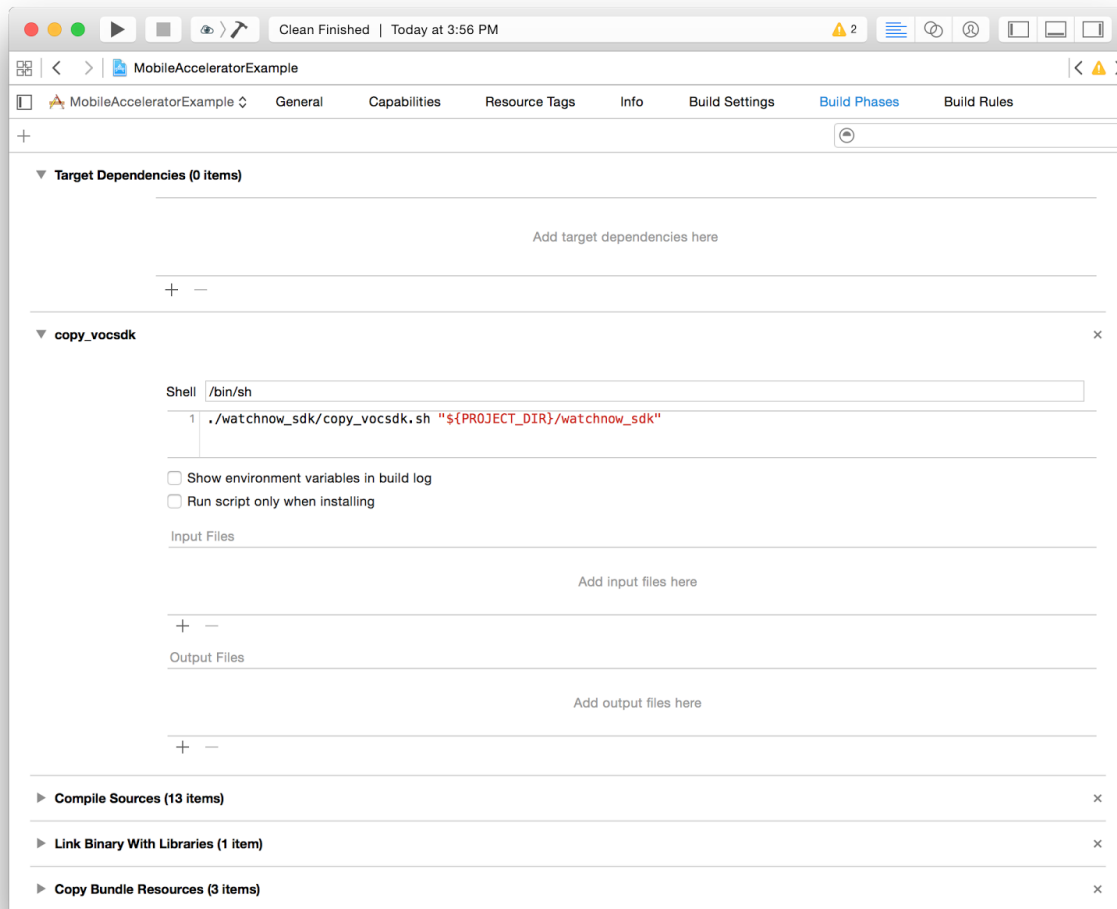
The second parameter, “\${PROJECT_DIR}/pcd_sdk”, is the location of the /iphoneos and /iphonesimulator folders from the distribution archive.

In this example, the folder structure is as follows:

```
/myproject/myproject.xcodeproj
/myproject/pcd_sdk/copy_vocsdk.sh
/myproject/pcd_sdk/iphoneos/
/myproject/pcd_sdk/iphonesimulator/
```

The script will copy the correct VocSdk.framework to the /myproject/ folder at the start of every build.

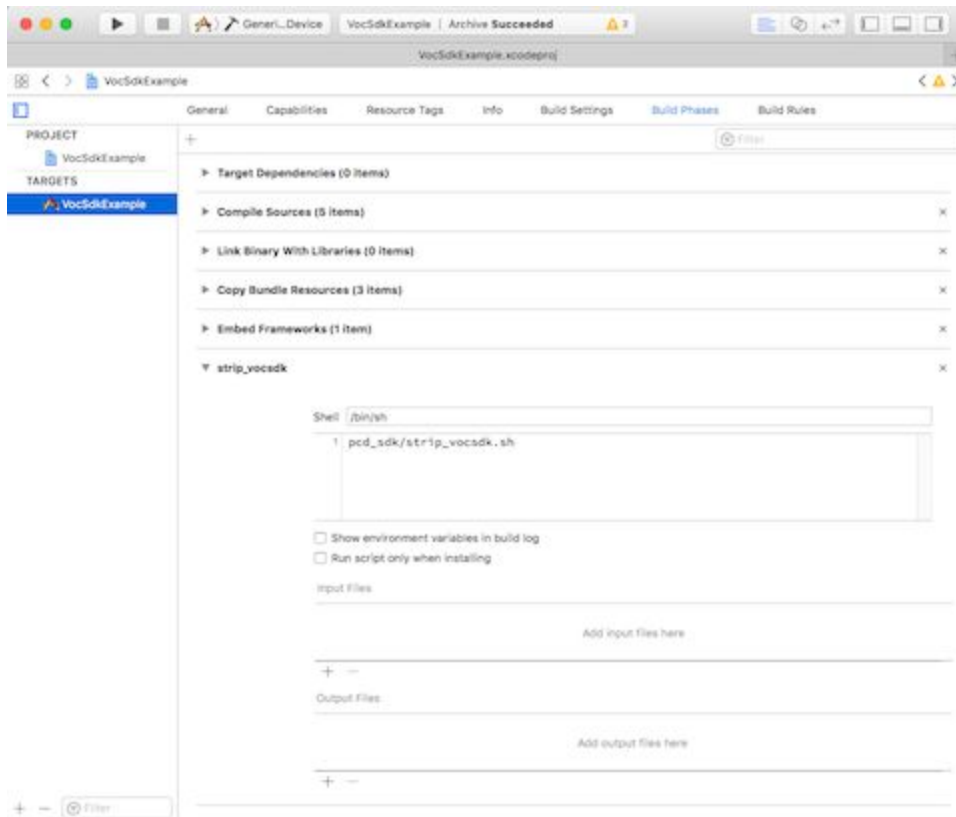
The final build phase should look like this:



6. As an alternative to using the `copy_vocsdk.sh` script to choose the proper framework for your app, you can build your app using the “fat” binary you copied into your project in step 3. Your app will build properly with this “fat” binary. However, if you attempt to submit your app to Apple’s App Store, it will be automatically rejected because the “fat” binary has iPhone Simulator “slices” within it. For this reason, we supply an alternative script that will only include the binaries for the architectures that your build supports. You can run this script during the final step of your build as follows:
 - a. In **Project Settings**, choose your build target then click on **Build Phases**.
 - b. Click the “+” to add a new build phase, and choose “New Run Script Phase.”
 - c. Ensure this build phase occurs after the **Embed Frameworks** phase. By default, your new script will be the last step, and this is fine.
 - d. Optionally single-click its name and rename it to `strip_vocsdk`.
 - e. Click the arrow to expand the `strip_vocsdk` row.
 - f. Leave the default shell setting of `/bin/sh`.

- g. Add the following text to the script area. The “strip_vocsdsk.sh” path is relative to your .xcodeproj file and may need modification depending on where you unzipped it (pcd_sdk in this example).

The strip_vocsdsk.sh will only process when the build configuration is for a “Release” build (e.g., when you archive a build). The script will locate the directory where your executable was built. It will then look at each embedded framework, and modify it by extracting the supported build architectures for your run (e.g., arm7, arm64) and then replacing the framework so that it only contains slices for those particular architectures.



3 Integrating with your iOS Application

3.1 Including the SDK

Import the VocSdk header in any class files where the SDK is required. Also define a single VocService object for your app. Create your VocService object in the app delegate to make the SDK available early in the app lifecycle and to simplify access to the VocService from other classes.

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

@property (strong, nonatomic) id<VocService> vocService;
```

3.2 Initialization

The SDK is initialized by the VocServiceFactory call

createServiceWithDelegate:delegateQueue:options:error:. Initialization and registration should take place early in the app lifecycle to begin prepositioning files as early as possible. The recommended place for this is in AppDelegate's *application:didFinishLaunchingWithOptions:.* The create call inputs a reference to the SDK delegate (see [SDK Delegate](#)) as well as a configuration options dictionary. The SDK delegate is the class you designate to respond to SDK activity.

During initialization, the SDK will attempt to register using the license key, if it is found in one of three places:

1. The Info.plist for the app, using the dictionary “vocsdk” within a “com.akamai” dictionary. If found, the dictionary pointed to by the “vocsdk” entry is parsed for the “license” key, and if found, the value associated with that dictionary entry will be used as the license key for registration purposes:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">
<dict>
```

```

<key>CFBundleDevelopmentRegion</key>
<string>en</string>
<key>CFBundleDisplayName</key>
<string>PCDSdkExample</string>

<!-- ... other Info.plist keys omitted ... -->

<key>com.akamai</key>
<dict>
    <key>vocsdk</key>
    <dict>
        <key>license</key>
        <string>your_license_key_goes_here</string>
    </dict>
</dict>
</dict>
</plist>

```

2. A configuration file for the app, identified by the value in the Info.plist key “com.akamai.vocsdk.config.file.” The value is a path to a config file, relative to the main bundle for your app. This file may be a property list (similar to the Info.plist for your app):

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>license</key>
    <string>your_license_key_goes_here</string>
</dict>
</plist>

```

Alternatively, the file may be a JSON file, as shown in the example below:

```
{  
  "license" : "your_license_key_goes_here"  
}
```

3. The options dictionary passed to the `createServiceWithDelegate:delegateQueue:options:error:` method, using the dictionary key “license.”

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions  
{  
    NSError *error = nil;  
  
    NSDictionary *options = @{ "license" : "your_license_key_goes_here" };  
  
    self.vocService = [VocServiceFactory createServiceWithDelegate:self  
                        delegateQueue:[NSOperationQueue mainQueue]  
                        options:nil  
                        error:&error];  
  
    if (!self.vocService) {  
        // error handling - could not start service  
        return NO;  
    }  
  
    // app initialized  
    return YES;  
}
```

Options passed in the options parameter dictionary take precedence over Info.plist options and file options in the following order of precedence:

1. The options dictionary
2. The Info.plist
3. Configuration property list file or JSON file in bundle

3.3 Registration

Registration is a one-time event that activates the VocService with a particular license key. This key is linked to a content catalog, generation of usage statistics, and other SDK capabilities. It is required

for most SDK features so registration occurs when the license key is passed at initialization time (see [Initialization](#)). The SDK maintains a *state* property that indicates whether registration already succeeded on this device. If `vocService.state` is equal to `VOCServiceStateNotRegistered` then registration may have failed, or may not have been attempted for some reason (e.g., there is no network connectivity). The SDK automatically attempts to re-register on unsuccessful registration due to device failures and server errors.

Once registration is successful, the SDK calls `VocServiceDelegate`'s *didRegister:*. This can be used to report or log any problems starting the SDK.

3.4 SDK Delegate

The SDK notifies your app of various events throughout its life cycle. Messages are sent asynchronously to an SDK delegate in your code that implements the *VocServiceDelegate* protocol. All of the delegate methods are optional.

```
- (void) vocService:(nonnull VocService *)vocService didBecomeNotRegistered:(nonnull
NSDictionary *)info;
- (void) vocService:(nonnull VocService *)vocService didFailToRegister:(nonnull NSError
*)error;
- (void) vocService:(nonnull VocService *)vocService didRegister:(nonnull NSDictionary
*)info;
- (void) vocService:(nonnull VocService *)vocService didInitialize:(nonnull NSDictionary
*)info;
- (void) vocService:(nonnull VocService *)vocService itemsDiscovered:(nonnull NSArray
*)items;
- (void) vocService:(nonnull VocService *)vocService itemsStartDownloading:(nonnull NSArray
*)items;
- (void) vocService:(nonnull VocService *)vocService itemsDownloaded:(nonnull NSArray
*)items;
- (void) vocService:(nonnull VocService *)vocService itemsEvicted:(nonnull NSArray *)items;
- (void) vocService:(nonnull VocService *)vocService hlsServerStarted:(nonnull NSURL*)url;
- (void) vocService:(nonnull VocService *)vocService hlsServerStopped:(nonnull NSURL*)url;
```

The SDK delegate is passed to the SDK in the initialization call. Typically, this is the app delegate since its lifetime will span that of the SDK, from registration until shutdown. Define your app delegate as follows to implement the SDK delegate protocol.

```
@interface AppDelegate : UIResponder <UIApplicationDelegate, VocServiceDelegate>
```

4 Accessing Content

4.1 Data Download

Data is downloaded based on the catalog provided by PCD Server. Downloaded data includes three object types.

Data Object	Description
Content	A single video, song, HTML page, image, etc.
Category	A named group of related content items (e.g., “Sports”).
Source	Content providers configured at the Web portal. Every content item belongs to a source.

4.1.1 Prepositioned Content

The default behavior of the SDK is to preposition content onto the device once the user has registered. This happens automatically while your program runs.

4.1.2 User/Client initiated content download

User initiated download is required if the client has a huge catalog of content and/or all the content will not be downloaded. PCD SDK organizes content around client’s content id and the client will be able to initiate a download for a given content id regardless if the content metadata is present locally or not.

4.2 Working with Data Sets

Content, sources, and categories are all accessible through sets. Sets are created by requesting objects matching a list of filters. Examples of filters are “all sources,” “all downloaded content,” “all downloading content,” or “all categories.”

Observers can be added to these sets to be notified of changes. Pass one or more listeners to the set to be notified about the addition, deletion, or update of the set’s objects. Update notifications

include changes internal to the object, such as the state of a video changing from “downloading” to “cached.”

4.3 Content Items

Content may consist of video, audio, HTML, or other data files. Content may refer to a single file or multiple related files on disk (e.g., a video and its thumbnail image), but is represented as a unified item through the SDK. Each item has a title, summary, unique ID, and other properties depending on its type. Each content item has a single source and belongs to one or more categories. Content may be accessed through sets or individually.

4.3.1 Accessing Content from Data Sets

To create and track a collection of items, first define the set and subscribe to the set listener protocol.

```
@interface MyViewController () <VocObjSetChangeListener>
@property (strong, nonatomic) id<VocItemSet> itemSet;
@end
```

Now create the set using filters. The following sample creates a set containing all known items. It adds the *self* object as a set change listener in order to react to changes to the set or the set’s items.

```
[appDelegate.vocService.getItemsWithFilter:[VocItemFilter allWithSort:nil]
    completion:^(NSError * __nullable error, id<VocItemSet> __nullable set) {
    if (error) {
        NSLog(@"Error getting item set: %@", error);
    }
    self.itemSet = set;
    [set addListener:self];
}];
```

Items in the set can now be accessed directly through the set’s *items* property. The set filter was “allWithSort:” so some of these items may be categories or sources. The following loop identifies file items by testing whether they conform to the *VocItem* protocol. Items that pass contain a *file* property with specific file-related data. A file’s *localPath* property, for example, is a standard path on the device, usable for file operations such as loading into image data.

```

for (id<VocItem> vocItem in self.itemSet.items) {
    if ([vocItem conformsToProtocol:@protocol(VocItem)]) {
        id<VocFile> file = vocItem.file;
        NSLog(@"file size %lld at local path '%@'",
              file.bytesDownloaded, file.localPath);
    }
}

```

4.3.2 Accessing Content Individually

4.3.2.1 Using uniqueId

Each item retrieved from a data set has a uniqueId string property and this is the id assigned to content by PCD Server. The uniqueId may be saved for future lookups since it will not change.

To directly access the content again later, make the VocService call getItemWithId:. This inputs the unique ID and returns the item asynchronously, or nil if not found.

```

NSString *uniqueID = @"123456";
[appDelegate.vocService getItemWithId:uniqueID completion:^(NSError * __nullable error,
id<VocItem> __nullable item) {
    if (error || !item) {
        NSLog(@"Item unavailable.");
    }
    NSLog(@"Item '%@' share URL: %@", item.title, item.shareURL.absoluteString);
}

```

4.3.2.2 Using content_unique_id

Each content needs a content_unique_id during ingest and as that name would suggest, the id should be unique. VocItem has a property, 'contentId,' which returns the content_unique_id of the content. The same content_unique_id can be used for future VocItem lookups in the PCD SDK. To directly access the content again later, make the VocService call getItemWithContentIds:sourceName:completion. If the item is not available locally, the PCD SDK will request the item's metadata from the PCD server. The metadata requested is attempted three times in case of device failures such as a lost connection, with 10 seconds between attempts. Once the SDK receives a response from the server, the call returns the itemSet asynchronously via the completion block.

```

NSSet *contentIds = [[NSSet alloc] initWithObjects:@"354", @"452", @"475", nil];
[appDelegate().vocService getItemWithContentIds:contentIds
 sourceName:@"ProviderName" completion:^(NSError * error, id<VocItemSet> itemSet)
{
    if (error)

```

```

{
    //Error will be set if any requested items are not found.
}

//Add the listener
self.itemSet = itemSet;
[itemSet addListener:self];

if (0 < itemSet.items.count){
    //process those items that are returned
}
}];

```

4.4 Playing HLS Video Content

HTTP Live Streaming (HLS) video can be played from fully or partially downloaded content. Playback requires starting the SDK's on-device HLS server. From that server, a playback link is then requested for the content.

Clients start the HLS server by calling `VocService::startHLSServerWithCompletion:completion` before playing the HLS content. The completion block is executed only after the HLS server is started. In the passed completion block, the client can access the HLS server URL and append to it the content item's relative URL. Feed this URL to the media player to play the HLS content. Once playback is finished, stop the media server by calling `VocService::stopHLSServer()`. The client controls the starting and stopping of the HLS server.

For example:

```

//Start HLS server
if ([vocItem conformsToProtocol:@protocol(VocItemHLSVideo)]) {
    [appDelegate.vocService startHLSServerWithCompletion:^(BOOL success){
        dispatch_async(dispatch_get_main_queue(), ^{
            id<VocItemHLSVideo> hlsVideo = (id<VocItemHLSVideo>)vocItem;
            NSURL *url = [appDelegate.vocService.hlsServerUrl
URLByAppendingPathComponent:hlsVideo.hlsServerRelativePath];
            //Set the media player's content URL to this URL
        });
    }];
}

```

```

//Stop HLS server
[appDelegate.vocService stopHLSServer];

```


4.5 Playing Non-HLS Content

Clients can play content from a URL if the item is not yet cached or from the local path if the item is cached.

For example:

```
NSURL *url = nil;
if ([vocItem conformsToProtocol:@protocol(VocItemVideo)]) {
    id<VocItemVideo> videoVocItem = (id<VocItemVideo>)vocItem;
    if (videoVocItem.state != VOCItemCached) {
        //item is not cached
        url = videoVocItem.file.url;
    } else {
        url = [NSURL fileURLWithPath:videoVocItem.localPath];
    }
}
//Set the media player's content URL to this URL
```

4.6 Content Item States

Each item progresses through different states before getting cached. VocItem has a *state* property which can take the following values.

VOCItemState	Description
VOCItemDiscovered	Item has been identified. Various device policies dictate whether the item gets queued for download.
VOCItemQueued	Item has been added to downloading queue; it is currently not downloading. It will start downloading.
VOCItemDownloading	Item is being downloaded.
VOCItemIdle	Item is currently not downloading. Item has been previously queued for downloading but did not finish. Download will resume automatically at some point.

VOCItemPaused	Item is currently not downloading. It has been explicitly paused with VocService pauseItems:.
VOCItemCached	Item download was completed successfully. Cached content is ready for use.
VOCItemFailed	If PCD SDK is unsuccessful to download any main content file after three retry attempts, the item will be marked as VOCItemFailed and the downloaded content will be removed. No more download attempts will be made on the item, unless an explicit download for this item is triggered again.
VOCItemDeleted	Item has been deleted and cannot be used any more. There are no cached files.
VOCItemPartiallyCached	Item has been cached for a specified duration. User can download the complete duration by initiating download.

Note: Another VocItem property, *downloadError*, can be used to verify the reason for download failure or download suspension.

4.7 Start/Cancel Content Downloads

The default download behavior of PCD SDK is to manage the content download by itself and the client does not need to initiate the download. The client can override the default PCD SDK download behavior using [SDK Settings API](#). Also, VocItem has a property, 'downloadBehavior,' that defines the download behavior of an item and the default value of the property is to follow the download behavior defined in SDK settings. The client can use 'downloadBehavior' property of an item to override the SDK download behavior.

```

vocItem.downloadBehavior = VOCItemDownloadFullAuto;

```

If the client has a use case to explicitly start content downloads, it can make the VocService call startDownloadUserInitiatedWithCompletion:.

```

[appDelegate.vocService

```

```
startDownloadUserInitiatedWithCompletion:^(NSError* error) {}];
```

Also, the client can cancel all the on-going download by making the VocService call `cancelCurrentDownloadWithCompletion:`.

```
[appDelegate.vocService cancelCurrentDownloadWithCompletion:nil];
```

4.8 Start Downloading a Specific Item

Once the client has access to the item or items, the client can download the item. Individual item download is not allowed if the item is already cached or deleted or is currently downloading. If the client has set the item download behavior of the SDK as `VOCItemDownloadNone`, the client must override the download behavior of the individual item for downloading. The client can start the download by making the VocService call `downloadItems:options:completion:`. One parameter of this method is the `downloadBehavior` applied to all items requested for download. Calling this method with multiple items results in the same queuing behavior as calling it multiple times with a distinct item each time. Note that calling this method repeatedly for the same content will not result in downloading the corresponding video multiple times.

```
[appDelegate.vocService downloadItems:@[vocItemHLSVideo1, vocIvocItemHLSVideo2]
                             options:@{@"videoPartialDownloadLength" : @(0),
                                         @"downloadBehavior" :
                                         @(VOCItemDownloadFullAuto)}
 completion:^(NSError * _Nullable error) {
 }];
```

4.9 Pause/Resume Downloading a Specific Item

The client can pause a specific content item download by making the VocService call `pauseItemDownload:`. `VocItem` has a readonly property, 'paused,' which the client can use to identify items that are paused. Alternatively, the same pause operation can be performed on one or more `VocItems` by making the VocService call `pauseDownloadForItems:completion:`. A paused download will not resume until a specific download request has been made.

```
[appDelegate.vocService pauseItemDownload:vocItemVideo completion:^(NSError
* _Nullable error) {
 }];
```

The client can resume the download of an item or items by making the VocService call `downloadItems:`.

```
[appDelegate.vocService downloadItems:@[vocItemVideo] completion:^(NSError *
    _Nullable error) {
}];
```

4.10 Track Progress of a Downloading Video

There are two properties in `VocItem` that will allow the client app to calculate download progress.

- `bytesDownloaded` - total number of bytes downloaded for the item.
- `size` - total size of the item.

The client app can listen for `bytesDownloaded` changes by subscribing to the `VocObjSetChangeListener` protocol (Refer [Listening for Data Set Changes](#)) or using key-value observing (KVO). Alternatively, the client app can use a timer to periodically poll `bytesDownloaded` and update the progress.

4.11 Content Sources

A content source identifies the provider (e.g., a channel, company, or aggregator) of content items.

Sources are defined on the SDK Web portal. The SDK allows sources to be toggled on or off.

To track changes to a source set, subscribe your interface to the `VocObjSetChangeListener` protocol. Also define a property to reference the source set.

```
@interface MyViewController () <VocObjSetChangeListener>
@property (strong, nonatomic) id<VocSourceSet> vocItemSourceSet;
@end
```

Create the source list in your implementation. This needs to be called only once per set, such as in `-viewDidLoad` for a view controller that uses sources.

```
[vocService getSourcesWithFilter:[VocItemSourceFilter allWithSort:nil]
    completion:^(NSError * __nullable error, id<VocSourceSet> __nullable sourceSet) {
    if (error) {
        NSLog(@"Error getting sources: %@", error);
        return;
    }
    [sourceSet addListener:self];
    self.vocItemSourceSet = sourceSet;
}];
```

The code above sets self as a listener to the set, which is possible because we follow the *VocObjSetChangeListener* protocol. This will notify our listener when the set will change and when it did change. We may also access individual sources by stepping through the source container, *vocItemSourceSet.sources*.

```
for (id<VocItemSource> source in self.vocItemSourceSet.sources) {
    if (source.subscribed) {
        NSLog(@"Subscribed to %@", source.displayName);
    }
}
```

4.12 Content Categories

Categories are named groups of related content, such as “family,” “trending,” or “news.” Working with a category makes it easy to display dedicated views for broad content types, or to toggle the display an entire group.

To create and follow a category set, subscribe your interface to the *VocObjSetChangeListener* protocol (the same as for sources). Also define a property to reference the category set.

```
@interface MyViewController () <VocObjSetChangeListener>
@property (strong, nonatomic) id<VocCategorySet> vocItemCategorySet;
@end
```

Create the category list in your implementation. This needs to be called only once per category set, such as in a view controller’s *-viewDidLoad* method. As shown for sources, this example adds our self object as a listener and retains a reference to the category set.

```
[vocService getCategoriesWithFilter:[VocItemCategoryFilter allWithSort:nil]
completion:^(NSError * __nullable error, id<VocCategorySet> __nullable categorySet) {
    if (error) {
        NSLog(@"Error getting categories: %@", error);
        return;
    }
    [categorySet addListener:self];
    self.vocItemCategorySet = categorySet;
}];
```

Categories may be iterated over by using *vocItemCategorySet.categories*.

```
for (id<VocItemCategory> category in self.vocItemCategorySet.categories) {
    if (category.selected) {
        NSLog(@"Selected category %@", category.displayName);
    }
}
```

4.13 Listening for Data Set Changes

Data sets send notifications when items are added, changed, or removed. This is the best way to detect when individual items have finished downloading. To listen for object set changes a class must subscribe to the *VocObjSetChangeListener* protocol.

```
@interface MyClass () <VocObjSetChangeListener>
```

Setting the data set listener is done via *-addListener:* and is demonstrated in the sections below for sources, categories, and items. Listeners receive callbacks before and after any changes.

```
-(void)vocService:(nonnull VocService *)vocService objSetWillChange:(nonnull
id<VocObjSet>)objSet
    added:(nonnull NSSet*)added
    updated:(nonnull NSSet*)updated
    removed:(nonnull NSSet*)removed
objectsAfterChanges:(nonnull NSArray*)objectsAfterChanges
{
    if ([objSet isEqual:self.itemSet]) {
        NSLog(@"%ld items will change", (unsigned long)updated.count);
    }
}
```

```
-(void)vocService:(nonnull id<VocService>)vocService objSetDidChange:(nonnull
id<VocObjSet>)objSet
    added:(nonnull NSSet*)added
    updated:(nonnull NSSet*)updated
    removed:(nonnull NSSet*)removed
```

```

objectsBefore:(nonnull NSArray*)objectsBefore
{
    if ([objSet isEqual:self.itemSet]) {
        NSLog(@"%ld items have changed", (unsigned long)updated.count);
    }
}

```

4.14 Data Set Filters

Each of the set-building methods accepts a filter that defines the set. In each example above we used the “all” filter to get a complete set of items. Specific subsets can also be created by using alternative filters. These sets will behave as before but will notify their listeners only when the filter is matched.

Object Type	Filter Protocol	Available Filters
Source	VocSourceFilter	allWithSort:
Category	VocItemCategoryFilter	allWithSort: categoriesSelected: categoriesSelectedWithContent: subCategories:SearchField: suggestions:
Item	VocItemFilter	allWithSort: itemsPartiallyDownloadedWithSort: itemsDownloadedWithSort: itemsDiscoveredWithSort: itemsQueuedWithSort: itemsDownloadingWithSort: itemsIdleWithSort: itemsPausedWithSort: itemsFailedWithSort: itemsInStates:sortDescriptors: itemsWithCategory:sortDescriptors: itemsWithContentIds:sourceName:sortDescriptors: itemsDownloadedWithCategory:sortDescriptors: itemsInStates:withCategory:sortDescriptors: itemsSavedWithSort: itemsDownloadedNotViewedWithSort: itemsInStates:notViewedWithSort:sortDescriptor:
Video (subset of Item)	VocItemVideoFilter	allWithSort: videosPartiallyDownloadedWithSort:

		videosDownloadedWithSort: videosDiscoveredWithSort: videosQueuedWithSort: videosDownloadingWithSort: videosIdleWithSort: videosPausedWithSort: videosFailedWithSort: videosWithCategory:sortDescriptors: videosDownloadedWithCategory:sortDescriptors: videosDownloadedOrDownloadingWithCategory: sortDescriptors: videosInStates:withCategory:sortDescriptors: videosSavedWithSort: videosDownloadedNotViewedWithSort: videosInStates:notViewedWithSort:sortDescriptor:
--	--	--

Filters used for creating sets

Most filters also accept an array of sort descriptors. The .sources, .categories, and .items properties within their respective sets are each an ordered array. The sort descriptor defines the order of those arrays. Pass a nil descriptor if order is unimportant.

```

NSArray* sortDesc = @[[[NSSortDescriptor alloc] initWithKey:@"name" ascending:YES
selector:@selector(caseInsensitiveCompare:)]];

[vocService getSourcesWithFilter:[VocItemSourceFilter allWithSort:sortDesc]
completion:^(NSError * __nullable error, id<VocSourceSet> __nullable set) {
    // set.sources is ordered by each source's "name" property
}];

```

4.15 Delete/Lock/Save an Item

PCD SDK provides two delete options for each item.

1. `VocItem::deleteItem`: deletes both the metadata and files associated with the item.
2. `VocItem::deleteFiles`: deletes just the files associated with the item. Since the metadata is not removed, the client can re-download the video at any point after deleting the files. Alternatively, the same operation can be performed on one or more `VocItems` by making the `VocService` call `deleteFilesForItems:options:completion:`. Optionally, the app can retain the video's thumbnail by setting `deleteThumbnail` parameter to `NO`.

Some content consumers exhibit unexpected behavior if the item gets deleted while it is consumed. The SDK doesn't know whether the item is currently consumed unless it is notified explicitly. It is

recommended to call `VocItem::lock` before the item gets consumed and `VocItem::unlock` after the consumption is completed.

PCD SDK initiates a purge mechanism based on certain device parameters. To avoid an item getting deleted using the purge mechanism, the user can mark the item as saved by calling `VocItem::save`.

4.16 Record the Video Consumption by the user

As per the business contract, when user completes watching a video fully or partially the client must report the consumption to the PCD SDK.

```
//When user navigates back from the video player
if ([vocItem conformsToProtocol:@protocol(VocItemVideo)]) {
    id<VocItemVideo> video = (id<VocItemVideo>)vocItem;
    if (video) {
        //The total watched duration of the video
        NSTimeInterval endPosition = CMTimeGetSeconds(player.currentItem.currentTime);
        //If the client bookmarks the end position of a video play and user resumes
        //watching from that position, then the start position is the bookmarked time.
        NSTimeInterval startPosition = 0;
        [video recordConsumption:[NSDate date] startingAt:startPosition endingAt:endPosition];
    }
}
```

4.17 Modify File Download Request

The client can implement a PCD SDK delegate method that allows the client to modify the download request of a file before it is queued for downloading. The client can achieve this by implementing `VocServiceDelegate::vocService:convertDownloadRequest:item:file:completion:`.

```
@interface AppDelegate () <VocServiceDelegate>
@end

@implementation AppDelegate

- (void) vocService:(nonnull id<VocService>)vocService convertDownloadRequest:(nonnull
NSMutableURLRequest*) request
    item:(nonnull id<VocItem>)item
    file:(nonnull id<VocFile>)file
    completion:(nonnull void (^)(nullable NSMutableURLRequest *request))completion
{
    switch ([file getFileType]) {
        case VocMainFile:

```

```

        case VochLSMainIndex:
            [request addValue:@"abc=123;" forHTTPHeaderField:@"Set-Cookie"];
            break;
        default:
            break;
    }
    completion(request);
}

```

4.18 Accessing Subtitles and Thumbnails

The client can make Voltem call `thumbnailsWithCompletion:` to access a collection of thumbnail associated with the item. Similarly, the client can make `VocItemVideo` call `subtitlesWithCompletion:` to access a collection of subtitle associated with the item. The `VocItem` state is not affected by the download state of thumbnail or subtitle files. If `autoDownload` property for these files are set to YES, the file will get downloaded on the item download request.

```

[vocItemVideo subtitlesWithCompletion:^(NSSet<id<VocFile>>* _Nullable subtitles) {
    if (subtitles) {
        for (id<VocFile> subtitle in subtitles) {
            if (!subtitle.autoDownload) {
                subtitle.autoDownload = YES;
            }
        }
    }
    [appDelegate.vocService downloadItems:@[vocItemVideo] completion:nil];
}];

```


4.19 Storing additional information on each item

`VocItem` has a property, `'persistentUserInfo,'` which can be used by the client to store additional information about an item and this information will be persisted in PCD SDK. `VocItem` has a similar property, `'userInfo,'` which can also be used by the client to store additional information on an item but it will be stored only in the memory.

```

NSMutableDictionary *userInfo = @{
    @"eventData": @{
        @"profileAccessDate": @"Jun 13, 2012 12:00:00 AM",
    }
};

```



```
};  
};  
vocItem.persistentUserInfo = userInfo;
```

Note that the above properties are in addition to the VocItem property 'sdkMetadataPassthrough,' which holds extra information added to the item during ingest. This property is modifiable.

5 SDK Settings

VocService has a property, 'config,' which provides a rich set of API to configure the SDK behavior. Default values for some of the configurations in VocConfig.h are set at the global level (to server values) and you could override those at the client level. You should discuss with the PCD SDK team about your default value preferences for permitted network types for download (wifi, cellular, or both), content items per category, daily download size limit, maximum size allowed per file download, permitted time of day for downloading, etc. Some of the important configurations are the following

5.1 Network Preference

The API provides control over whether downloads are permitted on wifi, cellular, or both. Setting the VocConfig *networkSelectionOverride* property affects downloads that have not already started. Use VocConfig::networkSelection to access the applied network selection.

```
self.vocService.config.networkSelectionOverride = VOCNetworkSelectionWifiAndCellular;
```


5.2 Item Download Behavior

The default behavior of the SDK is to preposition all the content from the content catalog. VocConfig has a property, 'itemDownloadBehavior,' which provides the client an option to define the download behavior of the PCD SDK. This configuration can be configured to restrict the auto download of the content and the user will be responsible for downloading the content. The content catalog will still be downloaded after registering the PCD SDK. The client has to make an explicit call to trigger a content download from the catalog. The SDK download behaviors are auto download, download of thumbnail files only and no download.

```
self.vocService.config.itemDownloadBehavior = VOCItemDownloadThumbnailOnly;
```

5.3 Auto Purge

The default behavior of the SDK is to perform content purge based on certain device specific parameters. VocConfig has a property, 'enableAutoPurge,' which provides the client an option to disable auto purge and the user will be responsible for the deletion of the content.



```
self.vocService.config.enableAutoPurge = NO;
```

Maximum Number of Concurrent Downloads

The default behavior of the SDK is to perform any number of concurrent file downloads. VocConfig has a property, 'maxConcurrentDownloads,' which provides the client an option to restrict the number of simultaneous downloads.

```
self.vocService.config.maxConcurrentDownloads = 10;
```



6 Download Policy

Whether the SDK permits downloads at this time can be determined from the `VocService` boolean property *downloadAllowed*. This is continuously updated as device conditions change, such as loss of network, the battery level dropping too low, or running out of drive space. To see the specific download status, access the `VocService` enumerated value *downloadPolicyStatus*. The status when downloads are allowed is `VOCPolicyStatusInPolicy`.

7 Content Migration

The client can make the VocService call `addDownloadedItem:completion:` for importing pre-existing media content into the PCD SDK. PCD SDK supports migration before and after registration for completely or partially downloaded content. For partially downloaded content, the SDK does not import the files but only the content metadata.

```
//Assuming the content files that need to be migrated are located in the Documents directory
NSArray *searchPaths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
NSString *contentPath = [searchPaths objectAtIndex:0];
contentPath = [contentPath stringByAppendingPathComponent:@"Contents"];
NSString* thumbFilePath = [contentPath stringByAppendingPathComponent:@"_thumb.jpg"];
NSString *documentPath = [contentPath stringByAppendingPathComponent:@"content.m3u8"];
NSMutableDictionary* info = [[NSMutableDictionary dictionary] mutableCopy];
info [VocImportFileMetadataContentId] = @"12345678";
info [VocImportFileMetadataProvider] = @"Provider_ABC";
Info [VocImportFileDownloadStatus] = @(VocMigratingContentFullyDownloaded);
info [VocImportFileMetadataFilePath] = documentPath;
info [VocImportFileMetadataDuration] = @"1000";
info [VocImportFileMetadataSize] = @(2000);
info [VocImportFileMetadataContentType] = VOCItemTypeHLSVideo;
//SDK assumes that the segment files, key files for HLS media are present in the same
directory which has the local manifest.
info [VocImportFileMetadataContentUrl] = @"https://../NewContent.m3u8";
info [VocImportFileMetadataTitle] = @"New Downloaded Content";
info [VocImportFileMetadataSummary] = @"New Downloaded Content Summary";
info [VocImportFileMetadataThumbFile] = thumbFilePath;
info [VocImportFileMetadataExpiryDate] = @(1481876040);
info [VocImportFileMetadataTimeStamp] = @(1481607060);
info [VocImportFileMetadataDownloadFinishedTime] = @(1481775040);
info [VocImportFileMetadataSDKPassthroughInfo] = @"saved=YES,liked=YES";

[appDelegate().vocService addDownloadedItem:info completion:^(NSError * _Nullable err,
id<VocItem> _Nullable vocItem) {
    if (!err) {
        NSLog(@"Imported item title: %@",vocItem.title);
    }else{
        NSLog(@"Import failed, error message: %@",err.localizedDescription);
    }
}
}];
```

8 Appendix - Requirements and Dependencies

8.1 Background Execution

The PCD SDK downloads content while the application is running. Various factors determine when to start downloading, how much to download, and when to pause downloads. Influencing factors include the state of the mobile network and the quality state of the provider network.

When your app is in the foreground, downloads are happening without any need for changes in your code. However, in order to get best results, the PCD SDK should be able to download when your app is not in the foreground. In order to do that, you need to enable your app for background execution. There are two types of background execution that the SDK uses to download content. Enable these two modes within the Xcode project settings, *Capabilities* tab, *Background Modes* section:

- Remote notifications (remote-notification)
- Background fetch (fetch)

To enable background fetch in the PCD SDK, you need to implement the system method `UIApplicationDelegate application:performFetchWithCompletionHandler:`

and, from there, pass the message to the `VocService` by calling `VocService application:performFetchWithCompletionHandler:`

Here is what that looks like:

```
- (void)application:(UIApplication *)application performFetchWithCompletionHandler:(void (^)(UIBackgroundFetchResult result))completionHandler
{
    if (![self.vocService application:application
performFetchWithCompletionHandler:completionHandler]) {
        completionHandler(UIBackgroundFetchResultNoData);
    }
}
```


8.2 Remote Notifications

To make remote notifications work in the PCD SDK you need to 1) enable the SDK backend to send push notifications to your app, and 2) make the supporting code changes.

To enable the PCD backend to send push notifications to your app, you need to upload your app push certificate (APNS) to the PCD license management portal. The PCD backend must have a valid APNS certificate for your app at all times otherwise push notifications will not work. If you revoke or renew your certificate, make sure you upload it to the PCD license management portal. Instructions on how to generate an APNS push certificate are available on Apple's web site at: https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/AddingCapabilities/AddingCapabilities.html#//apple_ref/doc/uid/TP40012582-CH26-SW11

The first step for your app is to register for remote notification and hand the push token to the SDK in the AppDelegate call *application:didRegisterForRemoteNotificationsWithDeviceToken:*.

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    //the client must register for remote notifications
    [[UIApplication sharedApplication] registerForRemoteNotifications];

    //the client must call this method during launch cycle
    //to request permission from user to use notification
    UIUserNotificationType notificationType = UIUserNotificationTypeAlert;
    [[UIApplication sharedApplication] registerUserNotificationSettings:
        [UIUserNotificationSettings settingsForTypes:notificationType categories:nil]];
}

- (void)application:(UIApplication*)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData*)deviceToken
{
    //pass push token to SDK
    [self.vocService setDevicePushToken:deviceToken];
}
```

Also in your application delegate, implement the system method:

```
UIApplicationDelegate application:didReceiveRemoteNotification:fetchCompletionHandler:
```

and pass this notification to the VocService:

```
VocService application:didReceiveRemoteNotification:fetchCompletionHandler:
```

For example:

```
- (void)application:(UIApplication *)application didReceiveRemoteNotification:(NSDictionary *)userInfo fetchCompletionHandler:(void (^)(UIBackgroundFetchResult result))completionHandler
{
    if ([self.vocService application:application didReceiveRemoteNotification:userInfo fetchCompletionHandler:completionHandler]) {
        // remote notification was for VoC SDK
        return;
    }

    // remote notification is not for VoC SDK, handle remote notification
    completionHandler(UIBackgroundFetchResultNoData);
}
```

If the client is testing against the development APNS instead of production APNS, the SDK needs to be configured to use development APNS before the SDK registration.

```
vocService.config.useProductionApns = NO;
```

Remote notification and fetch are enabled in background by adding the following key-subkey to the app's info.plist file

```
<key>UIBackgroundModes</key>
    <array>
        <string>fetch</string>
        <string>remote-notification</string>
    </array>
```

8.3 App Transport Security

Starting in iOS 9.0, a new app security feature called App Transport Security (ATS) has been introduced by Apple and it is enabled by default. With ATS enabled, connections must use secure HTTPS instead of HTTP. Additionally, if the app contents that PCD SDK needs to download contain non-SSL items, those downloads will fail. Application developers must ensure that either

1. [preferred] HTTPS is used for all content URLs, or
2. [insecure] ATS exceptions can be added for certain domains by adding the following key-subkey to the app's info.plist file.

Optional ATS key to allow insecure (HTTP) content:

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSExceptionDomains</key>
    <dict>
        <key>pvoc-anaina.com</key>
        <dict>
            <key>NSIncludesSubdomains</key>
            <true/>
            <key>NSExceptionAllowsInsecureHTTPLoads</key>
            <true/>
        </dict>
    </dict>
</dict>
</dict>
```

8.4 Call force fill on registered SDK

If the client restricts SDK's default content prepositioning behavior, it must make the VocService call `startDownloadUserInitiatedWithCompletion:` to keep the user as an active one. Perform this call only once per app launch and only if SDK is registered.

```
if (self.vocService.state != VOCServiceStateNotRegistered) {
    [appDelegate.vocService
     startDownloadUserInitiatedWithCompletion:^(NSError* error){
    }];
}
```

8.5 Avoid server response delay for initiating download

For clients that follow a user initiated content download approach using PCD SDK, the client makes the VocService call `getItemWithContentIds:sourceName:completion` to create the content item in PCD SDK (see [Using ContentUniqueid](#)). When the API call is made by the client, the PCD SDK interacts with the PCD server to retrieve the content information. This asynchronous call might encounter a delay depending upon the network. If the client requires the item object promptly for download, the client can make the VocService call `addDownloadedItem:completion:` to get an item that will be immediately queued for download. This call will internally update the item with the server information and then initiates the item download. When adding a new item for the discussed use case, there is an item information dictionary that needs to be prepared as a parameter for `addDownloadedItem:completion:` (see [Content Migration](#)). The client must add the *VocImportFileDownloadStatus* key in the information dictionary with the value as *VocMigratingContentNewlyAdding*. Also, the item information dictionary must have correct type, size and preferred quality information.