**Silverlight Media Framework v2 - Plugin Developer Guide**

**Introduction**

One of the new features introduced in SMF v2 is the ability to extend the player’s “out of the box” functionality with plugins. Plugins in SMF can perform a variety of specialized tasks like logging and media playback as well as add completely new behavior, like social media integration. In addition to loading plugins that are deployed with an SMF application, the player also has the ability to load external XAP files containing plugins over the network. This allows plugin developers to host and easily share plugins since all an application developer needs is the plugin’s URL.

Each plugin in SMF has a corresponding interface type and Export Attribute that must be applied to the plugin class. All plugin interfaces extend from IPlugin, which defines the minimum set of operations that any SMF plugin must implement, like Load() and Unload(). The actual plugin interfaces define the specific operations that are required for the specialized functionality they are meant to perform, like media playback for example. Finally, the Export Attributes that are associated with each plugin perform 2 tasks: 1) They enable MEF to recognize the class as a specific plugin implementation and 2) Provide a mechanism for specifying metadata about the plugin’s capabilities. At runtime SMF uses MEF to compile a list of all plugins that are currently available. When a plugin of a specific type is needed, the player will query each plugin’s metadata and select the one that most closely matches its requirements.

**Plugin Types**

**Plugin Basics**

IPlugin is the base interface for all SMF plugins and defines the minimum set of operations that any SMF plugin must implement. Likewise, the ExportPluginAttribute is the base class of all SMF export attributes and defines metadata members that all plugins can use to describe themselves.

**IPlugin Interface:**

* Members:
  + IsLoaded – return a boolean value indicating whether the plugin has been loaded.
* Methods:
  + Load() – The plugin should load any resources that are required for operation. This method is expected to be asynchronous and the plugin implementation must indicate that this operation has completed by firing the PluginLoaded event.
  + Unload() – The plugin should unload unnecessary resources and stop operation. This method is expected to be asynchronous and the plugin implementation must indicate that this operation has completed by firing the PluginUnloaded event.
* Events:
  + LogReady – Indicates that logging information is available.
  + PluginLoaded – Indicates that the Load() operation has completed.
  + PluginLoadFailed – Indicates that the Load() operation has failed.
  + PluginUnloaded – Indicates that the Unload() operation has completed.
  + PluginUnloadFailed – Indicates that the Unload() operation has failed.

**ExportPluginAttribute Members:**

* PluginName – The official name of this plugin.
* PluginDescription – A description of what the plugin does.
* PluginVersion – The version of this plugin.

**Log Writer**

Log Writer plugins record logging information that is generated by the player and other SMF plugins. These plugins are required to implement the ILogWriter interface and must be decorated with the ExportLogWriterAttribute.

**ILogWriter Interface:**

* Methods:
  + WriteLog(LogEntry) – The plugin should record the specified logging information contained in the LogEntry.
* Events:
  + LogWriteFailed – Indicates that recording of the logging information has failed.
  + LogWriteSuccessful – Indicates that recording of the logging information has completed successfully.

**ExportLogWriterAttribute Members:**

* LogWriterId – The unique id of this LogWriter. SMF will use this metadata value to load the correct Log Writer when specified on SMFPlayer.LogWriters.

**Plugin Lifetime:**

Log Writer plugins are instantiated and loaded at the following times:

* When the SMFPlayer initializes in the OnApplyTemplate() method.
* When the SMFPlayer.LogWriters property value changes.
* When an external .XAP has been downloaded.

The SMFPlayer does not explicitly unload Log Writer plugins, however they are free to unload themselves at any time.

**Example Usage:**

[ExportLogWriter(PluginName = "MyLogWriter",

PluginDescription = "This is an example of a Log Writer plugin.",

PluginVersion = "1.0",

LogWriterId = "MyUniqueLogWriterid")]

public class MyLogWriter : ILogWriter { … }

**UI Plugin**

UI Plugins are able to add UI Elements to the Player’s Visual Tree. These plugins are required to implement the IUIPlugin interface and must be decorated with the ExportUIPluginAttribute.

**IUIPlugin Interface:**

* Members:
  + Target – This is a string that indicates the location, within the Visual Tree, where the UI Element should be added. This string must match the name of some Panel within the player’s existing UI. SMF will use this value to locate the correct Panel and then add the UI Element to its children. If the value of Target is null or empty the UI Element will be added to PlayerRoot.
  + Element – This is the element that should be added to the player’s Visual Tree.

**ExportUIPluginAttribute has no members.**

**Plugin Lifetime:**

UI plugins are instantiated and loaded at the following times:

* When the SMFPlayer initializes in the OnApplyTemplate() method.
* When an external .XAP has been downloaded.

The SMFPlayer does not explicitly unload UI plugins, however they are free to unload themselves at any time. Unloading a UI plugin will cause it’s associated UIElement to be removed from the Visual Tree.

**Example Usage:**

[ExportUIPlugin(PluginName = "MyUIPlugin",

PluginDescription = "This is an example of a UI plugin.",

PluginVersion = "1.0")]

public class MyUIPlugin : IUIPlugin { … }

**Marker Provider**

Marker Providers are able to load Timeline Markers, Chapters, and Captions from an arbitrary data source and make them available within the player. These plugins are required to implement the IMarkerProvider interface and must be decorated with the ExportMarkerProviderAttribute.

**IMarkerProvider Interface:**

* Members:
  + Source – A URI indicating the location of the marker data that should be loaded by the plugin. This will value will be provided to the plugin by the player before Load() is called.
  + PollingInterval – Indicates the frequency with which data should be refreshed from the Source. If this value is null the Marker Provider should not poll.
* Methods:
  + BeginRetrievingMarkers() – The Marker Provider should start retrieving markers. If PollingInterval is non-null it should continue polling until StopRetrievingMarkers() is called.
  + StopRetrievingMarkers() – Indicates that the Marker Provider should stop retrieving markers and polling should be cancelled.
* Events:
  + NewMarkers – Indicates that new markers are available.
  + MarkersRemoved – Indicates that markers should be removed.
  + RetrieveMarkersFailed – Indicates that the retrieval of markers has failed.

**ExportsMarkerProviderAttribute Members:**

* SupportsPolling – Indicates whether this Marker Provider implements supporting polling a data source.
* SupportedFormat – Indicates the data format that is supported by this Marker Provider.

**Plugin Lifetime:**

The lifetime of a Marker Provider is tied to a specific Playlist Item. If a Playlist Item specifies a MarkerResource a Marker Provider will be instantiated and loaded in the *OnMediaPluginLoaded* method, which occurs after the Media Plugin has been loaded. Immediately after the MarkerProvider has completed loading, indicated by firing it’s *PluginLoaded* event, *BeginRetrievingMarkers()* will be called. When the current playlist item changes, *StopRetrievingMarkers()* will be called, immediately followed by *Unload()*.

**Example Usage:**

[ExportMarkerProvider(PluginName = "MyMarkerProvider",

PluginDescription = "This is an example of a Marker Provider plugin.",

PluginVersion = "1.0",

SupportedFormat = "TTAF1-DFXP",

SupportsPolling = true)]

public class MyMarkerProvider : IMarkerProvider { … }

**Generic Plugin**

Generic Plugins are a sort of open ended plugin that allows developers to extend functionality in ways that are not supported by any of the other plugins types. These plugins are given a reference to the player allowing them to perform many types of tasks. Creating a Generic Plugin requires implementing the IGenericPlugin interface and decorating the plugin class with the ExportGenericPluginAttribute.

**IGenericPlugin Interface:**

* Methods:
  + SetPlayer – This method is called by the player passing in a reference to itself.

**ExportGenericPluginAttribute has no members.**

**Plugin Lifetime:**

Generic plugins are instantiated and loaded at the following times:

* When the SMFPlayer initializes in the OnApplyTemplate() method.
* When an external .XAP has been downloaded.

The SMFPlayer does not explicitly unload Generic plugins, however they are free to unload themselves at any time.

**Example Usage:**

[ExportMarkerProvider(PluginName = "MyGenericPlugin",

PluginDescription = "This is an example of a Generic Plugin.",

PluginVersion = "1.0")]

public class MyGenericPlugin : IGenericPlugin { … }

**Media Plugins**

Media Plugins allow developers to write components that take over the media playback experience. The Media Plugins that are included with SMF wrap the Silverlight MediaElement and the SmoothStreamingMediaElement and leverage their playback functionality to support various types of media “out of the box”. By implementing a custom Media Plugin, it is possible to extend the functionality of an existing media playback component or even create an entirely new playback experience by supporting new types of media and formats. Implementation of a Media Plugin is a non-trivial and fairly complex effort. Instead of diving into the details of the interfaces we will limit the scope here to a high level discussion of each type of Media Plugin.

**IMediaPlugin** – This is the base interface from which all other Media Plugin interfaces extend. It exposes members that enable basic playback functionality and advertising support. Implementation of this interface is sufficient and required for a plugin to function as a Media Plugin in SMF.

**ILiveDvrMediaPlugin** – This interface is specialized for playback experiences that require Live “DVR” functionality. Implementations must have the ability to seek to past positions within a live stream and provide the player with updated information about the live position of the media. This interface is optional and not required for a plugin to provide basic playback functionality.

**IAdaptiveMediaPlugin** – This interface is specialized for experiences that are delivered using “Adaptive” media, an example of which is Microsoft’s [Smooth Streaming](http://www.iis.net/download/SmoothStreaming) technology. An Adaptive Media Plugin is required to make available its list of streams and tracks by implementing additional interfaces like ISegment, IMediaStream, and others. In addition, implementations are expected to provide the player with information about the current state of selected streams and bitrates. For a complete example, inspect the Microsoft.SilverlightMediaFramework.Plugin.SmoothStreaming project which is included in the SMF source code release. This interface is optional and not required for a plugin to provide basic playback functionality.

**Plugin Lifetime:** The lifetime of a Media Plugin is only as long as a single media. Each time a new Playlist Item is selected, any existing Media Plugin is unloaded, releasing its resources, and another one is selected. It is important to note that this does not mean that plugin instances will be destroyed or garbage collected. Media Plugin instances should be developed to be reusable so that calling Load() and Unload() effectively creates and destroys any underlying resources.

After a Media Plugin is selected, the player will determine whether it implements any optional interfaces (i.e. ILiveDvrMediaPlugin or IAdaptiveMediaPlugin) dynamically. Based on the plugin’s capabilities different services may be made available and the user interface may enter different visual states indicating that the experience supports certain capabilities. An example of this would be the selection box for audio streams. This UI feature is automatically enabled if the active Media Plugin implements IAdaptiveMediaPlugin and indicates that there is more than 1 available audio stream for the current media.

**ExportMediaPluginAttribute Members:**

* SupportsLiveDvr – Indicates that the plugin implements ILiveDvrMediaPlugin
* SupportedDeliveryMethods – Indicates the Delivery Methods that are supported by this plugin.

**Examples:**

[ExportMediaPlugin(PluginName = PluginName,  
        PluginDescription = PluginDescription,  
        PluginVersion = PluginVersion,  
        SupportedDeliveryMethods = SupportedDeliveryMethodsInternal)]  
    [PartCreationPolicy(CreationPolicy.NonShared)]  
    public class ProgressiveMediaPlugin : IMediaPlugin, IDisposable  
    {  
        private const string PluginName = "ProgressiveMediaPlugin";  
   
        private const string PluginDescription =  
            "Provides Progressive Download capabilities for the Silverlight Media Framework by wrapping the MediaElement.";  
   
        private const string PluginVersion = "2.2010.0811.1";  
   
        private const DeliveryMethods SupportedDeliveryMethodsInternal =  
            DeliveryMethods.ProgressiveDownload | DeliveryMethods.Streaming;

…

}

[ExportMediaPlugin(PluginName = PluginName,  
        PluginDescription = PluginDescription,  
        PluginVersion = PluginVersion,  
        SupportsLiveDvr = true,  
        SupportedDeliveryMethods = SupportedDeliveryMethodsInternal)]  
    [PartCreationPolicy(CreationPolicy.NonShared)]  
    public class SmoothStreamingMediaPlugin : IAdaptiveMediaPlugin, ILiveDvrMediaPlugin, IDisposable  
    {  
        private const string PluginName = "SmoothStreamingMediaPlugin";  
   
        private const string PluginDescription =  
            "Provides Smooth Streaming capabilities for the Silverlight Media Framework by wrapping the SmoothStreamingMediaElement.";  
   
        private const string PluginVersion = "2.2010.0811.1";  
   
        private const DeliveryMethods SupportedDeliveryMethodsInternal =  
            DeliveryMethods.ProgressiveDownload |   
            DeliveryMethods.Streaming | DeliveryMethods.AdaptiveStreaming;

…

}

**Plugin API**



