



## OSMF Release Samples

### Walkthrough 8: Integrating Simple Plug-ins

#### Overview:

This walkthrough illustrates how to dynamically load at runtime static (class-based) plug-ins that are compiled into the application. The plug-in is the Akamai Basic Streaming Plug-in, which is meant to help facilitate connections and streaming off the Akamai network. The sample will dynamically reference and instantiate the plug-in class via the AS3 `getDefinitionByName()` method which retrieves a class reference for instantiation by a string of the class path. Aside from simply loading the plug-in, this walkthrough will also show the syntax and use of the plug-in status event listeners so that as a developer you may react to success or failure of plug-in loading.

#### Objectives:

- Enable loading of plug-ins dynamic through a centralized method ( `loadPlugin()` )
- Load in a Static class based plug-in
- Handle the successful and failed loading of the plug-in

#### Setup

1. Open the file `WT08_SimplePlugin.as` in the `{SAMPLES_PROJECT}/src` directory.

NOTE: This file has been provided as a starting point for these walkthroughs.

2. Set the class file as the application file to compile. There are two different ways of doing this depending on which program you are building your application in.

##### **Flash Builder**

Right-click the `WT08_SimplePlugin.as` file and select `Set as Default Application` from the context menu that appears. This will add the project to the list of compilable applications. A blue dot on the file icon indicates that the file is the default application file.

##### **Flash Professional**

Open the `OSMF_SampleTemplate.fla` and save it as `WT08_SimplePlugin.fla`. Then change the document class for the file (in the Properties panel) to `WT08_SimplePlugin`.

#### Loading the plug-in

3. Find the comment which looks like the following: *//NOTE: Class force-loaded to compile into the application.* Note the class reference below the comment. By

putting that reference there it forces the plug-in class to be compiled into the application.

4. Under the "//Marker 1:" comment add 2 event listeners to the mediaFactory.
5. One for the MediaFactoryEvent.PLUGIN\_LOAD that is handled by the onPluginLoaded method
6. Another for the MediaFactoryEvent.PLUGIN\_LOAD\_ERROR that is handled by the onPluginLoadFailed

```
//Marker 1: Add the listeners for the plugin load call  
mediaFactory.addEventListener( MediaFactoryEvent.PLUGIN_LOAD,  
onPluginLoaded );  
mediaFactory.addEventListener( MediaFactoryEvent.PLUGIN_LOAD_ERROR,  
onPluginLoadFailed );
```

**NOTE:** Having a success handler to notify the application when the plug-in has loaded successfully can be imperative to the sequencing of a media player. Often times the plug-in will need to change or add something to the content that should be played - such as a pre-roll advertisement. If the content is already playing before the plug-in has been loaded successfully, then often the chance to interject or modify has been missed.

6. In the initPlayer() method locate the comment that begins "//Marker 2:".
7. Call the loadPlugin() method, passing it the static string BASIC\_STREAMING\_PLUGIN as the only parameter.

```
//Marker 2: Load the plugin  
loadPlugin( BASIC_STREAMING_PLUGIN );
```

8. Locate the loadPlugin() method.
9. Find the comment that begins "//Marker 3:".
10. Since we are using a static plug-in (Class based, not a SWF) we will create a Class type variable named pluginInfoClass and set it equal to the result of calling the getDefinitionByName() method, passing it the source parameter.

```
//Marker 3: Create the plugin resource using a static plugin  
var pluginInfoClass:Class = getDefinitionByName( source ) as Class;
```

**NOTE:** The use of String variables with the full class path, and dynamically retrieving a reference to the class via the getDefinitionByName() method is completely optional, and has nothing to do with OSMF specifically. The reason it was done this way for the sample is to illustrate a more realistic real-world scenario in which the plug-ins to be loaded may not be a fixed set, and could be determined by FlashVar's, or an XML configuration file. This way, all that is needed is the class to be loaded, and String variables that reference them.

11. Create a `MediaResourceBase`-typed variable named `pluginResource` and set it equal to a new `PluginInfoResource` object. Pass a new `pluginInfoClass()` object as the only parameter to the constructor.

```
//Marker 3: Create the plugin resource using a static plugin
var pluginInfoClass:Class = getDefinitionByName( source ) as Class;
var pluginResource:MediaResourceBase = new PluginInfoResource( new
pluginInfoClass() );
```

12. Under the `//Marker 4:` comment, call the `loadPlugin()` method on the `mediaFactory` object, passing it the `pluginResource` variable.

```
//Marker 4: Load the plugin
mediaFactory.loadPlugin( pluginResource );
```

## Loading the media after the plug-in has loaded

13. In the `onPluginLoaded` event handler method, under the `//Marker 5:` comment, call the `loadMedia()` method.

```
protected function onPluginLoaded( event:MediaFactoryEvent ):void
{
    trace( "onPluginLoaded()" );

    //Marker 5: Load the media

    loadMedia();
}
```

14. In the `loadMedia()` method, under the `//Marker 6:` comment, note that the path to the media to load points to the `STREAM_AUTH` variable, which is a URL that contains the authentication key for an Akamai stream. The URL in the `STREAM_AUTH` is a sample of what an Akamai stream with authentication parameters might look like. The plug-in can help streamline authentication through data such as this. Unfortunately, Akamai authentication tokens expire, so this URL may not work. For testing purposes, change the `STREAM_AUTH` in the `URLResource` to `PROGRESSIVE_SSL`.
15. Save and run the application. The video should load successfully.

Ουκ καλόν είναι τον άνθρωπον μόνον...

It's not good to be alone...