**This guide is intended for:**

* **KDP version 3**
* **api\_v3 (aka partner services 3).**

**Background**

uiConf - User Interface Configuration file  
The uiConf is a general name for xml files through which it is possible to customize the KDP for a specific need, layout and design. The uiConf affects three types of features:

* **Layout** - entirely defined by the uiConf files and is responsible for placing objects on stage (e.g. location of the play button).
* **Functionality** - partially defined by the uiCong files and partially hard-coded into the KDP widget and its modules.
* **Look & Feel** - defined in a compiled swf skin file that is linked on the uiConf file.

**uiConf ingestion - How to create a modified KDP and use it ?**

A custom KDP uiConf can be created in three main ways -

* Using the Application Studio inside the [KMC](http://kaltura.com/kmc).
* Creating a uiConf xml file locally and uploading it to the Kaltura Server using the APIs.
* Utilizing Kaltura's professional services and ordering a custom KDP design - [Contact us](http://corp.kaltura.com/about/contact).

To use a custom uiConf one must create his own version of a uiConf xml file and then use the public Kaltura API to upload the new uiConf and assign a widget id to be used as the new KDP instance identifier.  
When embedding the KDP, specify the created widget id on the src attribute inside the <[**object**](http://december.com/html/4/element/object.html)> tag.

**APIs used**

* session.start - to create a new session to work with the Kaltura API.
* uiConf.add - to upload the new uiConf xml and create a uiConf object on the server that later can be used.
* widget.add - to generate a widget representation of the uiConf.

**Creating a new uiConf using the API**

To create your own KDP widget with a customized uiConf XML follow these steps -

* Use the following set of definitions and rules (described in this document) to create your own uiConf.
* It is highly suggested to download an existing uiConf to use as a base for your work and extend or modify it.
* Call the following API calls in the following order -
  + session.start - pass this API call with your partner id, user id and **admin** secret, and specify this is an ADMIN type KS.
  + uiConf.add - this will create a new uiConf in the system, this uiConf id will be used when requesting the KDP from the Kaltura Server.
  + widget.add - this API is not mandatory, but is good practice. It will take your newly created uiConf id as a parameter and create a dedicated widget id to be used when requesting a KDP instance from the Kaltura Server.
* [Embed your KDP](http://www.kaltura.org/kdp-basic-usage-embedding-html-pages-and-playback) using the newly created widget id:

*<!--// replace “your\_widget\_id” with the widget id created by the widget.add api call -->*  
<[**object**](http://december.com/html/4/element/object.html) src=“http://www.kaltura.com/kwidget/wid/your\_widget\_id” ... >  
...  
</[**object**](http://december.com/html/4/element/object.html)>

**The uiConf XML**

Parts of the KDP uiConf were inspired by MXML of Adobe Flex, however KDP uiConf presents various modifications and changes to the standard to comply for the KDP needs as a dynamic player application.  
  
The KDP uiConf is defined according to the following rules:

1. **The <layout> Tag** - The root element in every KDP uiConf XML.
   * Every uiConf file must be wrapped inside a <layout> tag.
   * Every uiConf xml must contain a single <layout> tag.
   * The <layout> Tag has two attributes -
     + **id** – It is recommended that every component will have a defined id.
     + **skinPath** – relative path to the skin file (compiled swf that contains style and skin definitions
2. There are two types of element tags which make up the application layout:
   * Containers
   * Plugins and Components

All layout elements, plugins, containers and components, can get their attributes and style through xml attributes. IE if a plugin has a property called ‘someProp’ it could be defined in the UiConf : **<Plugin id=’myPlugin’ someProp=’someValue’ …>**

1. Following are common attributes that all components (aka controllers) and modules hold:
   * **width** - in pixels (width=100) or percentages (width=100%)
   * **height** - in pixels (height=100) or percentages (height=100%)
   * **x** - in pixels. This attribute will be ignored in case of an HBox or VBox.
   * **y** - in pixels. This attribute will be ignored in case of an HBox or VBox.
   * **maxWidth** - in pixels, defines the maximum width that can be applied to the component.
   * **maxHeight** - in pixels, defines the maximum height that can be applied to the component.
   * **minWidth** - in pixels, defines the minimum width that can be applied to the component.
   * **minHeight** - in pixels, defines the minimum height that can be applied to the component.
   * **id** - unique identifier of the component, this can be any alpha-numeric string and must always start with a letter.
   * **hideInFullScreen** - Boolean(true/false), determine whether or not to hide this component on FullScreen mode.
   * **styleName** - the class name of the style that should be applied to the component, a matched class should be defined on the loaded compiled skin swf.
   * **visible** – Boolean(true/false)controls the item visibility.
   * **includeInLayout** - detach the item from its container for absolute positioning.
2. **Containers** - These elements define the various containers in the layout

**<Canvas> tag** - Provides a layer on which other element can be placed in an absolute positioning.

* + All container is suggested for creating layers that are stacked on top of each other (like the effect of z-index in HTML's CSS, however, in the KDP uiConf the z-index is defined solely by the element's place in the code - from top down - so that the top most element has the lowest 'z-index').
  + If you add the ‘styleName’ attribute, the canvas will look for a visual asset in the skin swf with that linkage name. This supports the flash 9 scaling feature.

**<HBox> tag** - All elements placed within it are stacked horizontally.

Common used properties:

* horizontalGap (number) – defines the horizontal spaces in pixels between children of the HBox.
* verticalAlign (bottom/ middle / top) – defines the vertical alignment of all the Hboxs children
* horizontalAlign (left/ right/ center) - defines the horizontal alignment of all the Hboxs children
* styleName – this will point to the class of the asset in the skin will be the background asset of the. This support the 9scale flash feature.
* Padding - number of pixels of extra spacing between the edge of the container:
  + paddingLeft
  + paddingRight
  + paddingTop
  + paddingBottom

Here’s a typical implementation of an HBox, the controller of a player, lay outing typical components:

<HBox id=“ControllerScreen”

width=“100%” height=“30”

horizontalGap=“6”

paddingleft=“12” paddingRight=“12”

verticalAlign=“middle”

styleName=“darkBg” >

<Button id=“playBtn” … />

<Scrubber id=“scrubber” width=“100%” … />

<Button id=“fullScreenBtn” … />

</HBox>

**<VBox> tag** - All elements placed within it are stacked vertically.

Common used properties:

* verticalGap (number) – defines the vertical spaces in pixels between children of the HBox.
* verticalAlign (bottom/ middle / top) – defines the vertical alignment of all the Hboxs children
* horizontalAlign (left/ right/ center) - defines the horizontal alignment of all the Hboxs children
* styleName – this will point to the class of the asset in the skin will be the background asset of the. This support the 9scale flash feature.
* Padding - number of pixels of extra spacing between the edge of the container:
  + paddingLeft
  + paddingRight
  + paddingTop
  + paddingBottom

here’s a VBox containing title, video and the controller HBox.

<VBox id=“ControllerScreen”

width=“100%” height=“100%”

styleName=“topLevelContainer” >

<Label id=“movieTitle” width=“100%” height=“30” … />

<Video id=“video” width=“100%” height=“100%” … />

<HBox id=“ControllerScreen”

width=“100%” height=“30” … >

<Button id=“playBtn” … />

<Scrubber id=“scrubber” width=“100%” />

<Button id=“fullScreenBtn” … />

</HBox>

</VBox>

This is a diagram that shows the output of this layout structure:

movieTitle



**<Tile> tag** - Elements placed within it are stacked horizontally until they reach the Tile width and then a new “row” begins.

* All “cells” will have the height of the tallest cell and width of the widest cell.
* The Tile container is suggested for displaying rows or elements which are all of the same size.
* Gap between “rows” and “columns” defaults and can be changed using the verticalGap and horizontalGap

Here’s a demo of a tile containing 4 buttons:

<Tile id=“buttonsHolder” width=“200” height=“200”>

<Button …/>

<Button …/>

<Button …/>

<Button …/>

</Tile>

Controllers

* + **The <Button> tag** - A generic button component. The following attributes are available:
    - command - used to attach a certain behavior to the button. The following actions are available:
      * play – Play/pause the video
      * fullScreen – Switch the KDP instance to full screen mode.
    - Label – Text that appears on button.
    - tooltip – Tooltip text that appears when is moused-over
    - toggle – if this attribute is set to true – it will behave as an 8 states button instead of 4 (up,over,down,disable, selectedUp, selectedOver, selectedDown, selectedDisable)
    - upTooltip - if the button has toggle behavior – this will be the tooltip in the non-selected state
    - selectedTooltip - if the button has toggle behavior – this will be the tooltip in the selected state
    - labelPlacement – the positioning of the icon relative to the label:
      * bottom
      * left
      * right
      * top
    - kClick – in case there is no command attached to the button it can get one of 3 other actions:
  + jsCall – call to an external JS function:
    - kClick=“jsCall('gotoEditorWindow', mediaProxy.entry.id)”
    - The example calls a JS function on the HTML page called “gotoEditorWindow” and will pass it the entryId taken from the mediaProxy.
    - The KDP can also inject JS code to the page. This is done by writing some JS code within the uiConf xml file. In this case the player will use eval to inject the page the function and will call it. This code will be written in a special XML element in the uiConf, and the functionName will actually be the name of the matching JSBlock.

**<Button** ... kClick=“jsCall('alertEntry')”**>**  
...  
 **<javaScript>**  
        **<jsBlock** id=“alertEntry” **>**  
                function alertEntry (entryId){  
                        alert(entryId);  
                }  
        **</jsBlock>**   
 **</javaScript>**

* + navigate – open external URL:
    - kClick=“navigate('http://www.kaltura.com')”
    - kClick=“navigate('http://www.kaltura.com','\_blank')”
  + sendNotification – send a general pureMVC notification so some other plugin / proxy can receive it.
    - kClick=“sendNotification('yourEvent')”
    - kClick=“sendNotification('yourEvent','arg1')”

note: you can place more than 1 action in a kClick. This kClick for example will stop the playing and seek to the beginning of it:

kClick=“sendNotification('doPause');sendNotification('doSeek',0)”

**Button skinning:**

Glossary

1. classes in skin files are symbols marked as “export for actionscript” and the class name is the text written in the linkage box.
2. Skin assets - the assets that build the background of the button. There are up to 8 states :
   1. up, over, down, disabled, selected up, selected over, selected down, selected disabled
   2. some buttons uses the selected states (play pause are actually the same button where selected shows the pause and not selected is play. Same thing for full screen button)
3. Icons assets – the icons for the button. As the skin – the icons can appear as states icons, or one icon for all states. a button can get icon and/or label
4. Label is the text field that appears inside the button (right/top/left/bottom according to the labelPlacement attribute).

There is more than one way to design a button in KDP3:

* Use the default style

In this case you do nothing in terms of styles attributes. A button without styleName or skin / styles attribute will get the default button skin as it is set in the skin.swf. This default skin can have all 8 states, and a label, and by definition – does not have icons.

* Applying a styleName

Setting a styleName to a button is as simples as that:

<Button … styleName=“myStyle” />

In this case the KDP tries to find matching classes assets with the same postfix in the skin swf file.

It will try to set these assets as skin for that button:

* Button\_upSkin\_myStyle
* Button\_downSkin\_myStyle
* Button\_overSkin\_myStyle
* Button\_disabledSkin\_myStyle
* Button\_selectedUpSkin\_myStyle
* Button\_selectedOverSkin\_myStyle
* Button\_selectedDownSkin\_myStyle
* Button\_selectedDisabledSkin\_myStyle

These classes should be on the skin swf as assets on stage. The symbols name is irrelevant, but as best practice we recommend them to have the same name. If there are no such assets, the button will get a fallback skin that should be at every skin swf with the same structure but with the “default” postfix:

* Button\_upSkin\_default
* Button\_downSkin\_default
* Button\_overSkin\_default
* Button\_disabledSkin\_default
* Button\_selectedUpSkin\_default
* Button\_selectedOverSkin\_default
* Button\_selectedDownSkin\_default
* Button\_selectedDisabledSkin\_default

It will try to set these assets as matching icons:

* Button\_upIcon\_myStyle
* Button\_overIcon\_myStyle
* Button\_downIcon\_myStyle
* Button\_disabledIcon\_myStyle
* Button\_selectedUpIcon\_myStyle
* Button\_selectedOverIcon\_myStyle
* Button\_selectedDownIcon\_myStyle
* Button\_selectedDisabledIcon\_myStyle

If there are no matching icons – there will not be any default icon. In this case there should be a label.

It will try to set text format copied from a movieclip containing a single text field. The KDP will copy all of its attributes (size, color, bold, italic, spacing, margins, justify etc’) and set it to the label of this button. Handling the font will be elaborated next. To make that – the movieclip that holds that textfield must have this class ( linkage )

In case you need to embed a unique font for the label of the button– see Label part.

Applying direct assets

The 2nd way to Overrides and reuse:

We can use the advantage of override to reuse assets without duplicate them on the skin. For example if we have 3 buttons with the same skin but with different icons we can use the styleName to set the skin, and override the icons by using different icons attributes in the ui xml:

<Button … styleName=“myStyle”

upIcon=“someIcon” overIcon=“someIcon” downIcon=“someIcon” disabledIcon=“someIcon”

selectedUpIcon=“someIcon” selectedOverIcon=“someIcon” selectedDownIcon=“someIcon” selectedDisabledIcon=“someIcon” />

Though this example shows one icon for all states we can set up different icons for different states.

Same thing goes for skin override

**Appstudio properties**

The Kaltura studio requires some specific behavior from a button. This behavior overrides the regular look&feel behavior that was just described and this section will describe the override.

Setting sizes of buttons.

The button of the KDP is actually an autosize button. This means that if the button has a label it will set its width automatically. This means that in some cases the width/height we want, and the actual width/height are not equal. For this, we have 4 attributes that might help us getting more accurate dimensions. The attributes are minWidth, maxWidth, minHeight, maxHeight. This buttons height for example will be 22 pixels height (or less) and will have 40 pix width even if its label is short or icon is small.

<Button … minWidth=“40” maxHeight=“22” />

**<Image> tag** - used to load an image or interactive SWF.

Attributes:

* url - the URL to the image/swf to load.

**<Label> tag** - Line of text. If the text s too long to fit the label it will be ellipsis and its tooltip will show the full value.

Attributes

text – The text to appear. The text attribute can get static string or get binded to some data. Here’s some examples:

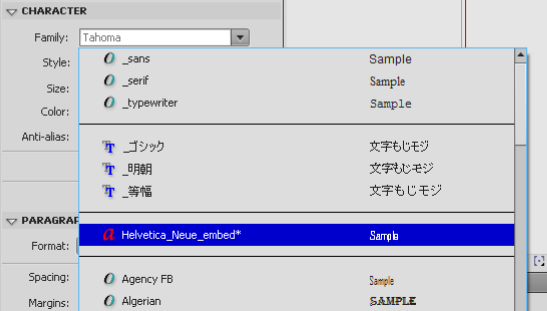
<Label id=”staticText” text=”Hello world” width=”120” styleName=”myLabel” />  
<Label id=”MediaName” text=”{mediaProxy.entry.name}” width=”100%” styleName=”myLabel” /> // this is a dynamic bind. Whenever a media is changed this label will be updated  
<Label id=”MediaNameWithPrefix” text=”Title:{mediaProxy.entry.name}” width=”100%” styleName=”myLabel” /> //the dynamic data will appear with a static prefix   
<Label id=”irMovieName” text=”{this.name}” width=”100%” styleName=”myLabel” /> //inside itemRenderer (one cell of a list) this will show the name of the entry. “this” refers to the data of the cell

**Skinning a label:**

The skinning of a label is straight forward. Place a single textfield in the skin swf file and design it (color, height alignment, padding spacig etc) convert it to a symbol and export it to actionscript class (Linkage). The name of the class is the name you write in the styleName attribute.

Font embedding

If you need a unique font embedding – you would need to do the following action in the skin swf file:

1. In the Library – add a new font. Chose the font you want. Make sure you chose both family and style if available. Remember the name you gave it.  
    
2. Place a textfield on the stage and set the font to the font you just created. Make sure you are setting up the embedded font from the library and NOT the original font source from your system font. It will usually be at the top of the list marked with an asterisk.   
   
   * In the code of the 1st frame, where the security code is written – add these 2 lines:

import flash.text.Font;

Font.registerFont(Helvetica\_Neue\_embed); //the name you gave the font without any quotes. Add this line for every font you embed.

* + **<Scrubber> tag** - A slider used to seek (navigate) the played video. – no special attributes
  + **<Spacer> tag** - A whitespace component used to reserve empty spaces.

Common attributes:

* + - width (pixels or percentage)
    - height (pixels or percentage)
  + **<watermark>** is a component that loads an external image and places it in one of the 4 corners of Its area. Usually the image is a transparent png. The image can be clickable and will redirect to a given url. Please note that this is a rectangle that places the image in one of its corners. The most common use for the location if the watermark component is over the video area, meaning that there is a canvas holding the video and the watermark with 100% percentage for their width & height

<Watermark id="watermark" width="100%" height="100%"

watermarkPath="http://www.kaltura.com/.../exampleWatermark.png"

watermarkClickPath="http://www.kaltura.com/"

watermarkPosition="bottomLeft" padding="5"/>

Common attributes:

* + - position:
      * topLeft (Default)
      * topRight
      * bottomRight
      * bottomLeft
    - watermarkClickPath – a URL for an image or swf file
    - watermarkClickPath – click watermark landing page url
    - padding – number of pixels from the border of the component to the loaded image
  + **The <Plugin> tag** -
    - A plugin is a compiled swf that provides additional functionality.
    - A <Plugin> can be one of 2 types:
      * A Kaltura official plugin. This type of plugin is being released with an official KDP release and its swf is placed on a specific location. In this case the id attribute of the plugin is the identifier of the plugin.
      * An external plugin – that is not part of the official KDP bundle. In this case the plugin must get a ‘path’ attribute that points to the path of the swf.
    - It is strongly recommended to follow the naming conventions of every plugin, IE ‘**doSomethingPlugin.swf**’. for more information see ‘how to create a kdp3 plugin’
  + **The <Screens> tag** - The screens element is a container that switches between different states according to the player state (start, play, pause, end) and their mouse-over state.

The Screens component has 2 modes:

* + - simple mode – switch between 4 states:
      * Start (when the player is loaded) and start-over (When the mouse is over the start screen)
      * Playing+Over. When the player is playing a media and the mouse is over the player
      * Pause + pause over (When the player is paused)
      * End movie + over (When the media is done). In playlist, if the playlist is in autoPlay mode this will actually be on the last entry.
    - Advanced mode – using all states, normal and mouse over
      * Start
      * Start+over
      * Playing
      * Playing Over
      * Pause
      * Pause Over
      * End screen
      * End Screen Over

The screens tags has pointers to xmls in a special xml node in the UiConf called <screens> (Lower case). This element sits outside the main container of the layout. The <screens> tags can contains one or more <screen> tag. These tags are actually small pieces of regular ui xml as the regular layout of the player. Each screen must have a unique “id” attribute so it could be referenced. The <Screens> component will refer these id’s.

The <screens> has a special attribute called mouseOverTarget – this is a pointer to another ui xml element that will be the hotspot for the mouse-over behavior. Through this property you would be able to set up for example if in playlist the hotspot will be the whole player or just the video area.

This is a demo of a typical implementation of <screens>. Note that the <screens> component in this demo uses the same screen for the pause state, and the start state (They both have a big play button). Playing+hover state has a share button, and end state + end state over have a replay button, and basically are using the same screen. The playing state (playing without over) is showing nothing so it will not hide the video area.

…

<Canvas id="PlayerHolder" height="100%" width="100%" styleName="black">

<Video id="video" width="100%" height="100%"/>

<Screens id="screensLayer" width="100%" height="100%"

mouseOverTarget="{PlayerHolder}"

startScreenOverId="startScreen" startScreenId="startScreen"

playScreenOverId="playScreenHover"

pauseScreenId="startScreen" pauseScreenOverId="startScreen"

endScreenId="endScreen" endScreenOverId="endScreen"/>

</Canvas>

The screens section in the UiConf XML would look like this:

<screens>

<!-- This screen will serve both start state and pause state -->

<screen id="startScreen">

<VBox id="startStateContainerStart" width="100%" height="100%" verticalAlign="middle" horizontalAlign="center">

<Button id="onVideoPlayBtnStartScreen" command="play"

minWidth="60" labelPlacement="top" label="Play"

styleName="onScreenBtn" icon="playIcon" />

</VBox>

</screen>

<!-- This screen will be the play+hover state -->

<screen id="playScreenHover">

<VBox id="playStateContainer" width="100%" height="100%" verticalAlign="middle" horizontalAlign="center">

<Button id="onVideoShareBtnPlayingScreen" kClick=”sendNotification('doGigya')”

minWidth="60" labelPlacement="top" label="Share"

styleName="onScreenBtn" icon="shareIcon" />

</VBox>

</screen>

<!-- This screen will be the end state -->

<screen id="endScreen">

<VBox id="endStateContainer" width="100%" height="100%" verticalAlign="middle" horizontalAlign="center">

<Button id="onVideoReplayBtnEndScreen" kClick=”sendNotification('play')”

minWidth="60" labelPlacement="top" label="Replay"

styleName="onScreenBtn" icon="replayIcon" />

</VBox>

</screen>

</screens>

**<Gigya>**

Gigya is a 3rd party plugin that allows to viral a KDP through embed code, email, popular bookmarks services or popular social networks. The plugin is a visual plugin that disable the player when active and shows the Gigya ui. Usually it is located over the video area. In playlist context players it could also appear over the whole player (video,controllers and playlist area).   
The Gigya plugin needs a trigger to initiate – usually by a button that sends the notification ‘doGigya’. This button could e anywhere, and other plugins could send this notification too.

Here’s a typical Gigya component, usually placed over the video area (in the same canvas that holds that player).

<Canvas … >

<Video id="video" width="100%" height="100%"/>

<Plugin id="gigya" width="100%" height="100%"

shareEmailBody="Hi,&lt;br&gt;I watched this video and thought you'd enjoy it too. &lt;br&gt;$URL$ to watch.&lt;br> $sender$"

shareEmailSubject="Take a look at this video"/>

…

<Canvas>

Properties:

* shareEmailBody – the body of the email that will be sent via Gigya. This should be encoded, this is why you see ,&lt;br&gt; instead of <br>.   
  There are 2 dynamic variables that you can use in the body of the email,
  + $URL$ - this will be replaced with a ‘click here’ text and will link to a page on Gigyas servers containing this player
  + $sender$ - the name of the sender.
* shareEmailSubject – the subject of the email.

Heres a typical button that triggers Gigya.

<Button id="shareBtnControllerScreen" kClick="sendNotification('doGigya')"

styleName="controllerBtn" icon="shareIcon" />

The Gigya inner configuration is an XML file generated in Gigya site and could be positioned within the KDP3 uiConf in a special XML node. This controls the look and feel of the gigya widget (border,buttons &strokes colors, texts, icons etc`) and also functional configuration (which social network to show, show hide tabs etc`). If you configure your own Gigya widget you must remember to mark the option of ‘close button” so the end user will be able to close it and get back to player mode.

<extraData>

<GigyaUI>

Your custom Gigya XML here – starts with <config> tag

</GigyaUI>

</extraData>

**<Timer>** tag

The timer is the component that shows the info (as numbers) of the current playing media

Common attributes:

* timerType– show time left / current position / total :
  + total – shows the duration of the current media
  + backwards – shows the time left from current point to the end of the media
  + forwards – shows the current duration of the media
  + both – toggles between backwards and forwards (user click changes the behavior)
* format – the format of the number:
  + mm:ss – show only minutes and seconds
  + hh:mm:ss – shows hours, minutes and seconds

Since the Timer extends the label – applying skin is the same as label – see styling a label in the label section.

**<VolumeBar>**

The volume bar is actually 2 components – volume and mute button. It should be located on the control bar. The button is a toggle mute/un mute and when the mouse is hovering over it, a volume bar appears, so the user can set the exact volume value.

The button of the volume bar is a regular KDP button so designing it is the same as designing a regular KDP button, only that the prefixes of the volume bar skins is fixed and the skin file already contains the assets.

1. **Using Dynamic Data** -
   * Bindings are a method of attaching dynamic data to attribute values.
   * This can be as visible text such as in a **<Label>** used to define a title
   * If the source of the bound data is changed, the changed data is reflected where you used the binding.
   * Limitations:
     + Bindings can only refer to a property of the KDP object. You cannot use logic or mathematical operators.
   * In order to use Bindings, the value part of the attribute's name=value pair must be wrapped in curly brackets. Examples:
     + **<Image** url=“{object.property}” **/>**
     + **<Label** text=“{object.object.property}” **/>**
     + **<Button** label=“ prefix: {object.object.property}” **/>**
2. **Open attribute/style policy** -
   * If a plugin/component/container gets any attribute, and it has a setter for this attribute or a public property, or a valid style attribute, it will get the value of it when it loads, before its init function (or initializePlugin function in case of plugin).

**<Module** someAttribute=“4”... **/>**

**Playlist**

1. **Playlist & Item renderer**
   * A Kdp can be configured as a playlist player. In this case the UiConf will hold 2 plugins:
     + playlistAPIPlugin – a logic plugin that loads playlist(s) parse and manage its queue.  
       <Plugin id=“playlistAPI” width=“0%” height=“0%” includeInLayout=“false”/>
     + List – a vertical list that shows the entries. The list plugin has an attribute to an item renderer – an XML that describes the inner structure of an item. The list is binded to the playlistAPI for its data  
          
       <Plugin id=“list” width=“100%” height=“100%”   
        styleName=“List\_background\_default”   
        dataProvider=“{playlistAPI.dataProvider}”   
        itemRenderer=“playlistItemRenderer” rowHeight=“70”/>
     + The UiConf XML holds a special XML call renderers that should hold a renderer XML node with a matching id. Here’s a typical one:

<renderers>

<renderer id=“playlistItemRenderer”>  
 <HBox height=“100%” width=“100%” x=“10” y=“10” verticalAlign=“top” paddingTop=“6”>  
 <VBox height=“100%” width=“100%” id=“playlistItemRendererCanvas” >  
 <HBox id=“nameAndDuration” width=“100%” >  
 <Image id=“playlistItemImage” height=“48” width=“72” url=“{this.thumbnailUrl}”/>  
 <Label id=“testlabel2” height=“18” width=“100%” text=“{this.name}” styleName=“itemRendererLabel” />  
 <Label id=“testlabel3” height=“18” width=“60” text=“{formatDate(this.duration, 'NN:SS')}” styleName=“itemRendererLabel” />  
 </HBox>  
 </VBox>  
 </HBox>  
 </renderer>  
 </renderers>

1. A special feature of this screen XML is the styleName. For elaboration of assigning styles assets to a controller – see skinning KDP document.