KDP plug-ins

## INTRO

One of the things that makes the KDP as dynamic as it is, is its built-in capability of loading and using plug-ins that complement its own capabilities. Plug-ins can be used for a multitude of purposes: additional media playback capabilities (i.e OSMF media plugins, such as the WV plug-in or the Akamai http-streaming plug-in), analytics reports (i.e Statistics plug-in, the Omniture plug-in), and enriching the video experience (i.e the Playlist plug-in, the Closed Captions plugin and more).

## Terminology used in this document

1. KML – refers to the special **K**altura **M**arkup **L**anguage used in the KDP layout xml files.
2. KML file/Layout XML/ UIConf xml – all 3 terms reference the same thing in this document: the xml file describing the KDP visual layout.
3. {YourClassPlugin}, {YourClassPluginCode} – naming conventions for the 2 main classes comprising the KDP plug-in. For instance – statisticsPlugin, StatisticsPluginCode.

# PLUGIN DECLARATION IN KML

The plug-in declaration in the uiconf xml serves to indicate to the KDP that the plug-in must be loaded during the KDP bootstrapping process. The declaration must also contain 2 types of data:

1. Parameters that will be passed to the plug-in itself. For example – if my plug-in requires a parameter "user" it will be passed here with its according value. These parameters will be touched upon more extensively later in this document.
2. Parameters that concern the loading process of the plug-in .swf. These parameters concern the KDP PluginManager class, which loads the plug-in, and **they have nothing to do with the plug-in itself.** After the plug-in I loaded, these parameters will be inaccessible to it. These parameters include:

* id – id of the plug-in. Must be a unique value. For instance : id="statistics"
* name – the name of the plug-in.
* path – the path from which the plugin is loaded. The rules for this parameter are:
* If no value was passed – use the path {flashvars.pluginDomain}\{id}Plugin.swf. I.E: plugins\statisticsPlugin.swf.
* If absolute URL was passed – use the absolute URL.
* If non-absolute URL starting with the “/” character was passed – the KDP’s httpProtocol and cdnHost flashvars are concatenated to the front of the URL. I.E: path="/content/uiconf/demos/demoPlugin.swf" and the flashvars specify httpProtocol="http", cdnHost="cdnakmi.kaltura.com" 🡪 the plug-in is loaded from <http://cdnakmi.kaltura.com/content/uiconf/demos/demoPlugin.swf>.
* If non-absolute URL starting with any character but “/” was passes – the KDP attempts to load the plugin from the {flashvars.pluginDomain} directory. This is useful if attempting to load several instances of same plug-in into the same uiconf, since it allows for several different id’s while loading them all from the same path.
* loadingPolicy – the mode of the plug-in load. This parameter indicates whether the PluginManager class should “freeze” the bootstrapping process until the plug-in is successfully loaded (or fails to load). Available values:
* wait – default value. Indicates that the initialization of the player must be paused until this plug-in is reported as loaded or failed to load.
* nowait – indicates that the loading process of the KDP dos not need to be paused for this plug-in. After starting the plug-in’s load, the KDP may resume its initialization process, even of the plug-in has not yet loaded.
* onDemand – indicates that the user will initiate the plug-in load through some means (another plug-in/external interface) and the PluginManager of the KDP does not need to load this plug-in. Used for bulky plug-ins that are not always necessary, such as the KalturaMixPlugin.
* preInitialize – indicates that the plug-I should be loaded before the skin.swf file (the plug-ins re normally loaded after the skin). This exists in case the plug-I is requires to replace some visual component class.
* asyncInit – ["true"/"false"] -flag indicating whether the plug-in should report its own initialization as complete/failed, rather than the initialization being considered automatically complete when the plug-in has loaded. This parameter was designed for plug-ins that require an asynchronous process to be performed in their initialization process.

The KDP plug-ins may come in 2 varieties: visual and non-visual (logic only). The only thing that separates the 2 varieties are visual attributes assigned to them in the KML file.

* visual plug-ins - example tag:

<Plugin id="visualPlugin" width="{substantial value}" height="{substantial value}" …/>

Note that this plug-in tag is a stand-alone component – it is up to you to position it in the appropriate container in your KML file.

* non-visual plug-ins – example tag:

<Plugin id="visualPlugin" width="0%" height="0%" includeInLayout="false" …/>

includeInLayout="false" – indicates that this plug-in must to occupy any amount of space in the layout. The best practice for non-visual plug-ins is to position them in the first HBox/VBox type container in the layout KML file.

# HOW TO CREATE A PLUGIN

The following 3 steps are all that is necessary to start a simple KDP plug-in:

1. Check out the kdp3Lib project and add it as a project to your plug-in’s library path (right-click on the plug-in project folder 🡪 properties 🡪 ActionScript Build Path 🡪 Library path).
2. Open a new ActionScript project.
3. Add the classes {YourClassPlugin} & {YourClassPluginCode} to your new project.
4. If your plug-in expects to receive attributes from the uiconf xml (the parameters from (1) of the previous section), add them as properties to the YourClassPluginCode class. The KDP will populate them during layout build process.

**YourClassPlugin** – this is the document class first loaded by the KDP3, this class must implement the IPluginFactory interface.

public interface IPluginFactory

{

/\*\*

\* this function creates a new IPlugin instance. It is implemented by the plugin application class

\* @param pluginName the name of the plugin to be created in case there are multiple plugins within

\* the same plugin swf file

\* @return

\*

\*/

function create(pluginName : String = null) : IPlugin;

}

When the method “create” is called the YourClassPluginCode is instanciated.

YourClassPluginCode – this class must implement the IPlugin interface. This interface allows the KDP to properly initialize the plug-in and set all of its attributes from the uiconf xml tag.

public interface IPlugin

{

/\*\*

\* This function is automatically called by the player after the plugin has loaded.

\* the facade used to communicate with the hosted player

\* @param

\*

\*/

function initializePlugin( facade : IFacade ) : void

/\*\*

\*

\* @param styleName

\* @param setSkinSize

\*

\*/

function setSkin( styleName : String , setSkinSize : Boolean = false) : void;

}

When the plug-in is first created, the KDP uses the attributes found on its KML tag to set the YourClassPluginCode properties. During this process, the plug-in’s **setSkin** function is also called. His function’s purpose is to render the plug-in’s pre-defined visual classes from the skin.swf file. For an example of this, you might take a look at the List plugin, which provides a visual representation for the KDP playlist.

The “**initializePlugin**” function is called after all of the YourClassPluginCode properties were set. This function receives a reference to the KDP’s façade – the KDP instance of the ApplicationFacade, which is the pureMVC equivalent of the Controller in the MVC design pattern. This enables the plug-in to establish listeners to the KDP notifications.

If the plug-in is visual, rather than merely logical, this class is also the main visual component added to the player layout. Make sure to add any other visual components that you wish to showas part of the plug-in to this class instance.

It is also a best practice for KDP plug-ins to define a Mediator for the plug-in, whether it is a visual component in the KDP layout or not. The Mediator provides the plug-in with the capability to listen to the KDP’s events (notifications), which allows the plug-in to respond to key events occurring in the player. You can look at any Mediator in the KDP to see exactly how it is defined.

The best practice to defining a plug-in’s mediator is:

private var \_myPluginMediator : MyPluginMediator;

**public** **function** initializePlugin(facade:IFacade):**void** {

\_myPluginMediator= **new** MyPluginMediator(this);

facade.registerMediator(\_myPluginMediator);

**.**

**.**

**.**

Note: the façade is only used to register the mediator. After this, the mediator class automatically contains a reference to it.

# USING KALTURA CLIENT

The Kaltura Client allows the component using it to reference and use all available Kaltura API callbacks. In order to use this functionality in your plug-in, do the following:

1. Check the as3FlexClient library out from the svn repository.
2. Build the as3FlexClient, make sure there are no comple-time errors.
3. Right-click on your plug-in project folder 🡪 properties 🡪 ActionScript Build Path 🡪 Library path 🡪 Add project.. 🡪 select “as3FlexClient”.

Your plug-in now holds a reference to the Kalura API callbacks. In order to use them, you need a reference to the Kaltura Client instance used by the KDP itself. Getting this instance demands you define a Mediator class for your plug-in. The Kaltura Client can be retrieved using the ServicesProxy class:

**private** **var** \_kalturaClient : KalturaClient

.

.

.

**override** **public** **function** onRegister():**void**

{

\_kalturaClient = (facade.retrieveProxy(ServicesProxy.NAME) **as** ServicesProxy).kalturaClient;

.

.

.

}

# Optimization

The KDP plug-ins can be optimized for the kdp3Lib project in order to significantly reduce the size of their compiled .swf files. Here is how you do this:

Right-click on your plug-in project folder 🡪 Properties 🡪 ActionScript Compiler tab 🡪 in the “additional compiler arguments” line, enter:

-load –externs ${DOCUMENTS}\KDP3\bin-debug\linkReport.xml