# Open Source Media Framework – Plug-in Developer's Guide

## 1. Overview

Open Source Media Framework (OSMF) is an ActionScript framework for building media players and media player plug-ins. OSMF is a collection of free, open-source components that simplify media player creation for the Adobe Flash Platform. The framework, documentation, and latest updates can be found on the OSMF web site: http://www.osmf.org.

This document explains what OSMF plug-ins are, how they work, and how to build a plug-in.

### *Why Plug-ins?*

Let's start with some of the problems that OSMF is attempting to address. Today's media player does much more than just play media. In many cases, it connects to a Content Delivery Network (CDN), presents advertising, captures user events to report to an analytics server, presents social features in the user interface, and so on. Typically, these additional functions are handled by third parties who have deep expertise in their particular domain, expertise that the player developer may lack. In an ideal world, the player developer would be able to leverage the efforts of these third parties, without having to have a deep understanding of how their implementation works, and would have the ability to switch from one third party to another with minimal change to their player.

This is the fundamental problem that OSMF's plug-in architecture attempts to solve. In OSMF, the player developer builds their player, the third party builds their plug-in, and the framework acts as the broker between the two, handling integration through a standardized set of APIs. The advantage for the player developer is that they can integrate a new plug-in, or switch from one plug-in to another, without having to change the implementation of their player. The advantage for the plug-in developer is that they can build their plug-in once, and be confident that it will work with any OSMF-based player.

### *What is a Plug-in?*

An OSMF plug-in is nothing more than code that hooks into the framework to provide additional or customized functionality. A media player is responsible for loading a plug-in. A plug-in can be declared (and therefore loaded) either as a **static plug-in** or as a **dynamic plug-in**. A static plug-in is incorporated into the media player at compile time. A dynamic plug-in is loaded by the media player at runtime from a SWF file. Static plug-ins provide faster startup time, whereas dynamic plug-ins allow for centralized hosting.

Plug-ins can encapsulate visual or non-visual media. A visual plug-in could encapsulate a SWF overlay which pauses the main video when the user interacts with the SWF. A non-visual plug-in could encapsulate code that tracks how the user interacts with the video, and reports that data to an analytics server.

An OSMF plug-in is *declarative*. This simply means that a plug-in *declares* its capabilities (through an OSMF API), and the framework is responsible for introspecting the plug-in to determine its capabilities, and then establishing relationships between the media player and the plug-in based on those capabilities. Plug-ins are *not* given free rein to access any object within the media player. Rather, OSMF acts as the broker between the media player and the plug-in. This approach ensures that communication between the media player and the plug-in is standardized, eliminating the need for custom integration work when adding or switching plug-ins.

The MediaFactory class is the class which handles this arbitration between player and plug-in. The player developer uses a MediaFactory to load plug-ins and create MediaElements. Although it's possible to directly instantiate a MediaElement, using the MediaFactory to create a MediaElement allows loaded plug-ins to affect the outcome.

## 2. Using Plug-ins in a Media Player

This section of the document explains how to use plug-ins from the perspective of the media player developer.

### *Loading a Plug-in*

A player developer can load a plug-in into their media player by using the **MediaFactory** class. To load a plug-in, the player developer would do the following:

1. Create an instance of MediaFactory (or DefaultMediaFactory).
2. Add event listeners for the "pluginLoad" and "pluginLoadError" events.
3. Create a new resource representing the plug-in:
    a. For static plug-ins, create an instance of PluginInfoResource, passing it an instance of the PluginInfo that the plug-in declares.
    b. For dynamic plug-ins, create an instance of URLResource, passing it the URL of the dynamic plug-in's SWF file.
4. Call MediaFactory.loadPlugin, passing in the resource from step #3.
5. If the event listener for "pluginLoad" is invoked, then the plug-in has been successfully loaded.
6. If the event listener for "pluginLoadError" is invoked, then the plug-in has failed to load.

Here is some sample code illustrating how to load a dynamic plug-in:

```
public class HelloWorldPluginApp extends Sprite
{
        public function HelloWorldPluginApp()
        {
                // Create the DefaultMediaFactory and add listeners
                // for the loading of the plug-in.
                var factory:MediaFactory = new DefaultMediaFactory();
                factory.addEventListener(MediaFactoryEvent.PLUGIN_LOAD, onPluginLoad);
                factory.addEventListener(MediaFactoryEvent.PLUGIN_LOAD_ERROR, onPluginLoadError);

                // Create a resource for the plug-in SWF.
                var resource:URLResource = new URLResource("http://example.com/samplePlugin.swf");

                // Load the plug-in SWF.
                factory.loadPlugin(resource);
        }
```

```
        private function onPluginLoad(event:MediaFactoryEvent):void
        {
                trace("The plug-in loaded successfully.");
        }

        private function onPluginLoadError(event:MediaFactoryEvent):void
        {
                trace("The plug-in failed to load.");
        }
}
```

*Note: A plug-in may fail to load due to inconsistency between where the media player is run and where the plug-in SWF resides. In short, the plug-in and the player must either both be run from the network sandbox, or both be run from a local sandbox.*

### *Using a Plug-in*

A media player never uses a plug-in directly. Instead, the media player will create a MediaElement using MediaFactory.createMediaElement, and interact with that MediaElement. If the resource that is passed to the createMediaElement method is one that the plug-in recognizes and knows how to handle, then the plug-in can choose a different MediaElement to return, or modify the behavior of the MediaElement to return. However, the player developer doesn't need to know the details of whether the plug-in can handle the resource, and what the plug-in does if it can. The player developer simply calls the createMediaElement method and works with the returned MediaElement.

Here is some sample code extending the above example to show how to use a loaded plug-in:

```
public class HelloWorldPluginApp extends Sprite
{
        public function HelloWorldPluginApp()
        {
                // Create the DefaultMediaFactory and add listeners
                // for the loading of the plug-in.
                var factory:MediaFactory = new DefaultMediaFactory();
                factory.addEventListener(MediaFactoryEvent.PLUGIN_LOAD, onPluginLoad);
                factory.addEventListener(MediaFactoryEvent.PLUGIN_LOAD_ERROR, onPluginLoadError);

                // Create a resource for the plug-in SWF.
                var resource:URLResource = new URLResource("http://example.com/samplePlugin.swf");

                // Load the plug-in SWF.
                factory.loadPlugin(resource);
        }

        private function onPluginLoad(event:MediaFactoryEvent):void
        {
                trace("The plug-in loaded successfully.");

                // Create a resource for the video to display.
                var resource:URLResource = new URLResource("http://example.com/video.flv");

                // Generate a MediaElement from the MediaFactory, using the video resource.
                // By using the MediaFactory, the plug-in gets a chance to intercept the
                // request and influence the result.
                var media:MediaElement = factory.createMediaElement(resource);

                // Create a MediaPlayer to display the media.
                var mediaPlayer:MediaPlayer = new MediaPlayer();

                // Assign the generated MediaElement to the MediaPlayer.
                mediaPlayer.media = media;
        }
```

```
        private function onPluginLoadError(event:MediaFactoryEvent):void
        {
                trace("The plug-in failed to load.");
        }
}
```

### *Plug-in Collisions*

In some cases, a player application might have loaded multiple plug-ins, each of which can handle a given resource. In such cases, the framework may not know which plug-in should "win" (i.e. which plug-in should be used to create the MediaElement.) For example, if the player application loads multiple CDN plug-ins, and then calls MediaFactory.createMediaElement with a resource that represents a video URL, how does the MediaFactory know which CDN plug-in to use? OSMF provides two different mechanisms for resolving these situations: plug-in metadata and plug-in resolution.

#### Plug-in Metadata

Plug-in metadata is resource-level metadata that further qualifies what the resource represents, so that plug-ins can be more precise in indicating whether they can handle a resource. For example, suppose a player application loads two plug-ins, one for CDN Foo and another for CDN Bar. It then calls MediaFactory.createMediaElement as follows:

```
var resource:URLResource = new URLResource("rtmp://example.com/mystream");
var mediaElement:MediaElement = mediaFactory.createMediaElement(resource);
```

The MediaFactory will ask the plug-in for CDN Foo whether it can handle the resource "rtmp://example.com/mystream", and the answer will be "yes". The MediaFactory will then ask the plug-in for CDN Bar whether it can handle the same resource, and the answer will be "yes". So the MediaFactory is in a situation where it has to choose from multiple options, with no way to know which choice is the right one.

Suppose, however, that the player developer knows that the resource should be handled by CDN Bar because it originates from their network. They could then specify some additional metadata on the resource to indicate this:

```
var resource:URLResource = new URLResource("rtmp://example.com/mystream");
resource.addMetadataValue("http://example.com/publisherId", "Bar");
var mediaElement:MediaElement = mediaFactory.createMediaElement(resource);
```

The new line of code adds resource-level metadata to the video URL, using the namespace "http://example.com/publisherId" with value "com.bar". If the plug-ins for both CDN Foo and CDN Bar are modified to look for this metadata value and only answer "yes" if the value matches their company name, then the MediaFactory will know exactly which plug-in to use, and generate the right MediaElement.

> *Note: The above approach relies on coordination between related plug-ins to standardize on metadata namespaces. Such standardization efforts are still in their infancy.*

## Plug-in Resolution

Continuing with the previous example: suppose that plug-in metadata is insufficient to resolve the collision between multiple plug-ins, for example if neither CDN Foo nor CDN Bar checked for resource-level metadata. MediaFactory.createMediaElement ultimately will return one MediaElement. How does it decide which one to create and return (i.e. which plug-in to choose)?

The answer is that the MediaFactory will use plug-in resolution, via the protected MediaFactory.resolveItems method. This protected method takes a Vector of MediaFactoryItem objects, and returns the one that should be used to create the MediaElement. By default, this method will do the following:

1. Choose the first MediaFactoryItem it finds whose id property does not contain the string "org.osmf".
2. If there is no such MediaFactoryItem, then it will choose the first MediaFactoryItem available. (The reason behind the comparison with "org.osmf" is to give precedence to plug-ins over native media implementations.)

   *Tip: When creating a plug-in, make sure the ID does not contain the string "org.osmf" or it might be given a lower priority and therefore not be invoked.*

If the default implementation of MediaFactory.resolveItems is insufficient for a specific scenario, then the player developer can subclass MediaFactory (or DefaultMediaFactory) and provide an alternate implementation.

### *Plug-in Load Failures*

A dynamic plug-in can fail to load for a number of reasons:

1. If the URL of the dynamic plug-in is invalid, then the load will fail.
2. If the player application is in a network sandbox and the plug-in is local, then the load will fail. Similarly, if the player application is in a local sandbox and the plug-in is in a network sandbox, then the load will fail. (Note that to load a plug-in locally, the plug-in URL should include the "file://" prefix.)
3. If the version of OSMF that the plug-in was compiled against is incompatible with the OSMF version that the player application is using, then the load will fail. (See the "Plug-in Versioning section of this document for more details.)
4. If a plug-in does not have a "pluginInfo" property of type PluginInfo (or if any of the methods on its PluginInfo generate errors), then the load will fail.

The best way to diagnose plug-in load failures is to enable framework logging. (To enable logging, set the CONFIG::LOGGING compilation flag to true for the OSMF project.) The log messages generated by OSMF during the loading of a plug-in will often shed light on the reason for a load failure.

## 3. Building a Plug-in

This section of the document explains how to build and deploy a plug-in from the perspective of the plug-in developer.

### *Building a Basic Plug-in*

The starting point for a plug-in, whether it should be static or dynamic, is the **PluginInfo** class. A PluginInfo object encapsulates all of the information about a plug-in. In order for a media player application to load a plug-in, it must be able to access the PluginInfo object for that plug-in.

At its most basic level, a PluginInfo object exposes one or more "descriptors" of how to create a MediaElement. These descriptors are represented by the **MediaFactoryItem** class. A MediaFactoryItem exposes properties that the MediaFactory can use to determine whether the MediaFactoryItem can handle a given resource, and to create the MediaElement if it can handle that resource.

Here's a plug-in that simply generates a new VideoElement if the resource starts with the string "rtmp":

```
// Root object which exposes the PluginInfo object.
public class HelloWorldPlugin extends Sprite
{
	public function HelloWorldPlugin()
	{
		super();

		_pluginInfo = new HelloWorldPluginInfo();
	}

	// OSMF will invoke this getter to retrieve the plug-in's
	// PluginInfo object.
	public function get pluginInfo():PluginInfo
	{
		return _pluginInfo;
	}

	private var _pluginInfo:PluginInfo;
}

// The PluginInfo object.
public class HelloWorldPluginInfo extends PluginInfo
{
	public function HelloWorldPluginInfo()
	{
		var items:Vector.<MediaFactoryItem> = new Vector.<MediaFactoryItem>();

		// Create the MediaFactoryItem and add to our list of items.
		var item:MediaFactoryItem = new MediaFactoryItem
			( "com.example.helloworld"
			, canHandleResourceFunction
			, mediaElementCreationFunction
			);
		items.push(item);

		// Pass the list to the base class.
		super(items);
	}

	// OSMF will invoke this function for any resource that is passed
	// to MediaFactory.createMediaElement.  The method must take a single
	// parameter of type MediaResourceBase and return a Boolean.  The
	// plug-in should return true if it can create a MediaElement for that
	// resource.
	private function canHandleResourceFunction(resource:MediaResourceBase):Boolean
	{
```

```
                var result:Boolean = false;

                // Only return true if the resource is an URLResource...
                var urlResource:URLResource = resource as URLResource;
                if (urlResource != null)
                {
                        // ... and if the URL starts with "rtmp".
                        result = urlResource.url.indexOf("rtmp") == 0;
                }

                return result;
        }

        // OSMF will invoke this function for any resource that is passed to
        // MediaFactory.createMediaElement, and for which the corresponding
        // canHandleResourceFunction returns true.  The method must take
        // no parameters and return a MediaElement.
        private function mediaElementCreationFunction():MediaElement
        {
                return new VideoElement();
        }
}
```

In the above example, the workflow between player and plug-in is as follows:

1. The player application loads the plug-in by calling MediaFactory.loadPlugin.
2. When the plug-in is successfully loaded, the player application receives a “pluginLoad” event from the MediaFactory.
3. The player application creates an URLResource and set the url property to “rtmp://example.com/mystream”.
4. The player application invokes MediaFactory.createMediaElement, passing in that URLResource.
5. For each MediaFactoryItem that it knows about, the MediaFactory calls the canHandleResourceFunction. When HelloWorldPluginInfo’s canHandleResourceFunction is called, it returns true because the URLResource’s url property matches the condition (i.e. starts with “rtmp”).
6. Because the canHandleResourceFunction returns true, the MediaFactory invokes HelloWorldPluginInfo’s mediaElementCreationFunction to generate a VideoElement.
7. The MediaFactory assigns the URLResource to the generated VideoElement, and returns the result to the player application.
8. The player application assigns the generated VideoElement to its MediaPlayer, and begins playback.

Notice that the player application is shielded from having to know whether a plug-in is involved in the creation of the returned MediaElement. The player application simply requests a MediaElement for a given resource, and works with the result.

Here’s a summary of the actors and responsibilities in this workflow:

- **Player Application**
  - Responsible for loading plug-ins.
  - Asks the MediaFactory to generate a MediaElement for a given resource.

- **Plug-in**
    - Responsible for indicating whether it can "handle" a resource.
    - Responsible for generating a MediaElement when it can handle a resource.
- **MediaFactory (OSMF)**
    - Acts as the broker between player application and plug-in.

Now that we have a functional plug-in, the next question is how best to deploy it. As described earlier, there are two ways to deploy/load plug-ins: statically and dynamically.

### *Implementing Static Plug-ins*

Static plug-ins are compiled directly into the player application. Typically, the plug-in developer will provide either source code or a library (SWC) for the plug-in. The player developer can link the plug-in into their application and use the APIs directly. Here is how to do in Flex Builder:

- If the plug-in is provided in the form of source code, then go to the project properties, select ActionScript Build Path (or Flex Build Path, for Flex applications), select the Source path tab, select Add Folder, and enter the path to the source.
- If the plug-in is provided in the form of a library (SWC), then go to the project properties, select ActionScript Build Path (or Flex Build Path, for Flex applications), select the Library path tab, select Add SWC, and enter the path to the SWC.

The player application will then be ready to load the plug-in, which can be achieved through the following line of code:

```
mediaFactory.loadPlugin(new PluginInfoResource(new HelloWorldPluginInfo()));
```

### *Implementing Dynamic Plug-ins*

Dynamic plug-ins are loaded at runtime into the player application. The player developer only needs to know the URL where the plug-in SWF resides. Thus, the plug-in developer needs to generate a SWF for the plug-in, place it on a web server, and inform the player developer of the URL to the SWF.

Let's say that the plug-in is compiled as HelloWorldPlugin.swf, and placed at the URL "http://example.com/HelloWorldPlugin.swf". The player developer could load the plug-in with the following line of code:

```
mediaFactory.loadPlugin(new URLResource("http://example.com/HelloWorldPlugin.swf"));
```

### *Initializing Your Plug-in*

Initialization logic for a plug-in should be placed in the PluginInfo.initializePlugin method. The initializePlugin method will be invoked once the plug-in has been successfully loaded. Initialization logic should not be placed in the PluginInfo's constructor, as that method may be invoked multiple times, or be invoked even if the plug-in fails to load.

The initializePlugin method takes a MediaResourceBase as a parameter. This is the exact same resource that the player provides to the MediaFactory.loadPlugin method. If your plug-in needs information (such as account information) from the player application in order to be initialized correctly, then the

player developer should assign that information as metadata values to the resource that they pass to the loadPlugin method, so that your initializePlugin method can retrieve and process it. If your plug-in requires any such initialization information, it's important to document the names and types of resource metadata that it expects so that users of your plug-in know what information to provide.

## *Types of Plug-ins*

There are several different types of plug-ins supported by OSMF.

### Standard Plug-ins

A **standard plug-in** (also referred to as a MediaElement plug-in) is a plug-in whose sole purpose is to create and return a MediaElement that will be used directly in the player application. The HelloWorldPlugin from the previous sections is a standard plug-in. Standard plug-ins are primarily concerned with returning the correct value for their canHandleResourceFunction so that they provide a MediaElement in response to the appropriate resource.

Standard plug-ins have a MediaFactoryItem whose MediaFactoryItemType is STANDARD (the default).

There are two specific sub-types of standard plug-ins: custom MediaElement plug-ins and built-in MediaElement plug-ins.

A **custom MediaElement plug-in** returns a MediaElement subclass that is not part of OSMF. For example, a custom MediaElement plug-in might return a RecommendationsSWFElement that loads a custom SWF to show UI for recommended videos. Custom MediaElement plug-ins are typically used to provide additional functionality to a player application.

A **built-in MediaElement plug-in** returns a MediaElement subclass that is part of OSMF. For example, a build-in MediaElement plug-in might return a VideoElement that uses a custom NetLoader to do token-based authentication for a particular CDN. Built-in MediaElement plug-ins are typically used to alter the manner in which the main media implementations (video, audio, images, and SWFs) behave.

### Proxy Plug-ins

A **proxy plug-in** is a plug-in that returns a ProxyElement that will be used to proxy another created MediaElement. Proxy plug-ins allow the plug-in developer to non-invasively alter the behavior of created MediaElements. For example, suppose a plug-in wanted to disable seeking for any VideoElement generated by the player application (e.g. to prevent ad skipping). A proxy plug-in could achieve this without requiring any changes to the player application itself.

A proxy plug-in will generate a ProxyElement that is inserted in front of the MediaElement that the MediaFactory would normally return. Because ProxyElement and MediaElement share the same interface, the player application can interact with the ProxyElement without being aware that it's a ProxyElement. Any method or property that the player application invokes will return the value from the proxied MediaElement, unless the ProxyElement itself intercepts that request to modify the behavior. Because the ProxyElement wraps the other MediaElement, but can change the

MediaElement's behavior, we say that a proxy plug-in can *non-invasively* alter the behavior of a created MediaElement.

When a MediaFactory has loaded a proxy plug-in, the workflow of the MediaFactory.createMediaElement method changes, as follows:

1. The MediaFactory will first determine whether a standard plug-in can "handle" the resource.
2. If such a standard plug-in exists, then the MediaFactory will obtain a MediaElement from the MediaFactoryItem's mediaElementCreationFunction.
3. Instead of returning the standard MediaElement, the MediaFactory will instead iterate over all proxy plug-ins, asking each one if it can "handle" the same resource. For each proxy plug-in that can handle the resource, the MediaFactory will obtain a ProxyElement from the MediaFactoryItem's mediaElementCreationFunction, and assign the previously created MediaElement as the proxiedElement. It will then return the ProxyElement to the caller of the createMediaElement function.
4. If no standard plug-in exists in step #2, then the MediaFactory will return null, and the proxy plug-in won't be used.

There are a few key points to be aware of when building a proxy plug-in:

1. The MediaFactoryItem must specify a MediaFactoryItemType of PROXY.
2. The MediaFactoryItem's mediaElement method must return a ProxyElement or subclass of ProxyElement. The created ProxyElement should not have its proxiedElement set, the MediaFactory will assign a value to the proxiedElement property.
3. The MediaFactoryItem's canHandleResourceFunction should return true for any resource whose corresponding MediaElement should be proxied by the ProxyElement.

An example should help clarify these points. Suppose you wanted to build a plug-in that prevents seeking for all progressive videos. The following PluginInfo class does just that:

```
// The PluginInfo object for the proxy plug-in.
public class ProxyPluginInfo extends PluginInfo
{
	public function ProxyPluginInfo()
	{
		var items:Vector.<MediaFactoryItem> = new Vector.<MediaFactoryItem>();

		// Create the MediaFactoryItem and add to our list of items.
		var item:MediaFactoryItem = new MediaFactoryItem
			( "com.example.proxyplugin"
			, canHandleResourceFunction
			, mediaElementCreationFunction
			, MediaFactoryItemType.PROXY
			);
		items.push(item);

		// Pass the list to the base class.
		super(items);
	}

	// OSMF will invoke this function for any resource that:
	// a) is passed to MediaFactory.createMediaElement, and
	// b) for which a standard MediaFactoryItem's canHandleResourceFunction
```

```
        //    also returns true.
        // If this method returns true, then the ProxyElement generated
        // by this plug-in will wrap the MediaElement generated by the
        // MediaFactoryItem in step B, above.
        private function canHandleResourceFunction(resource:MediaResourceBase):Boolean
        {
                var result:Boolean = false;

                // Only return true if the resource is an URLResource...
                var urlResource:URLResource = resource as URLResource;
                if (urlResource != null)
                {
                        // ... and if the URL starts with "http".
                        result = urlResource.url.indexOf("http") == 0;
                }

                return result;
        }

        // The ProxyElement generated by this function will wrap, or
        // proxy, another MediaElement generated by the MediaFactory.
        // This method should *not* assign a value to the proxiedElement,
        // the MediaFactory in the player application will do that.
        private function mediaElementCreationFunction():MediaElement
        {
                // Source code for UnseekableProxyElement is available at:
                // http://opensource.adobe.com/svn/opensource/osmf/trunk/apps/samples/
                //   framework/ExamplePlayer/org/osmf/examples/seeking/UnseekableProxyElement.as

                return new UnseekableProxyElement(null);
        }
}
```

Suppose that the player application loads this plug-in, and then attempts to create a MediaElement, as follows:

```
mediaPlayer.media = factory.createMediaElement(new URLResource("http://example.com/video.flv"));
```

Here's how the workflow would unfold, based on the four steps described previously:

1. The MediaFactory finds a MediaFactoryItem that can handle video.
2. It then calls the MediaFactoryItem's mediaElementCreationFunction, which returns a VideoElement. In the absence of the proxy plug-in, this is the MediaElement that would be returned.
3. The MediaFactory then iterates over the proxy plug-ins, invoking the canHandleResourceFunction for each one. The canHandleResourceFunction for ProxyPluginInfo's MediaFactoryItem returns true because the resource's URL starts with "http". As a result, the MediaFactory calls the MediaFactoryItem's mediaElementCreationFunction, which returns the UnseekableProxyElement. Last, the MediaFactory assigns the VideoElement to the UnseekableProxyElement's proxiedElement property, and returns the UnseekableProxyElement to the caller.

### Reference Plug-ins

A **reference plug-in** is a plug-in whose operation depends upon gaining a *reference* to one or more MediaElements created by the MediaFactory. In most cases, the reference plug-in will also generate MediaElements of its own. For example, a reference plug-in might encapsulate a SWF overlay which needs to gain reference to the main VideoElement so that it can pause the video when the user clicks on

the SWF overlay. Alternatively, a reference plug-in might encapsulate some client-side tracking logic which listens to events from the main VideoElement, and report them to an analytics server.

To create a reference plug-in, all you need to do is provide an implementation for the PluginInfo's mediaElementCreationNotificationFunction. The mediaElementCreationNotificationFunction will be invoked for every MediaElement created by the MediaFactory. Here's a modified version of HelloWorldPluginInfo, which generates a trace statement for each created MediaElement:

```
// The PluginInfo object.
public class HelloWorldPluginInfo extends PluginInfo
{
	public function HelloWorldPluginInfo()
	{
		var items:Vector.<MediaFactoryItem> = new Vector.<MediaFactoryItem>();

		// Create the MediaFactoryItem and add to our list of items.
		var item:MediaFactoryItem = new MediaFactoryItem
			( "com.example.helloworld"
			, canHandleResourceFunction
			, mediaElementCreationFunction
			);
		items.push(item);

		// Pass the list to the base class.
		super(items, creationNotificationFunction);
	}

	// OSMF will invoke this function for any resource that is passed
	// to MediaFactory.createMediaElement.  The method must take a single
	// parameter of type MediaResourceBase and return a Boolean.  The
	// plug-in should return true if it can create a MediaElement for that
	// resource.
	private function canHandleResourceFunction(resource:MediaResourceBase):Boolean
	{
		var result:Boolean = false;

		// Only return true if the resource is an URLResource...
		var urlResource:URLResource = resource as URLResource;
		if (urlResource != null)
		{
			// ... and if the URL starts with "rtmp".
			result = urlResource.url.indexOf("rtmp") == 0;
		}

		return result;
	}

	// OSMF will invoke this function for any resource that is passed to
	// MediaFactory.createMediaElement, and for which the corresponding
	// canHandleResourceFunction returns true.  The method must take
	// no parameters and return a MediaElement.
	private function mediaElementCreationFunction():MediaElement
	{
		return new VideoElement();
	}

	// OSMF will invoke this function for any MediaElement returned by
	// MediaFactory.createMediaElement.  Note that this function will
	// even be invoked for MediaElements created prior to the loading
	// of this plug-in, so as to isolate the plug-in from load order
	// dependencies.
	private function creationNotificationFunction(media:MediaElement):void
	{
		trace("New element created");
	}
}
```

Note that the PluginInfo's creationNotificationFunction will be invoked for *all* MediaElements created by the MediaFactory, even if a MediaElement was created prior to the loading of the plug-in. The reason for this behavior is to ensure that the player application doesn't exhibit different behavior whether they load the plug-in before or after the creation of the MediaElement.

### Proxy vs. Reference Plug-ins

The following rules of thumb should help clarify when to create a proxy plug-in and when to create a reference plug-in:

1. Create a proxy plug-in when you want to adjust the behavior of another MediaElement.
    - *Example: A plug-in that prevents the pausing and seeking of any video.*
2. Create a reference plug-in when you want to adjust the behavior of another MediaElement, but also provide a media implementation of your own.
    - *Example: A plug-in that presents an overlay SWF which can pause the video when it's clicked.*
3. Create either a proxy plug-in or a reference plug-in when you want to monitor, but not change, the behavior of another MediaElement.
    - *Example: A plug-in that sends tracking events to an analytics server.*

### *Plug-in Versioning*

When a plug-in is loaded, OSMF performs a few checks to verify that the version of OSMF used by the plug-in is compatible with the version of OSMF used by the player. The default rules are as follows:

1. A player can load a plug-in that was built against an older version of OSMF, down to the earliest compatible version (more on that below).
2. A player cannot load a plug-in that was built against a newer version of OSMF.

For example, if a player was built against OSMF version 1.5, it can load plug-ins built against version 1.0 or version 1.5, but it cannot load plug-ins built against version 2.0.

The earliest compatible version is version 1.0, until and unless we make significant enough changes to OSMF that version 1.0 plug-ins will no longer work in the latest version. (Changes to the earliest compatible version number will be infrequent.)

## 4. Additional Resources

The following resources may be useful for learning more about OSMF plug-ins:

OSMF 1.0 ASDocs – Reference for all OSMF APIs

OSMF Developer's Guide – Contains additional information on plug-ins, plus general usage information

OSMF Plug-ins – Link to source repository containing the OSMF sample plug-ins, including the SMIL, MAST, and Captioning plug-ins.

AkamaiPluginSample sample app (source) (demo) – Demonstrates how to load and use a variety of plug-ins.

ControlBarPluginSample sample app (source) (demo) – Demonstrates how to create a visual plug-in. In this example, the video player's control bar is loaded as a plug-in.