OPEN SOURCE MEDIA FRAMEWORK
Developer's Guide

Open Source Media Framework Developer's Guide

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Last updated 9/7/2011

Contents

Chapter 1: OSMF Overview

Open Source Media Framework (OSMF) is a pure ActionScript® 3.0 framework that gives developers complete flexibility and control in creating their own rich media experiences. OSMF is a free collection of open-source components that simplify media player creation for the Adobe® Flash® Platform.

The OSMF production-ready code base saves you time and effort, so you have more time for the features with the greatest impact for your customers. OSMF also has an open and extensible architecture that allows you to integrate third-party services quickly and easily.

The framework, documentation, and latest updates are on the OSMF website: www.osmf.org. For further support, see www.adobe.com/support/osmf.

Features

OSMF 1.0 provides these features:

- Support for both standard and advanced delivery methods, including progressive download, RTMP and HTTP streaming, RTMP and HTTP Dynamic Streaming, and live streaming with RTMP and HTTP.
- Automatic content protection with Flash® Access™ 2.0.
- Simple integration of third-party plug-in services such as CDNs, advertising, and analytics. The flexible architecture supplies the option of compiling plug-ins statically or loading them dynamically, so plug-in providers can perform immediate upgrades and versioning.
- Quality of service (QoS) features, including dynamic (multi-bitrate) streaming.
- Support for all Flash® Player 10.0 media formats, including:
  - Streaming video (FLV, F4V, MP4, MPEG-4, MP4, M4V, F4F, 3GPP)
  - Progressive audio (mp3)
  - Progressive video (FLV, F4V, MP4, MP4V-ES, M4V, 3GPP, 3GPP2, QuickTime)
  - Images (PNG, GIF, JPG)
  - SWF files
  - Limited support for streaming audio (mp3, AAC, Speex, Nellymoser)

OSMF 1.5 adds support for:

- RTMFP Multicast
- Stream Reconnect

OSMF 1.6 enables:

- "Playback with the Stage Video pipeline" on page 8
- "Supporting alternate audio tracks in HTTP Streaming videos" on page 22, or "late-binding" of audio
- "Live DVR and Rolling Window for HTTP dynamic streams" on page 19

Development environment

To use the basic features of OSMF, plug-in and player developers must have a minimum of Flash Player 10.0 or later installed as part of their development environment. End users must also have a minimum of Flash Player 10.0 or later installed.

Flash Player 10.1 is required for full access to all OSMF 1.5 features, including:

- HTTP Streaming (also requires Flash® Media Server 3.8 or greater)
- Stream Reconnect (also requires Flash Media Server 3.5.3 or greater)
- RTMFP multicast (also requires Flash Media Server 4.0 or greater)

Flash Player 10.2 is required for Stage Video hardware acceleration.

OSMF is implemented using ActionScript 3.0, without any Flex® APIs or Flash authoring components, so it's not dependent on a particular development platform.

To use OSMF in the development environment of your choice:

- Download the OSMF source zip file.
- Extract the OSMF.swc file from the source zip.
- Add it to your library folder or path. (The details vary depending on which development environment and platform you are using.)

Specific installation instructions for Adobe development environments are in the release notes that are included within the OSMF download package.

Chapter 2: Basic Playback

Open Source Media Framework (OSMF) provides basic building blocks that you can quickly use to create your media player and begin playback:

- "Media elements and resources" on page 3 and "Media factories" on page 5 present the initial concepts and steps for how your player handles content with OSMF.
- "Media players and media containers" on page 6 describes how to use the OSMF classes that play and display content.
- "Media player sprites and a simpler player" on page 7 presents the simplest method of starting basic playback.
- OSMF 1.6 automatically supports "Playback with the Stage Video pipeline" on page 8, greatly improving the efficiency of rendering for your player.

Media elements and resources

Any type of content that an OSMF media player plays is considered to be a type of media element. Examples of media elements include videos, sound files, images, or SWF content. A media element can also be a grouping of media that is played as if it is a single media file.

`MediaElement` is the base class that represents every category of media in the framework. All media types included in the framework extend `MediaElement`. Custom media types introduced by developers must also extend `MediaElement`. Using `MediaElement`, OSMF players can be notified of and respond to events within the display list hierarchy for Flash Player applications.

To construct a media element, you supply a media resource, specifying the content to play. Typically, you use a URL or file path as the defining data for a media resource. However, a media resource can also be something more complex, such as an array of streams or a plug-in application. `URLResource` is the type most commonly used to instantiate media elements.

Media element types

OSMF defines many types of standard media elements. Among these, the core elements represent basic media types, including:

- `AudioElement`. This type supports both streaming and progressive delivery formats and can play mp3 or AAC files. It handles mp3 files over HTTP, as well as audio-only streams from Flash Media Server.
- `ImageElement`. You can use this type to load and present PNG, GIF, or JPG images.
- `VideoElement` and `LightweightVideoElement`. `LightweightVideoElement` supports progressive and simple RTMP streaming only. It is the parent class for `VideoElement`, which supports all OSMF video delivery modes.
- `SWFElement`. Use this element to play SWF (Flash movie) files.

Each of the core media element types is a child of the `LoadableElementBase` class, a direct subclass of `MediaElement`. Including `LoadableElementBase`, the direct subclasses of `MediaElement` are:

- `LoadableElementBase`, which provides base functionality for media elements that require loading before they are played. It is not typically instantiated directly but is the base class for the core media element types.

- `CompositeElement` enables functional groupings of core elements. You do not typically create instances of `CompositeElement` directly, but instead use its subclasses, `ParallelElement` and `SerialElement`. See "Creating a composite element" on page 4 for more information and a code sample.
- `HTMLElement` represents a piece of media external to the Flash SWF file that is within an HTML region. It serves as a bridge to external JavaScript APIs and is discussed in "External HTML control with JavaScript" on page 34.
- `ProxyElement` is a media element that wraps another media element for the purpose of controlling access to the wrapped element. `ProxyElement` and its subclass `DurationElement` are discussed in "Proxy elements" on page 24.
- `BeaconElement` is a media element that contains the URL to retrieve when a play request (HTTP GET) is received. It is useful for media analytics/reporting.

Creating a composite element

A composite element describes a single media experience made up of individual parts. Here are a few examples of media compositions:

- 10-minute video stream + midroll ad + additional 10-minute video stream + postroll ad
- 20-minute video stream + companion banner SWF ads shown next to the video
- Two videos at the same time, as in picture-in-picture or multiple camera angles

`CompositeElement` defines this type of media element, but typically you do not create instances of this type directly. Instead, you create media compositions based on its two subclasses:

- `ParallelElement` is a media composition whose elements are presented in parallel (concurrently).
- `SerialElement` is a media composition whose elements are presented serially (in sequence).

However many individual elements make up a media composition, the overall composition operates as a single, unified media element. For example, if `SerialElement` encapsulates a series of videos, the `SerialElement` object behaves as if it's a single video element, but one that plays several videos in sequence. Or, if a `ParallelElement` object encapsulates an image and a piece of audio, the object behaves as if it's a single media element with the audio characteristics of the audio file and the display characteristics of the image.

Note: *If files of different types are being loaded in a ParallelElement, it is possible that a file of one type (for example, an image) may complete loading before other members of the element (such as a progressively downloaded video).*

Because a `CompositeElement` object maintains a list of `MediaElement` children, any of which can be `CompositeElement` objects themselves, a media composition can be expressed as a tree structure.

Note: *Media compositions that include 4 or more parallel elements can perform poorly in some browser environments.*

The following code excerpt shows the basic construction of a composite element.

```
private function createMediaElement():MediaElement
{
    var serialElement:SerialElement = new SerialElement();
    // First child is a progressive video.
    serialElement.addChild
        ( new VideoElement
            ( new URLResource(http://myserver.com/assets/testprogressive.mov)
            )
        );
    // Second child is a Flash movie. DurationElement causes it to play for 3 seconds.
    serialElement.addChild
        ( new DurationElement
            ( 3, new SWFElement
                ( new URLResource(http://myserver.com/assets/testSWF.swf)
                )
            )
        );
    // Third child is a streaming video.
    serialElement.addChild
        ( new VideoElement
            ( new URLResource(http://myserver.com/assets/teststream.mp4)
            )
        );
    return serialElement;
}
```

Media factories

The simplest way to turn a media resource into an element of the appropriate type is to use a media factory. The `DefaultMediaFactory` class takes a media resource as input and translates it into the correct type of `MediaElement`. `DefaultMediaFactory` implements support for most common `MediaElement` subclasses, such as audio, video, image, SWF, and F4M elements.

It is typically best to instantiate a media element indirectly with `DefaultMediaFactory`, rather than doing so directly. By doing so, `DefaultMediaFactory` cannot only create appropriate media elements from URLs, it can also perform convenience services for the content. For example, `DefaultMediaFactory` communicates with proxy plug-ins, so that they know about the element being created and can perform their services for it.

The following code fragment demonstrates how media resources, elements, and factories work together:

```
// Create a new default media factory.
var myFactory:MediaFactory = new DefaultMediaFactory();
// Create the resource to play.
var myResource:URLResource = new URLResource("http://myserver.com/content/test.flv");
// Create a media element using the factory. Specifically, the factory recognizes
// the .flv extension of the resource and creates a VideoElement.
var myElement:MediaElement = myFactory.createMediaElement(myResource);
```

"Media elements and resources" on page 3 described media resources as the building blocks of media elements. While `DefaultMediaFactory` can typically translate a resource to an element automatically, there are situations where it can't. For example, your player may need to load a file from a URL that has no file extension, such as one served by a PHP script. In this case, you must explicitly provide the file type to OSMF; `DefaultMediaFactory` cannot do it for you.

You inform OSMF of the file type by setting the `mediaType` or `mimeType` properties of the media resource. (See `osmf.media.MediaType` for the current list of supported types and their string values.) The following code shows an example of one way to handle this situation:

```
// Create a new image element.
var element:ImageElement = new ImageElement();
// Create a new resource from the URL to a jpg file.
var resource:URLResource = new URLResource("http://myserver.com/content/sample");

// Assign the proper media type to the resource.
resource.mediaType = MediaType.IMAGE;

// Once the resource has its media type, assign the resource to the element.
element.resource = resource;
```

Media players and media containers

The `MediaPlayer` class is the controller class for playing any type of media element. This class roughly corresponds to an actual media player and encapsulates the low-level tasks required to play media. `MediaPlayer` methods and properties map to user interface control actions, such as play, pause, seek, and stop. `MediaPlayer` does not contain any display or "view" functionality.

To play a media element, it must be assigned to a `MediaPlayer` object. Once you do so, `MediaPlayer` automatically starts monitoring the assigned element and tracks when its capabilities become available. The `MediaPlayer` class both automatically loads the assigned media and plays the media once loading completes.

Still, loading and playing are not enough. If you use `MediaPlayer` and `MediaElement`, you can make a simple player application, but no one can see what is playing. From the perspective of the Model-View-Controller software design pattern, the `MediaPlayer` class provides a controller, and the `MediaElement` class is the model. The `MediaContainer` class provides the display or view for your program.

`MediaContainer` is a display object container, descending from the `Sprite` class. `MediaContainer` implements much of the code to display both the content and the interface for your player application. Once you use `MediaContainer` with `MediaPlayer` and `MediaElement`, you have a functional, visible media player.

Note: *For more information on display object containers and the display list in the Flash Platform, see* `flash.display.DisplayObjectContainer`.

The following sample shows the construction of a simple OSMF media player. This player starts playback automatically, without user interaction. To create an interactive user interface for your player, see Handling User Interaction. To create an even simpler auto-playback media player, see "Media player sprites and a simpler player" on page 7.

```
package
{
    import flash.display.Sprite;
    import org.osmf.containers.MediaContainer;
    import org.osmf.media.DefaultMediaFactory;
    import org.osmf.media.MediaElement;
    import org.osmf.media.MediaFactory;
    import org.osmf.media.MediaPlayer;
    import org.osmf.media.URLResource;

    [SWF(width="640", height="352")]
    public class HelloWorld extends Sprite
    {
        public function HelloWorld()
        {
            // Create the container class that displays the media.
            var container:MediaContainer = new MediaContainer();
            addChild(container);

            //Create a new DefaultMediaFactory
            var mediaFactory:MediaFactory = new DefaultMediaFactory();

            // Create the resource to play.
            var resource:URLResource = new URLResource("http://myserver.com/test.flv");

            // Create MediaElement using MediaFactory and add it to the container class.
            var mediaElement:MediaElement = mediaFactory.createMediaElement(resource);
            container.addMediaElement(mediaElement);

            // Add MediaElement to a MediaPlayer. Because the MediaPlayer
            // autoPlay property defaults to true, playback begins immediately.
            var mediaPlayer:MediaPlayer = new MediaPlayer();
            mediaPlayer.media = mediaElement;
        }
    }
}
```

Media player sprites and a simpler player

So far we've created a basic media player using `MediaFactory`, `MediaPlayer`, and `MediaContainer`. `MediaPlayerSprite` is an application-level class that combines these classes into one convenience class. It uses `DefaultMediaFactory` to generate the media element and load the content, `MediaContainer` to display the content, and `MediaPlayer` to control the content. (It also allows you to provide a custom media player, container, or factory as parameters, if you want.)

Unlike the `MediaPlayer` class, `MediaPlayerSprite` inherits from `Sprite`, thus allowing an all-in-one component for a player. Creating a media player with `MediaPlayerSprite` is extremely simple and quick, as shown in the sample following.

```
package
{
    import flash.display.Sprite;
    import org.osmf.media.MediaPlayerSprite;
    import org.osmf.media.URLResource;

    [SWF(width="640", height="352")]
    public class HelloWorld extends Sprite
    {
        public function HelloWorld()
        {
            // Create the container class that displays the media.
            var sprite:MediaPlayerSprite = new MediaPlayerSprite();
            addChild(sprite);

            // Assign the resource to play. This generates the appropriate
            // MediaElement and passes it to the MediaPlayer. Because the MediaPlayer
            // autoPlay property defaults to true, playback begins immediately.
            sprite.resource = new URLResource("http://myserver.net/content/test.flv");
        }
    }
}
```

Controlling content scale with MediaPlayerSprite

For visual content, determining how it is sized and scaled is often integral to a successful viewing experience. Because this functionality can be so important, OSMF provides access to scale modes within `MediaPlayerSprite`.

To control the content scale mode, set `MediaPlayerSprite.scaleMode` with one of the following constants:

LETTERBOX	Sets the content width and height as close as possible to that of the container while maintaining the content's original aspect ratio. The player background may appear along the vertical or horizontal edges of the content, if its LETTERBOX size does not exactly match the current dimensions of the container. This is the default scale mode.
NONE	Allows no scaling of content. The content is displayed at its intrinsic size.
STRETCH	Sets the content width and height to that of the container, possibly changing the content's original aspect ratio.
ZOOM	Sets the content to maintain its original aspect ratio, while filling the entire visible area of the container. No background is allowed to appear with the ZOOM setting, so portions of the content may extend beyond the visible bounds of the container and the visible content may appear horizontally or vertically clipped as a result.

Playback with the Stage Video pipeline

OSMF 1.6 provides automatic support for the Stage Video pipeline, supported by Flash Player 10.2. Flash Player 10.1 introduced support for hardware-accelerated *decoding* for some types of video content. Now, Flash Player 10.2 uses Stage Video to accelerate the *presentation* of content, as well.

The traditional method for rendering video in Flash Player uses the Video API pipeline. The `Video` object is treated as is any other display object on the stage. This allows the developer a great deal of creative control to blend or reshape videos, for example, but it comes at the cost of increased CPU usage.

The `flash.media.StageVideo` object, however, is not a display object. Instead, it sits behind the stage, behind all display objects. This allows Flash Player to take advantage of the device's graphics processing unit (GPU) when rendering the video. This, in turn, greatly decreases the work of the CPU and allows higher frame rates and greater pixel fidelity and quality.

OSMF uses the Stage Video pipeline by default, whenever it is available on a given device or system. Whenever Stage Video becomes unavailable, OSMF falls back to the normal Video pipeline for displaying streams. During the playback of a video stream, OSMF can switch between Stage Video and Video mode multiple times, as the availability of Stage Video changes. If Stage Video becomes available in the middle of playback, the switch still happens. The `Video` object is only used if a version of Flash Player prior to 10.2 is installed on the client device, or if Flash Player 10.2 or newer is available, but the device itself is not capable of using Stage Video.

Because the device's hardware displays the video, using a `StageVideo` object has the following constraints compared to a `Video` object:

- The number of `StageVideo` objects that can concurrently present videos is limited by the hardware.
- The video timing is not synchronized with the timing of any Flash content that the runtime presents.
- The video display area can only be a rectangle. You cannot use more advanced display areas, such as elliptical or irregular shapes. This supports the most common case for presenting video: a rectangular display area overlaid with video controls.
- You cannot rotate, apply filters, color transformations, or an alpha value to the video. Blend modes that you apply to display objects that are in front of the video do not apply to the video.
- You cannot bitmap cache the video.

OSMF does not currently provide support for StageVideo API properties such as zoom or pan.

For more information on Stage Video, see http://www.adobe.com/devnet/flashplayer/stagevideo.html

Enabling and Disabling Stage Video in OSMF

Existing OSMF-based video players are given Stage Video capability simply by compiling against the OSMF 1.6 release, with two caveats:

- If the movie is not being displayed in full-screen mode, Stage Video must be activated by setting your HTML page's `wmode` parameter to `direct` (`wmode: "direct"`). In full-screen mode, Stage Video is available with any `wmode` value.
- Because Stage Video renders under the display list, using opaque backgrounds under video assets can obscure the `StageVideo` object. If the video is not visible, check to see if you have an opaque object in your display list that could cover the Stage Video. To address this issue, you can make the display object transparent, for example. Similarly, in Flex applications, the video is not displayed unless you add `backgroundAlpha="0"` to the Application container and to any other containers that could obscure the video.

The availability of the `StageVideo` objects is checked when a `LightweightVideoElement` or `VideoElement` object is added to the Stage.

Note: *Use the* `-swf-version=11` *compilation argument when compiling media players with the new OSMF 1.6 binaries or source code. However, even though you must use the new compilation target to build an OSMF 1.6 player, your users are not forced to upgrade to Flash Player 10.2 when viewing media on the player.*

If you have a player that uses features which are currently unsupported by Stage Video, you can still use OSMF 1.6 and exercise the option to disable Stage Video support. To turn off Stage Video support in an OSMF player, simply add the following snippet to your player's code, before creating any media element:

```
OSMFSettings.enableStageVideo = false;
```

Changing the value of this property does not affect current media elements. It affects only the creation of future media elements.

Chapter 3: Interaction

"Basic Playback" on page 3 describes how to create a simple media player. What the player displays, however, is minimal: a window containing content that automatically begins playback. For the user to interact with your player at all, you must implement an interface. Open Source Media Framework provides multiple means of doing so.

You can start by looking at the OSMF player code posted at Strobe Media Playback. The Strobe Media Playback (SMP) project implements a full player interface with event handling. The code is open source and can be edited and reused. For more information, see the documentation at the SMP wiki.

You can also build your own interface for your player. One way to do this is to construct individual interface pieces, add them to the player's media container, and manage their response to events. An example of this process is presented in "Simple user interaction" on page 11.

Another way to build a player interface is to implement it as a plug-in. "Writing a control bar plug-in" on page 38 provides an example implementation.

Simple user interaction

The `Sprite` class is a basic display list building block: a display list node that can display graphics and can also contain children. `Sprite` is an appropriate base class for objects that do not require timelines, such as user interface (UI) components. This example implements a piece of the interface (an overlay Play button) as a `Sprite` object. After the `Sprite` is defined, it is added to the player's media container and given the ability to respond to events.

Note: *When creating a real-world player, more interaction than just an overlay Play button is usually required. Scrub bars, audio buttons, pause buttons, and more are common parts of a player user interface. These controls can also be created using the same approach that is taken for the overlay Play button.*

The following function creates the `Sprite` object and constructs the Play button image that visually overlays the player's content area.

```
private function constructPlayButton():Sprite
{
    var result:Sprite = new Sprite();

    var g:Graphics = result.graphics;
    g.lineStyle(1, 0, 0.5);
    g.beginFill(0xA0A0A0, 0.5);
    g.moveTo(X - SIZE / F, Y - SIZE);
    g.lineTo(X + SIZE / F, Y);
    g.lineTo(X - SIZE / F, Y + SIZE);
    g.lineTo(X - SIZE / F, Y - SIZE);
    g.endFill();

    return result;
}

private static const SIZE:Number = 100;
private static const X:Number = 320;
private static const Y:Number = 180;
private static const F:Number = 1.2;
```

You then add the `Sprite` object to the same container that holds the media element by calling `addChild`:

```
var playButton:Sprite = constructPlayButton();
addChild(playButton);
```

Next, you add functionality to the button, so that your player can respond when the user clicks Play. You do this by attaching an event listener to the `Sprite` object. This allows the object to receive notification for whatever events you specify. In this case, when a mouse click is detected on the Play button, it causes the media player to begin playback.

```
playButton.addEventListener
    ( MouseEvent.CLICK,
        function():void{ mediaPlayer.play(); }
    );
```

Finally, your player must synchronize the visibility of the Play button with the current state of playback. If playback is paused, the Play button should be visible so the user can start playback up again. But if playback is occurring, the Play button should not be displayed.

To accomplish this, you can add another event listener, this time to your media player class. Your player can then respond to `PlayEvent.PLAY_STATE_CHANGE` by setting the Play button visibility on and off, accordingly.

```
mediaPlayer.addEventListener
    ( PlayEvent.PLAY_STATE_CHANGE
    , function():void{ playButton.visible = !mediaPlayer.playing; }
    );
```

The following sample shows the construction of the Play button overlay in its entirety:

```
package
{
    import flash.display.Graphics;
    import flash.display.Sprite;
    import flash.events.MouseEvent;
    import org.osmf.containers.MediaContainer;
    import org.osmf.events.PlayEvent;
    import org.osmf.layout.LayoutMetadata;
    import org.osmf.media.*;

    [SWF(width="640", height="360", backgroundColor="0x000000",frameRate="25")]
    public class MyOSMFPlayer extends Sprite
    {
        public function MyOSMFPlayer()
        {
            mediaPlayerSprite = new MediaPlayerSprite();
            mediaPlayerSprite.mediaPlayer.autoPlay = false;
            mediaPlayerSprite.resource = new URLResource(VIDEO_URL));
            // Give the bounds of the MediaPlayerSprite
            mediaPlayerSprite.width = 640;
            mediaPlayerSprite.height = 360;
            addChild(mediaPlayerSprite);

            var playButton:Sprite = constructPlayButton();
            addChild(playButton);

            playButton.addEventListener
                (MouseEvent.CLICK,
                 function():void{mediaPlayerSprite.mediaPlayer.play();} );

            mediaPlayerSprite.mediaPlayer.addEventListener
```

```
                (PlayEvent.PLAY_STATE_CHANGE, function():void{playButton.visible =
                 mediaPlayerSprite.mediaPlayer.playing;} );
        }
    }

    private var mediaPlayerSprite:MediaPlayerSprite;

    private static const SIZE:Number = 100;
    private static const X:Number = 320;
    private static const Y:Number = 180;
    private static const F:Number = 1.2;

    private static const VIDEO_URL:String = "http://myserver.net/content/test.flv";
    private function constructPlayButton():Sprite
    {
        var result:Sprite = new Sprite();

        var g:Graphics = result.graphics;
        g.lineStyle(1, 0, 0.5);
        g.beginFill(0xA0A0A0, 0.5);
        g.moveTo(X - SIZE / F, Y - SIZE);
        g.lineTo(X + SIZE / F, Y);
        g.lineTo(X - SIZE / F, Y + SIZE);
        g.lineTo(X - SIZE / F, Y - SIZE);
        g.endFill();

        return result;
    }
}
```

Chapter 4: Delivering Content

The intended audience for your media player may be inside an enterprise network or distributed around the world. Knowing the strengths and limitations of the various "Delivery methods" on page 14 available to an Open Source Media Framework player is the first step in choosing how you deliver content. OSMF also provides various delivery enhancements for your player, as described in "Optimizing delivery with Dynamic (MBR) Streaming for HTTP and RTMP streams" on page 17, "Delivering DVR features in HTTP and RTMP live streams" on page 19, and "Supporting alternate audio tracks in HTTP Streaming videos" on page 22.

Delivery methods

OSMF supports a variety of delivery protocols:

- HTTP, for live and recorded streaming, as well as progressive download
- RTMP, for live and recorded streams

OSMF also supports advanced playback and delivery features for HTTP and RTMP streams. These include digital video recording features and multi-bitrate ("adaptive" or "dynamic") streaming.

Additionally, new with Flash Media Server 4.0, OSMF 1.5 supports:

- RTMFP, to enable multicasting of live streams

About HTTP streaming

HTTP has long been the most basic of content delivery methods. Using HTTP, applications can play video from a standard web server simply by making a request to download a file. This delivery method is known as "progressive" download and has a number of downsides for modern media players. These negatives include greater than necessary bandwidth consumption and the inability to play live content.

OSMF uses Flash Player (version 10.1 or greater) and Flash Media Server (version 3.8 or greater) to provide an alternative to progressive download: HTTP *streaming*. With HTTP streaming, your player can deliver high-quality content more efficiently and with more features, while still using standard HTTP servers. Supported content types include:

- Recorded (does not require Flash Media Server)
- Recorded multi-bitrate ("dynamic")
- Live
- Live multi-bitrate
- Live with DVR functionality
- Live multi-bitrate with DVR functionality

HTTP streaming also supports each of the previous use cases with Flash Access content protection.

Using HTTP streaming

To use HTTP streaming for a media file, you must supply metadata describing how the media is packaged. You provide this metadata to OSMF in the form of a Flash Media Manifest file (file extension: F4M). Once you create an F4M file, you pass its URL to `DefaultMediaFactory`, which automatically handles loading the stream.

An F4M manifest file is an XML document containing metadata related to a single piece of media. The manifest file can contain references to more than one instance of the media, such as for multiple bit rates. However, the manifest does not encapsulate more than one distinct piece of media and does not function as a playlist.

Manifest information can include the location of the media, DRM authentication information, media bootstrap information, multi-bitrate information (MBR), and more. For a complete description of the F4M format, see the Flash Media Manifest File Format Specification.

For details on the larger HTTP streaming workflow, including methods of creating F4M manifests for live and recorded content, see the HTTP streaming overview in the document Using Adobe HTTP Dynamic Streaming.

For further information on using HTTP streaming features in your OSMF player, see "Optimizing delivery with Dynamic (MBR) Streaming for HTTP and RTMP streams" on page 17, "Delivering DVR features in HTTP and RTMP live streams" on page 19, and "Supporting alternate audio tracks in HTTP Streaming videos" on page 22.

About RTMP streaming

Because RTMP was designed as a streaming protocol, it has had many upsides over standard HTTP progressive delivery. These include supporting both recorded and live streams, multi-bitrate streaming, and DVR capabilities. Using Flash Player 10.1 and Flash Media Server 3.5.3, OSMF provides support for new RTMP features, including Stream Reconnect.

Stream Reconnect automatically allows an RTMP stream to continue to play through the buffer when a connection is disrupted. By default, as specified by the `NetLoader.reconnectTimeout` property, OSMF tries to reconnect an RTMP stream for 120 seconds. If the stream is reconnected before the buffer empties, the stream is not interrupted, and buffering restarts unnoticeably. In general, the greater the buffer size, the less chance of stream interruption. "Managing buffer size for RTMP and HTTP streams" on page 19 provides recommendations on setting the buffer size for RTMP streams.

`DefaultMediaFactory` loads RTMP streams automatically. However, it is important to note that, by convention, RTMP streams should not include a file extension as part of their URL.

Note: *When playing RTMP live streams, your player should not enable Pause button functionality, unless the live stream also has DVR support.*

See the following for more information on using RTMP features in an OSMF player:

- "Optimizing delivery with Dynamic (MBR) Streaming for HTTP and RTMP streams" on page 17
- "Delivering DVR features in HTTP and RTMP live streams" on page 19

About RTMFP multicasting

HTTP and RTMP are both application-level protocols built on TCP (transmission control protocol). Using TCP is a standard method to obtain reliable data transfers over the web. However, the reliability of TCP comes at the cost of potentially high data latency.

Services that require low latency, as for live interaction with voice over IP or for online games, use the UDP protocol. UDP (user datagram protocol) is an alternative to TCP that delivers data rapidly. UDP is also compatible with packet broadcast (sending to all nodes on a local network) or multicast (sending to all subscribers).

Using UDP multicast, you can build a media player to

- broadcast enterprise-scope information, such as a company meeting
- broadcast video within and beyond your network without a content delivery network (CDN)
- allow internal network clients to participate in a peer-to-peer (P2P) group, to help ensure high-quality video delivery

Using UDP, Adobe has developed the Real-Time Media Flow Protocol (RTMFP), an alternative to TCP-based RTMP and HTTP. To use RTMFP multicast, clients must run an application built for Adobe Flash Player 10.1 or greater or Adobe® AIR®. There are three forms of multicast that RTMFP supports:

Native IP multicast. With IP multicast, a multicast address is associated with a group of interested receivers. The sender sends a single datagram to the multicast address. Then, the intermediary routers copy the message and send the copies to receivers who've registered their interest in data from that sender.

P2P (application-level) multicast. P2P multicast uses Flash Player applications within the multicast group to route and relay data. Because the RTMFP protocol supports groups, this enables an application to segment its users to send messages and data only between members of the group. Application-level multicast provides one-to-many (or a few-to-many) streaming of continuous live video and audio.

Fusion multicast. With Fusion multicast, native IP multicast and P2P/application multicast are combined. Clients can receive content via native IP multicast or fall back to P2P/application multicast if IP multicast is not available.

RTMFP is a managed connection that requires the authorization of a Flash Media Server 4.0 instance to make the introductions. Clients must remain connected to the server to retain the direct connection. Additionally, the server provides a script that handles publish/unpublish requests and republishes the live stream into a target Flash group. For further information on RTMFP, see the Building Peer-Assisted Networking Applications in the Flash Media Server 4 Developer's Guide.

***Note:** Dynamic streaming and DVR functionality (such as pausing or stopping a video) are not supported with RTMFP multicast at this time.*

Using RTMFP multicasting

The OSMF `DefaultMediaFactory` class automatically handles multicast content. You can provide a multicast-enabled resource to `DefaultMediaFactory` in two ways:

- Use the Flash Media Server "Configurator" tool to generate a multicast-enabled F4M manifest file. In Windows, the default location for the Configurator is `C:\Program Files\Adobe\Flash Media Server 4\tools\multicast\configurator`. See "Using HTTP streaming" on page 15 for more information on F4M files.
- Or, use the `MulticastResource` class to provide a multicast-ready resource for `DefaultMediaFactory` to handle.

When creating and identifying an RTMFP multicast, two important values are the multicast group specification and the stream name.

- The group specification is a string identifying the peer-to-peer group to join, including its name, capabilities, restrictions, and the authorizations of this member.
- The stream name is a string identifying the stream to join. If the supplied string does not refer to an existing application on the server, it is invalid. However, an invalid string only results in a connection not being established; no error is returned.

***Note:** If you use the Flash Media Server 4.0.0 Configurator to generate the F4M file, you must manually change the names of two generated values. Change the name `rtmfpGroupspec` to `groupspec`, and the name `rtmfpStreamName` to `multicastStreamName`.*

The following code excerpts compare the ways to instantiate a multicast stream.

Using an F4M file:

```
mediaFactory = new DefaultMediaFactory();
mediaPlayer.media = mediaFactory.createMediaElement
    (
        new URLResource("http://example.com/multicast.f4m")
    );
```

Using `MulticastResource` with `DefaultMediaFactory`:

```
mediaFactory = new DefaultMediaFactory();
var resource:MulticastResource = new MulticastResource("rtmfp://server/app");
resource.groupspec = "AJHJKSFDS";
resource.streamName = "OPIOPROPIIOPOP";
mediaPlayer.media = mediaFactory.createMediaElement(resource);
```

Note: *For IP multicast, OSMF does not return an error when attempting to connect to a blocked network segment.*

Optimizing delivery with Dynamic (MBR) Streaming for HTTP and RTMP streams

Delivery quality, specifically how well a player responds to bandwidth changes, is a key differentiator of media players. Dynamic Streaming helps your player adapt to changing network conditions.

With Dynamic Streaming, when the player senses a network bandwidth change, it responds by switching playback to a content file with a more appropriate bit rate. Dynamic Streaming requires you to have multiple bit rate (MBR) versions of content for the player to switch among. When you provide MBR content, a network experiencing a temporary reduction in bandwidth does not have to pause playback for the user. Instead, the player shifts to using a lower bit rate version of the content that is playing. If the player does not find multi-bitrate versions of content, Dynamic Streaming does not function.

To enable HTTP Dynamic Streaming, you supply OSMF with an F4M file, as described in these Quick Start tutorials.

For RTMP Dynamic Streaming, you can use `MediaPlayerSprite` and `DynamicStreamingResource` as shown in the following sample. `DynamicStreamingResource` contains multiple `DynamicStreamingItem` values, each of which represents a single stream.

```
package
{
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import org.osmf.elements.VideoElement;
    import org.osmf.media.MediaPlayerSprite;
    import org.osmf.net.DynamicStreamingItem;
    import org.osmf.net.DynamicStreamingResource;

    public class DynamicStreamingExample extends Sprite
    {
        public function DynamicStreamingExample()
        {
        super();

            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            var mediaPlayerSprite:MediaPlayerSprite = new MediaPlayerSprite();

            var videoElement:VideoElement = new VideoElement();

            /**
            * By convention, most rtmp URLs do not use file extensions.
            */
            var dynResource:DynamicStreamingResource = new
                DynamicStreamingResource("rtmp://myserver.net/ondemand");

            /**
            * The first two parameters of each DynamicStreamingItem specify
            * stream name and bitrate and are required.
            * The second two parameters of each DynamicStreamingItem are optional;
            * they specify the stream's width and height, in that order.
            */
            dynResource.streamItems = Vector.<DynamicStreamingItem>(
                [ new DynamicStreamingItem("mp4:myserver/content/
                    demo_768x428_24.0fps_408kbps.mp4", 408, 768, 428)
                , new DynamicStreamingItem("mp4:myserver/content/
                    demo_768x428_24.0fps_608kbps.mp4", 608, 768, 428)
                , new DynamicStreamingItem("mp4:myserver/content/
                    demo_1024x522_24.0fps_908kbps.mp4", 908, 1024, 522)
                , new DynamicStreamingItem("mp4:myserver/content/
                    demo_1024x522_24.0fps_1308kbps.mp4", 1308, 1024, 522)
                , new DynamicStreamingItem("mp4:mmyserver/content/
                    demo_1280x720_24.0fps_1708kbps.mp4", 1708, 1280, 720)
                ]);

            videoElement.resource = dynResource;

            addChild(mediaPlayerSprite);
            mediaPlayerSprite.media = videoElement;
        }
    }
}
```

Managing buffer size for RTMP and HTTP streams

OSMF does not set a default size for the RTMP or HTTP stream buffer. You must set `MediaPlayer.bufferTime` to a size appropriate to the needs of your player.

- **For RTMP Dynamic Streaming/MBR content,** set the buffer size to 8 seconds or greater. This applies to any RTMP dynamic stream, whether live or recorded. For live MBR content, you must set the buffer size to a nonzero value or MBR switching fails.
- **For RTMP streams that are not dynamic/MBR,** set the buffer size to between 2 and 4 seconds.
- **For HTTP Dynamic Streaming,** see the sample HTTPStreamingNetLoaderWithBufferControl.as in the OSMFPlayer source repository for an example of managing buffer size.

Delivering DVR features in HTTP and RTMP live streams

In the past, there was no way to view live content with a user experience comparable to that of recorded content. Recorded content could be played back at any time, with standard DVR-type controls such as pause and rewind. In contrast, with live streaming, users could join an event and see live content, but they could not pause or rewind while the recording was still in progress.

Live DVR is the process of recording/caching a live stream on a server, while the event is still being recorded. Users can not only view the live stream, but they can also pause, rewind, and seek in the recorded sections of the stream.

OSMF provides live DVR support for both HTTP dynamic streams and RTMP streams.

Live DVR and Rolling Window for HTTP dynamic streams

On the server side, providing the core DVR features for HTTP live streams requires that the server be equipped with Flash Media Server 3.8 or above. Enabling DVR is specified by the `<dvrInfo>` tag within the F4M file produced by the live packager. For instructions on properly setting up a DVR-enabled F4M file, see Tutorial: Live HTTP Dynamic Streaming with DVR.

In addition to the basic live DVR features, OSMF 1.6 also provides the ability to work with a "rolling window" for HTTP live streams. This feature is not currently available for RTMP live streams.

Note: *Live DVR rolling window requires OSMF 1.6, Flash Media Server 4.5, and a 2.0 format Flash Media Manifest (F4M 2.0) file.*

About DVR rolling window

Because offering unlimited access to many hours of cached live content is rarely practical, content providers need to be able to set a maximum duration for the viewing "window" that is available. However, because the event may continue for longer than the maximum window duration, the available content window can be said to "roll" forward with the current live point.

Example: A live event begins at 09:00, with a maximum available window duration of 1 hour. At 09:05, at the beginning of the event, there is very little cached content to view, so the actual amount of viewable content is only 5 minutes. At 10:00, the cached content is a full 1 hour in length, so a user can access a full hour of content, extending from 09:00 to 10:00. However, as further time elapses and the live event continues, the available content window rolls forward, such that someone viewing the event at 10:30 only has access to a window of content from 09:30 to 10:30. The previously available content beginning at 09:00 is no longer accessible to viewers.

Setting a DVR rolling window in Flash Media Server

With Flash Media Server 4.5, content providers can use the disk management feature, together with a version 2.0 Flash Media Manifest (F4M 2.0) file, to specify a DVR rolling window.

By default with Flash Media Server 4.5, disk management is on and stores 3 hours of a live stream (this value can be adjusted in fractional units of an hour). the following Event.xml file stores 1.5 hours (90 minutes) of content, as specified by the `DiskManagementDuration` value:

```
<Event>
    <EventID>liveevent</EventID>
    <Recording>
    <FragmentDuration>4000</FragmentDuration>
    <SegmentDuration>60000</SegmentDuration>
    <DiskManagementDuration>1.5</DiskManagementDuration>
    </Recording>
</Event>
```

To provide access to a specified amount of this content cache, the content provider can set the `windowDuration` attribute for the `dvrInfo` tag in an F4M 2.0 Manifest.xml file.

Note: *The `dvrInfo` tag in F4M 1.0 files contains the `beginOffset` and `endOffset` attributes, which could be used to create a type of open-ended rolling window. These attributes are deprecated with F4M 2.0.*

Set `windowDuration` as follows:

- To a value greater than zero, to indicate the number of seconds of recorded content that are available behind the live point. With HTTP streaming, the stream is typically a number of seconds behind the live point in most cases, so for DVR to behave predictably, a value of 60 seconds is recommended as the minimum `windowDuration`.
- To a value of negative one (-1), to indicate that the available recorded content behind the live point is unlimited. This is the default setting.
- To accommodate seeking and pausing at the beginning of the rolling window, the maximum `windowDuration` should be (`DiskManagementDuration - segmentDuration`). Setting `windowDuration` to a larger value than this may cause unpredictable behavior.
- Setting `windowDuration` to zero is not recommended. To indicate a stream is "pure" live, with no DVR capability, the `dvrInfo` tag should be omitted.

The following Manifest.xml file sets a rolling window of 60 minutes' (3600 seconds) size:

```
<dvrInfo windowDuration="3600"></dvrInfo>
```

Note: *To set a DVR rolling window, use the Flash Media Server Set-level File Generator. This tool is installed to the <flashmediaserver>/tools/f4mconfig folder. You must use f4mconfig.html to create the file to load the individual F4M created by Flash Media Server. Alternately, you can (1) copy a file obtained as a Flash Media Server response, (2) set the `baseURL` for the file to the position of the first F4M file, and (3) edit the namespace manually: `<manifest xmlns="http://ns.adobe.com/f4m/2.0">`*

Working with a DVR rolling window in OSMF

You are not required to make any changes to an existing OSMF player that supports classic DVR, to support DVR rolling windows. OSMF manages the rolling window (including its current duration and live point) automatically. If the stream is recording, OSMF always starts playback from the live point, regardless of the use of `dvrInfo` and `windowDuration`. This behavior is called "snap to live" and is new with OSMF 1.6.

You can obtain the specified size of a rolling window from the `windowDuration` property of the `DVRTrait` for the media element associated with the content. However, because `DVRTrait.windowDuration` does not provide the actual amount of available content, the true amount of recorded content could be less than this value if the event has been happening for less than `windowDuration`.

To determine the actual amount of available content, as well as the current position of the playhead, you must examine properties of the media element's `TimeTrait`, as follows:

- `TimeTrait.duration` - the total duration of the available cached content.
- `TimeTrait.currentTime` - the current position of the playhead, relative to the beginning of the rolling window; this value never exceeds `TimeTrait.duration`.

You can use these values, along with the `windowDuration` property, if you want to customize your player's user interface. For example, you might want to determine how long the player's scrub bar should be, or whether to use times relative to the beginning of the content or relative to the live point. An example of the latter would be to calculate `TimeTrait.currentTime - TimeTrait.duration`. This would allow your player to display times prior to the live point as negative numbers, reflecting the number of seconds the playhead is behind the live event. However, again, there is no requirement for you to do any of this or to use the `DVRTrait.windowDuration` property at all, in order to have a working player.

The following table provides an example of OSMF's behavior when playing content with a DVR rolling window. This example is for a viewing session that begins 30 minutes after the start of the event recording, but which is immediately "rewound" 15 minutes back from the live point; `windowDuration` has been set by the content provider to be 60 minutes.

Elapsed time	TimeTrait.duration	TimeTrait.currentTime	Comments
30	30	15	User starts viewing and immediately seeks back to the middle of the available content
45	45	30	After 15 more minutes have passed, the elapsed time for the event is still less than the maximum size set for the rolling window
60	60	45	The event's recorded content has reached the size limit set by windowDuration
75	60	45	The event recording has exceeded windowDuration, so OSMF trims the available content to equal windowDuration. The current playhead time remains at 45 minutes because the user is still 15 minutes behind the live point within a rolling window of 60 minutes' duration.
90	60	45	The event concludes at minute 90, but the user is still 15 minutes behind the live point of the 60-minute rolling window.
105	60	60	15 minutes after the event conclusion, the user has now caught up to the end of the available 60-minute cache.

While there are no required player changes to make use of DVR rolling window, it is recommended that you implement a simple safety mechanism for your player, to ensure that the user doesn't "fall out" of the rolling window. While the Strobe Media Playback project does not officially support DVR rolling window, it does provide an example of this safety mechanism at `player/StrobeMediaPlayback/src/StrobeMediaPlayback.as`. The player listens to `TimeEvent.CURRENT_TIME_CHANGE` for the MediaPlayer, and if the time becomes closer to the margin of the window than that specified by a preset value, the player seeks a set amount forward.

Live DVR for RTMP streams

To provide DVR capability for RTMP streams requires a minimum of Flash Media Server 3.5 and the DVRCast application

OSMF automatically supports RTMP streams with DVR functionality, as long as the streams are properly identified when the media element is constructed. You must use a `StreamingURLResource` object (rather than a plain `URLResource`) and set the stream type property of the resource to `streamType.DVR`.

Supporting alternate audio tracks in HTTP Streaming videos

OSMF 1.6 supports multiple language tracks for HTTP video streams, without requiring duplication and repackaging of the video for each audio track. This "late" binding of an audio track to a video stream allows content publishers to easily provide multiple language tracks for a given video asset, at any time before or after the asset's initial packaging.

The initial packaging of the video asset can include an audio track, and the publisher can still choose to provide one or more additional audio tracks for the video. OSMF then provides the support that allows you to give viewers the option of switching to an alternate audio track either before or during playback.

For your player, OSMF signals that multiple audio streams are available. It also provides a mechanism to iterate through the available alternate streams.

OSMF detects the presence of alternate audio tracks from the F4M manifest. If an alternate audio track is selected, OSMF obtains the fragments for that track, and replaces the audio packets from the main video stream with the alternate ones. OSMF applies (multiplexes) the selected alternate track before passing the video stream to your player.

Late-binding audio supports:

- Changing audio tracks both before the beginning of playback and during playback.
- Live and VOD (recorded) content.
- Playback of multibitrate (MBR) video streams with alternate (non-MBR) audio streams. Current MBR behavior and logic should work as expected and the selected alternate tracks are maintained during the playback and not affected by bit-rate switching.
- Playback of DVR video streams with alternate audio streams. Current DVR behavior and logic should work as expected and the selected alternate tracks are maintained during playback and unaffected by going to live or seeking back into the DVR content.
- Attribute tags for alternate audio tracks: `lang` (for the language of the track) and `label` (for additional descriptions, such as "director's commentary" or "audio descriptions for the visually impaired").
- Gap resilience. If a selected alternative audio track becomes unavailable during playback, playback continues without audio. Thereafter, at the beginning of each fragment OSMF attempts to reactivate the selected audio track. OSMF also dispatches an `NetStream.Play.StreamNotFound` event containing the URL for the audio track, so that you can have your player respond to the failure.

The following limitations with late-binding audio currently apply:

- Multi-bitrate (MBR) audio tracks are unsupported.
- Switching between audio tracks is not typically instantaneous; there may be a short delay of a second or more.

The following snippet is taken from the OSMF sample HelloWorld8.as, which demonstrates simple event handling for a video with alternate audio streams:

```
// Assuming the media container and media factory exist, and that the media
// element for this video has been created from an F4M file, we create the media
// player and add event listeners to handle alternative audio stream changes.

_player = new MediaPlayer();
_player.addEventListener(MediaPlayerStateChangeEvent.MEDIA_PLAYER_STATE_CHANGE,
onPlayerStateChange);
_player.addEventListener(AlternativeAudioEvent.NUM_ALTERNATIVE_AUDIO_STREAMS_CHANGE,
onNumAlternativeAudioStreamsChange);
_player.addEventListener(AlternativeAudioEvent.AUDIO_SWITCHING_CHANGE,
onAlternateAudioSwitchingChange);
// Listen for relevant player state change events, such as READY.
private function onPlayerStateChange(event:MediaPlayerStateChangeEvent):void
{   switch (event.state)
    {
        // When the media is ready, you can see what alternative audio tracks are present.
        case MediaPlayerState.READY:
            if (!_playbackInitiated)
            { _playbackInitiated = true; trace("Alternative languages",
                _player.hasAlternativeAudio ? "available" : " not available" );
                if (_player.hasAlternativeAudio)
                {   for (var index:int=0; index<_player.numAlternativeAudioStreams;
                        index++)
                    { var item:StreamingItem=_player.getAlternativeAudioItemAt(index);
                     trace("[LBA] [", item.info.language, "]", item.info.label);
                    }
                    // This is a good point to allow the user to select a language.
                    // Here we pass 0, which plays the first alternative track.
                    // Passing -1 plays the video stream's embedded audio track.
                     _player.switchAlternativeAudioIndex(0);
                }
                if (_player.canPlay)
                {
                    _player.play();
                }
            }
            break;
        // Handle other player state change events (such as PLAYING, etc.) here.
    }
}
// Listen for a change in the number of alternative streams associated with this video.
private function onNumAlternativeAudioStreamsChange(event:AlternativeAudioEvent):void
{ if (_player.hasAlternativeAudio)
    { trace("Number of alternative audio streams = ", _player.numAlternativeAudioStreams); }
}

// Listen for when an audio stream switch is in progress or has been completed.
private function onAlternateAudioSwitchingChange(event:AlternativeAudioEvent):void
{ if (event.switching)
    { trace("Alternative audio stream switch in progress"); }
    else
    { trace("Alternate audio stream switch completed"); trace("Current alternate audio index
is [" + _player.currentAlternativeAudioStreamIndex + "]."); }
}
```

Chapter 5: Enhancing Content

You can use OSMF to customize the content your player plays. The following enhancement mechanisms are a good starting point. You may also want to read "Advanced Topics" on page 31 for more possibilities.

- `ProxyElement` is a powerful `MediaElement` subclass. Learn how "Proxy elements" on page 24 work, then try out `DurationElement` to create a slide show.
- OSMF also defines a mechanism for the use of metadata. See "Metadata" on page 25 for information on the OSMF `Metadata` subclasses to use for content layout and cue points.

Proxy elements

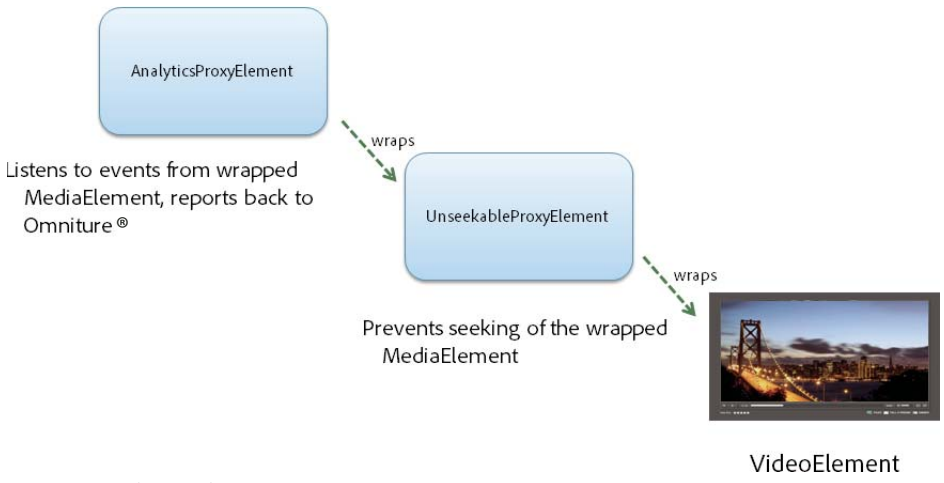
A `ProxyElement` is a class that wraps (proxies) another `MediaElement`, exposing the same API. The purpose of this class is to control access to the wrapped element. The `ProxyElement` class allows layering of functionality and is a core building block for plug-ins, as described in "Plug-ins" on page 36.

`ProxyElement` is not instantiated directly but rather used as the base class for creating wrappers for specific purposes. `ProxyElement` can be subclassed for any trait or set of traits. The subclass controls access to the wrapped element either by overriding one or more of the wrapped element's traits or by blocking them. This technique is useful because it allows you to noninvasively modify the behavior of another `MediaElement`.

`DurationElement` is an OSMF-provided `ProxyElement` subclass that adds temporal capabilities to a non-temporal media element. For example, you can use it to wrap a set of `ImageElement` objects to create a slide show, as explained in "Using DurationElement" on page 25.

Other `ProxyElement` possibilities include:

- Unskippable video ads, implemented by a `ProxyElement` that blocks the `Seek` trait of its wrapped `MediaElement`.
- User analytics, via a `ProxyElement` that listens for changes to the wrapped `MediaElement` and reports them to a server.
- Seamless video switching, from a `ProxyElement` that wraps two `VideoElement` objects, and switches from one to the other without rebuffering.



Nesting ProxyElement objects

Using DurationElement

`DurationElement` is a proxy element that provides temporal capabilities. You can use by itself to provide delays between the presentation of other media elements in a composition. Or, you can use `DurationElement` to wrap a non-temporal `MediaElement`, such as an `ImageElement`, to give it temporal capabilities.

`DurationElement` overrides the wrapped element's `TimeTrait` to return a custom instance of that trait. `DurationElement` does not work with elements that already have a `TimeTrait`, such as `VideoElement`. See "Traits" on page 33 for a more detailed discussion of this type.

This first example shows `DurationElement` instances being used to provide delays between video elements:

```
var sequence:SerialElement = new SerialElement();

sequence.addChild (new VideoElement(new URLResource
    ("http://www.example.com/video1.flv")));
sequence.addChild(new DurationElement(5));
sequence.addChild(new VideoElement(new URLResource
    ("http://www.example.com/ad.flv")));
sequence.addChild(new DurationElement(5));
sequence.addChild(new VideoElement(new URLResource
    "http://www.example.com/video2.flv")));

// Assign the SerialElement to the MediaPlayer.
player.media = sequence;
```

The following example presents a sequence of rotating banners. The delays separating the appearances of the banners are created with `DurationElement` instances. In addition, the images themselves are wrapped in `DurationElement` objects to enable them to support a duration.

```
// The first banner does not appear for five seconds.
// Each banner is shown for 20 seconds.
// There is a 15-second delay between images.

var bannerSequence:SerialElement = new SerialElement();

bannerSequence.addChild(new DurationElement(5));
bannerSequence.addChild(new DurationElement(20,new ImageElement(new URLResource
    ("http://www.example.com/banner1.jpg")));
bannerSequence.addChild(new DurationElement(15));
bannerSequence.addChild(new DurationElement(20,new ImageElement(new URLResource
    ("http://www.example.com/banner2.jpg")));
bannerSequence.addChild(new DurationElement(15));
bannerSequence.addChild(new DurationElement(20,new ImageElement(new URLResource
    ("http://www.example.com/banner3.jpg")));
```

Note: *To specify a start and end time for a video element, use the `StreamingURLResource` properties `clipStartTime` and `clipEndTime`.*

Metadata

OSMF defines metadata as key-value pairs, where keys are strings, and values are arbitrary objects that can be any ActionScript data structures. The OSMF `Metadata` class provides a strongly typed API for working with these key-value pairs, as well as events for detecting changes to the metadata.

Two areas where OSMF provides defined metadata mechanisms are for "Layout control" on page 26 and "Timelines and cue points" on page 28.

Layout control

As described in "Basic Playback" on page 3, using the `MediaContainer` class (either directly or indirectly via `MediaPlayerSprite`) is what gives your player visibility. `MediaContainer` also provides control over how content is sized and positioned within your media player.

When you construct a media container, you can specify `LayoutMetadata` properties that apply to each `MediaElement` child of the container. You use these properties to position and scale the child media elements. If you are using `MediaPlayerSprite`, you can still set `LayoutMetadata` properties for the child `Sprite` object.

Default layout metadata properties

If you don't specify specific layout parameters for a media element, OSMF creates a default set when that element is rendered. Note that if you do set any layout parameters (aside from `index`) for a media element then you must set them all, because OSMF then does not apply the default parameters.

The default layout parameters, and their default values, are:

- `index`: By default any newly created media element is placed at the top of the display list. The index property is the only default layout parameter that can be set separately from the rest of the layout attributes. This allows you to adjust a given media element's z-index value and still have the remainder of the default layout parameters applied to the element.
- `horizontalAlign`: The default horizontal alignment of an element is at the horizontal `center` of the parent container. Other possible values are `left` and `right`.
- `verticalAlign`: The default vertical alignment of an element is at the verticle `middle` of the parent container. Other possible values are `bottom` and `top`.
- `scaleMode`: The default scale mode is `letterbox`. Other possible values are `none`, `stretch`, and `zoom`. For detailed descriptions of each of these values, see "Controlling content scale with MediaPlayerSprite" on page 8.
- `percentWidth`: This relative layout property is set to 100%, so the element appears as 100% of its parent container's width.
- `percentHeight`: This relative layout property is set to 100%, so the element appears as 100% of its parent container's height.
- `snapToPixel`: This option ensures that the layout renderer positions and sizes elements on whole pixel boundaries. For most media, putting it on sub-pixel boundaries results in a less sharp image, therefore pixel clamping is set to `true` by default.
- `includeInLayout`: By default, this is set to `true`, therefore the element participates in the layout process.
- `layoutMode`: These modes determine the relative positioning of media elements in a container.
  - `NONE`: The default layout mode. With `NONE`, there are no constraints on the relative positions of elements in a container. Note that `NONE` is not recommended for layouts that include composite elements; instead, use `OVERLAY`.
  - `VERTICAL`: In this mode, the container's media elements are positioned in a vertical layout, one below the other. The `y` value for an element is calculated based on the sum of the heights of any previously positioned elements.
  - `HORIZONTAL`: With this mode, media elements are positioned in a horizontal layout, one next to the other. The `x` value for an element is calculated based on the sum of the widths of any previously positioned elements in the container.

- `OVERLAY`: Similar to `NONE`, with `OVERLAY` there are no constraints on the relative positioning of media elements in a container. However, unlike `NONE`, using `OVERLAY` ensures consistent positioning behavior when composite elements are included in the layout.

The following code shows an example of adjusting only the z-index of a media element:

```
var videoElement:VideoElement = new VideoElement(new URLResource(REMOTE_PROGRESSIVE));
var videoLayoutMetadata:LayoutMetadata = new LayoutMetadata();
videoLayoutMetadata.index = 100;
videoElement.addMetadata(LayoutMetadata.LAYOUT_NAMESPACE, videoLayoutMetadata);
```

If you do not explicitly set the index value, the order in which elements are added to a parallel element defines their order in the z-index. That is, the newest display object is by default displayed on top of any others. Thus, the following code results in the duration element being displayed over the video element:

```
var parallelElement:ParallelElement = new ParallelElement();
parallelElement.addChild( videoElement );
parallelElement.addChild( durationElement );
```

Again, to alter this default behavior you manually set the index property.

Optional layout metadata properties

There are additional `LayoutMetadata` properties. You can choose to set these if you want to further adjust the size and visual positioning of the content. They are, in descending order of precedence:

1 **Padding layout properties:**`paddingTop, paddingLeft, paddingBottom, paddingRight` (in pixels). If you have defined padding layout values for a media element, they are not overridden by any other layout metadata settings. Padding values create a blank border around content. The border is created inside the bounds of the current content area, rather than being applied outside the content. Therefore, while padding does not change the size of the element's overall content area, it does reduce the area available to display content.

   Note: *The right and bottom padding margins are created by altering the width or height of the content, respectively. If you do not define* `paddingBottom` *and* `paddingRight`*, then no padding is applied, even if you have set the top and left padding properties.*

2 **Absolute layout properties:**`x, y, width, height` (in pixels). These values are explicit directions on position and size settings. Regardless of the child's context, these values are respected. Since these values are absolute (not relative to the parent container's size), it is possible that the items in the layout do not fit. If that happens, then the basic layout clips out the offending child regions.

3 **Relative layout properties:**`percentX, percentY, percentWidth, percentHeight` (as a percentage of the corresponding container's size). These properties express position and size for the child element as percentages of the parent container size. The absolute layout properties (`x, y, width,` and `height`) take precedence over the relative values that this class defines.

4 **Offset layout properties:**`left, top, right, bottom` (in pixels). These properties define an absolute offset for child content from the parent container's borders. Both absolute layout properties and relative layout properties, defined above, take precedence over these offset layout properties.

The following shows a simple example of applying the `width` and `height LayoutMetadata` properties to the content of a media container.

```
package org.osmf.layout
{
    import flash.display.Sprite;
    import org.osmf.media.MediaPlayerSprite;
    import org.osmf.media.URLResource;

    public class LayoutMetadataExample extends Sprite
    {
        public function LayoutMetadataExample()
        {
            var sprite:MediaPlayerSprite = new MediaPlayerSprite();
            addChild(sprite);
            sprite.resource = new URLResource("http://myserver.com/content/test.flv");

            // Construct a metadata instance and
            // set an absolute width and height of 100 pixels:
            var layoutMetadata:LayoutMetadata = new LayoutMetadata();
            layoutMetadata.width = 100;
            layoutMetadata.height = 100;

            // Apply the layout metadata to the media element at hand, resulting
            // in the video displaying at 100x100 pixels:
            sprite.media.addMetadata(LayoutMetadata.LAYOUT_NAMESPACE, layoutMetadata);
        }
    }
}
```

Timelines and cue points

While you don't have to use OSMF to use cue points, the framework does provide cue point support. The underlying logic for ActionScript cue points in OSMF can be found in `TimelineMetadata` in the `org.osmf.metadata` package.

`TimelineMetadata` encapsulates metadata associated with the timeline of a `MediaElement`. `TimelineMetadata` uses the `TimelineMarker` class to represent both keys and values. That is, a `TimelineMarker` is stored as both key and value. The `TimelineMarker` class represents an individual marker in the timeline of a `MediaElement` object. `TimelineMarker` objects are aggregated by a `TimelineMetadata` object.

A `TimelineMetadata` object dispatches a `TimelineMetadataEvent` when the `currentTime` property of the media element's `TimeTrait` matches any of the time values in its collection of `TimelineMarker` objects.

The `addMarker` method maintains the `TimelineMarker` instances in time order. If another `TimelineMarker` with the same time value exists within this object, then the existing value is overwritten. The AS `TimelineMetadata` cue points are accurate within 250 milliseconds and reliable when the user seeks or pauses.

The `Metadata` subclass `CuePoint` represents a cue point in the timeline of a media element. A cue point is a media time value that has an associated action or piece of information. Typically, cue points are associated with video timelines to represent navigation points or event triggers. The `CuePoint` class extends `TimelineMarker` and can be added to a `TimelineMetadata` object.

You can add custom data via `CuePoint`. If you need more data for a single cue point, you can supply it when you instantiate `CuePoint`. The fourth parameter in the `CuePoint` constructor is an `Object`, which can be anything you want, such as:

```
var cuePoint:CuePoint = new CuePoint(CuePointType.ACTIONSCRIPT, 162.12, "MyCuePoint",
{key1:"value 1", key2:"value 2"});
```

The following sample demonstrates adding an event listener to retrieve embedded cue points for an F4V (H.264) file. These are cue points that were embedded during encoding (that is, at encode time):

```
package
{
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import org.osmf.elements.VideoElement;
    import org.osmf.events.MediaElementEvent;
    import org.osmf.events.TimelineMetadataEvent;
    import org.osmf.media.MediaPlayerSprite;
    import org.osmf.media.URLResource;
    import org.osmf.metadata.CuePoint;
    import org.osmf.metadata.TimelineMetadata;

    public class TimelineMetadataExample extends Sprite
    {
        public function TimelineMetadataExample()
        {
            super();

            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            var mediaPlayerSprite:MediaPlayerSprite = new MediaPlayerSprite();
            var urlResource:URLResource = new URLResource
                ("rtmp://myserver.net/content/test.f4v");
            videoElement= new VideoElement();
            videoElement.resource = urlResource;
            videoElement.addEventListener(MediaElementEvent.METADATA_ADD, onMetadataAdd);
            addChild(mediaPlayerSprite);
            mediaPlayerSprite.media = videoElement;
        }
        private function onMetadataAdd(event:MediaElementEvent):void
        {
            if (event.namespaceURL == CuePoint.DYNAMIC_CUEPOINTS_NAMESPACE)
            {
                var timelineMetadata:TimelineMetadata = videoElement.getMetadata
                    (CuePoint.DYNAMIC_CUEPOINTS_NAMESPACE) as TimelineMetadata;
                timelineMetadata.addEventListener
                    (TimelineMetadataEvent.MARKER_TIME_REACHED, onCuePoint);
            }
        }
        private function onCuePoint(event:TimelineMetadataEvent):void
        {
            var cuePoint:CuePoint = event.marker as CuePoint;
            trace("Cue Point at " + cuePoint.time);
        }
        private var videoElement:VideoElement;
    }
}
```

You can also set runtime ActionScript cue points, as shown here:

```
var dynamicTimelineMetadata = new TimelineMetadata(mediaPlayer.media);
mediaPlayer.media.addMetadata
    (CuePoint.DYNAMIC_CUEPOINTS_NAMESPACE, dynamicTimelineMetadata);

// You can add as many cue points as you like here
var cuePoint:CuePoint = new CuePoint
    (CuePointType.ACTIONSCRIPT, 120/*this is a time value in seconds*/, "my cue point",
        null/*this could be custom name-value pair data*/);

dynamicTimelineMetadata.addMarker(cuePoint);
```

Chapter 6: Advanced Topics

Open Source Media Framework provides the following advanced tools for your player development and customization:

- For debugging and optimizing your code, OSMF supplies detailed log messages. And because OSMF logging data is produced in a standard format, it is shareable between a player and plug-ins. For details, see "Logging" on page 31.
- OSMF media players use the `MediaElement` class to work with content. But, to really understand `MediaElement` or to go beyond it, you need to know about "Traits" on page 33.
- When you embed an OSMF media player in an HTML page, communication between the player and the HTML elements of the page can be necessary. OSMF allows your player to interact with JavaScript, as described in "External HTML control with JavaScript" on page 34.
- Third-party, plug-in services (such as advertising, analytics, or content delivery networks) are widely available to enhance the services that your media player offers. See "Plug-ins" on page 36 for a discussion of how to use these services.

Logging

OSMF logging data is produced in a standard format, making it shareable between a player and plug-ins. The default implementation sends log messages to the debug console of the Flash Builder and Flex Builder development environments.

By default, logging is turned off. To turn it on, set the compiler option `-define CONFIG::LOGGING` to `true`.

The `Logger` class defines the capabilities of a logger, the object that OSMF applications interact with to write logging messages. Usually in an OSMF application there are multiple instances of `Logger`.

The `LoggerFactory` class generates `Logger` objects and serves as the initial contact point for an application. There is typically only one instance of `LoggerFactory` per application. If you want to enhance the logging abilities of your OSMF application, you can subclass the `LoggerFactory` class.

This first code sample provides a brief outline of creating a custom `Logger` subclass:

```
package
{
    import org.osmf.logging.Logger;

    public class ExampleLogger extends Logger
    {
        public function ExampleLogger(category:String)
        {
            super(category);
        }

        override public function debug(message:String, ... rest):void
        {
            trace(message);
        }
    }
}
```

This next sample code creates a new `LoggerFactory` subclass which uses the custom `Logger` subclass:

```
package
{
    import org.osmf.logging.Logger;
    import org.osmf.logging.LoggerFactory;

    public class ExampleLoggerFactory extends LoggerFactory
    {
        public function ExampleLoggerFactory()
        {
            super();
        }
        override public function getLogger(category:String):Logger
        {
            return new ExampleLogger(category);
        }
    }
}
```

The final sample shows the application calling the custom `LoggerFactory`:

```
package
{
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import org.osmf.elements.VideoElement;
    import org.osmf.logging.Logger;
    import org.osmf.logging.Log;
    import org.osmf.media.MediaPlayerSprite;
    import org.osmf.media.URLResource;

    public class LoggerSample extends Sprite
    {
        public function LoggerSample()
        {
            super();

            Log.loggerFactory = new ExampleLoggerFactory();
            logger = Log.getLogger("LoggerSample");

            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            var mediaPlayerSprite:MediaPlayerSprite = new MediaPlayerSprite();
            var urlResource:URLResource = new URLResource
                ("rtmp://myserver.net/content/test_640_500_short");
            var videoElement:VideoElement = new VideoElement(urlResource);

            addChild(mediaPlayerSprite);

            logger.debug("Ready to play video at " + urlResource.url);
            mediaPlayerSprite.media = videoElement;
        }
        private var logger:Logger;
    }
}
```

For a robust example of logging with OSMF, see the Strobe Media Playback debug console. For further information on Strobe Media Playback's implementation of OSMF logging, see the debug console's wiki page.

Traits

"Media elements and resources" on page 3 introduced the ways OSMF represents content via the `MediaElement` class and its subclasses. However, if you want to perform more in-depth work with media elements, it can be useful to understand the `MediaElement` type in detail.

`MediaElement` objects express what they do via traits. Traits represent fundamental capabilities or characteristics of a piece of media. A trait can represent the availability of audio, or the ability to play or seek. The sum of all traits on a media element define the overall abilities of the media element. Traits are dynamic, and are added and removed from media at runtime, in response to other events.

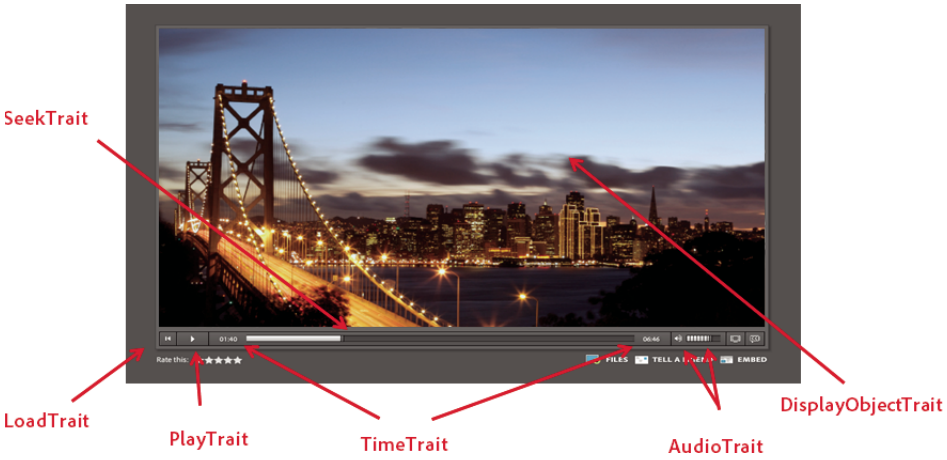
Different media elements combine different sets of traits, depending on the characteristics of the media they represent. For example, an audio element may have the `audio`, `load`, `play`, and `time` traits. You can use traits to build custom `MediaElement` subclasses.

The total set of traits in the framework is fixed, as they form the core vocabulary of the OSMF system. Creating custom individual traits is not supported. The following table lists a few basic traits and some of the media types to which they apply. For a complete list of traits, see the ActionScript documentation for org.osmf.traits.MediaTraitType.

MediaElement	PlayTrait?	DisplayObjectTrait?	TimeTrait?	AudioTrait?
VideoElement	Yes	Yes	Yes	Yes
AudioElement	Yes	No	Yes	Yes
ImageElement	No	Yes	No	No

`MediaPlayer` handles the traits of any `MediaElement` subclass. For example, `MediaPlayer` inspects the traits of a media element before loading it, in order to properly set up the player.

Trait values are subject to change, and traits dispatch events to signal these changes. For example, if the user changes the volume level, a `volumeChange` event is dispatched, because the audio trait's `volume` property has changed. `MediaPlayer` constantly monitors the availability and state of the media element's traits and responds to trait change events.



Visualizing traits

Traits typically expose getters without setters, except for traits that require externally settable properties, such as the `volume` or `pan` properties of an `audio` trait. You find out whether a media element has a trait of a given type by calling the `MediaElement.hasTrait()` method. For example, if `hasTrait(MediaTraitType.AUDIO)` returns `true`, you know that the element has an `audio` trait. You use `MediaElement.getTrait(MediaTraitType.AUDIO)` to get an object of this type.

External HTML control with JavaScript

Say that a web page contains both a video in an OSMF player and an HTML banner ad at the top of the page. When the user clicks the banner ad, the banner ad grows to cover the entire page, and the video in the OSMF player pauses. When the user dismisses the ad, the video resumes playing. Such functionality can be implemented using the `HTMLMediaContainer` and `HTMLElement` classes.

`HTMLMediaContainer` uses the Flash external interface feature to expose the container's child media elements to JavaScript. On being initialized, the `HTMLMediaContainer` object inserts JavaScript into the hosting HTML document, using `flash.external.ExternalInterface`. The added JavaScript allows elements in JavaScript to be recognized as `MediaElement` objects by the media player. Next, `HTMLMediaContainer` invokes a callback method that the HTML client can use to do further setup work.

`HTMLElement` is a media element that represents a piece of media external to the Flash SWF and within an HTML region. It serves as a bridge between the OSMF APIs for controlling media, and a corresponding (external) JavaScript implementation. The `HTMLElement` class is used to address specific portions of the hosting HTML page from the media player.

For example, `HTMLElement` can be used to wrap an HTML banner ad:

```
var banner1:HTMLElement = new HTMLElement();
banner1.resource = new URLResource("http://www.iab.net/media/image/468x60.gif");
```

`HTMLMediaContainer` invokes a callback method in JavaScript when it gets an `HTMLElement` object added (which in turn happens when `HTMLElement` is added to the `HTMLMediaContainer` instance). The callback invoked on an element being added to `HTMLMediaContainer` includes a reference to a `MediaElement` JavaScript object.

This JavaScript object holds `boolean` fields that represent whether the element is playable, temporal, and audible. These switches can be set from JavaScript at will. On doing so, for OSMF, the element appears to respectively carry the `PlayTrait`, `TimeTrait`, and `AudioTrait` traits.

All `HTMLElement` objects also have the trait `LoadTrait`. On the element receiving a 'load' or 'unload' instruction, it forwards the command to JavaScript. The JavaScript `MediaElement` allows for the loaded state to be set. When no 'load' or 'unload' methods have been set for the element on the JavaScript side, the loaded state is governed by the Flash side.

When a trait is switched on from JavaScript this results in its corresponding trait being added to the media element on the OSMF side. On the JavaScript side, an API allowing the trait to operate is made available.

The following brief sample shows one way to handle this interaction. For an extended ActionScript and HTML sample, see HTMLMediaContainerSample.

```
package
{
    import flash.display.Sprite;
    import org.osmf.elements.HTMLElement;
    import org.osmf.media.URLResource;
    public class HTMLMediaContainerExample extends Sprite
    {
        public function HTMLMediaContainerExample()
        {
            super();

            // This invokes a JavaScript callback (onHTMLMediaContainerConstructed)
            // that adds listeners to new elements being added to the container.
            var container:HTMLMediaContainer = new HTMLMediaContainer();

            var element:HTMLElement = new HTMLElement();
            element.resource = new URLResource("http://example.com/asset/handle.pgn");

            // This invokes a JavaScript callback (container.onElementAdd) that
            // allows JavaScript to process the passed URL, and to communicate back
            // to the framework what traits the element supports:
            container.addMediaElement(element);
        }
    }
}
```

Plug-ins

A modern media player does much more than play media. It can also use a content delivery network (CDN), present advertising, capture user events to report to an analytics server, and so on. But the media player does not usually handle this work by itself. This additional functionality is typically provided by third-party software known as "plug-ins."

A plug-in is code that you invite to work with your player. When you load a plug-in, you give it permission to provide additional functionality for your player. Plug-ins are not given unlimited access to your media player. OSMF acts as a broker between your media player and the plug-in. This approach ensures that communication between the media player and the plug-in is standardized, making it simple to add, update, or switch plug-ins.

Plug-ins can define additional behavior for media, a new type of media, or both. Plug-ins provide the following benefits:

- Enable additional or customized functionality without requiring changes to the underlying media player code.
- Simplify integration with third-party solutions and services. Examples include advertising, reporting, or content delivery.
- Allow implementation of complementary social media features such as rating and sharing.
- Can be static or dynamic. Static plug-ins are incorporated into the media player at compile time. Dynamic plug-ins are loaded by the media player at runtime from a SWF file.

To see some examples of plug-ins, check out the OSMF plug-in partner page.

Using plug-ins

To use plug-ins, your media player must be written using `DefaultMediaFactory` or another `MediaFactory` subclass, rather than instantiating media elements directly. If there are plug-ins available to your player, OSMF uses `MediaFactory` to interact with them. `MediaFactory` finds the appropriate plug-in for the media resource type you provide. Then, in conjunction with the plug-in, `MediaFactory` returns a plug-in-enabled `MediaElement` to your player.

On the plug-in side, each plug-in is required to provide a media description of type `MediaFactoryItem`. The `MediaFactoryItem` object has a `canHandleResourceFunction` property that identifies the resource types that the plug-in can handle.

Static plug-ins

To begin using a static plug-in, download the posted plug-in source or SWC library. Add the result to your media player project. The advantage of using static plug-ins is that there is no load time for the plug-in, since it's compiled directly into the application. After you load a plug-in, there is no further direct interaction between your player code and the plug-in. You provide a media resource to the factory, and the plug-in alters the resulting `MediaElement` based on its capabilities.

1. Link the library or source of the plug-in.
2. Create a `PluginInfoResource` object from an instance of the `PluginInfo` subclass:

```
var myPlugin:PluginInfoResource = new PluginInfoResource(new MyPluginInfo);
```

Dynamic plug-ins

With a dynamic plug-in, you can use the plug-in from its original server location, or you can download and post the plug-in on a server you control. If you use the original location, then when the developer posts updates to the plug-in your player is automatically updated, too. Either way, with dynamic plug-ins you do not need to recompile the player application to include updates.

When it loads a plug-in dynamically, OSMF verifies that the plug-in was compiled with a version of OSMF that the player supports. If there is a version mis-match between the player and the plug-in, the plug-in loading fails. Because the player receives only a general plug-in load error, developers may want to check the version of the plug-in to find out whether this was the cause. The simplest way to check plug-in version compatibility is to call the `PluginInfo.isFrameworkVersionSupported` method in `org.osmf.media`.

The basic steps to load a dynamic plug-in are as follows:

1. Create a `URLResource` object that points to the plug-in SWF file location.
2. Listen for the `MediaFactoryEventPLUGIN_LOAD` and/or `PLUGIN_LOAD_ERROR` events.
3. Perform the actual loading.

The following code snippet shows the basic flow of these calls:

```
var factory:MediaFactory = new DefaultMediaFactory();
var myPluginURL:URLResource = new URLResource(PLUGIN_URL_ONMYSERVER);
factory.addEventListener(MediaFactoryEvent.PLUGIN_LOAD, onPluginLoaded);
factory.addEventListener(MediaFactoryEvent.PLUGIN_LOAD_ERROR, onPluginLoadError);
factory.loadPlugin(myPluginURL);
```

Writing a control bar plug-in

"Interaction" on page 11 mentions that one way to implement controls for your player is as a plug-in. This section provides an example of one such plug-in. It is not intended to be a complete description of writing plug-ins in general. For that information, see the OSMF Plug-In Developer's Guide.

In the case of this plug-in, the interface object is modeled as a type of media element. For example, you can think of an overlay play button as a piece of media that "plays" in parallel with a video that it controls. This example implements a plug-in that defines a `MediaElement`-based `ControlBarElement` object. The `ControlBarElement` object displays a control bar. The plug-in also contains a reference to the video element it controls.

ControlBarPlugin

The control bar plug-in is implemented as two classes: `ControlBarPlugin` and `ControlBarElement`.

Note: *The source for this sample is located at the OSMF open source repository.*

The first class, `ControlBarPlugin`, extends the `Sprite` class and is the plug-in SWF document class. It defines the mandatory `pluginInfo` getter method that the framework uses to obtain a `PluginInfo` reference.

It also defines two callback methods. The first, `canHandleResourceCallback`, is invoked when the framework determines whether a plug-in can construct a media element for a given resource type. The second, `mediaElementCreationCallback`, is invoked when the framework is requesting the plug-in to provide a media element instance. The latter returns an object that is defined in the plug-in's other class, "`ControlBarElement`" on page 40.

```
package
{
    import __AS3__.vec.Vector;

    import flash.display.Sprite;
    import flash.system.Security;

    import org.osmf.media.MediaElement;
    import org.osmf.media.MediaFactoryItem;
    import org.osmf.media.MediaResourceBase;
    import org.osmf.media.PluginInfo;
    import org.osmf.metadata.Metadata;

    public class ControlBarPlugin extends Sprite
    {
        /**
         * Constructor
         */
        public function ControlBarPlugin()
        {
            // Allow any SWF that loads this SWF to access objects and
            // variables in this SWF.
            Security.allowDomain("*");

            super();
        }

        /**
         * Gives the player the PluginInfo.
         */
        public function get pluginInfo():PluginInfo
```

```
        {
            if (_pluginInfo == null)
            {
                var item:MediaFactoryItem
                    = new MediaFactoryItem
                        ( ID
                        , canHandleResourceCallback
                        , mediaElementCreationCallback
                        );

                var items:Vector.<MediaFactoryItem> = new Vector.<MediaFactoryItem>();
                items.push(item);

                _pluginInfo = new PluginInfo(items, mediaElementCreationNotificationCallback);
            }

            return _pluginInfo;
        }

        // Internals
        //

        public static const ID:String = "com.example.samples.controlbar";
        public static const NS_CONTROL_BAR_SETTINGS:String =
"http://www.osmf.org/samples/controlbar/settings";
        public static const NS_CONTROL_BAR_TARGET:String =
"http://www.osmf.org/samples/controlbar/target";

        private var _pluginInfo:PluginInfo;
        private var controlBarElement:ControlBarElement;
        private var targetElement:MediaElement;

        private function canHandleResourceCallback(resource:MediaResourceBase):Boolean
        {
            var result:Boolean;

            if (resource != null)
            {
                var settings:Metadata
                    = resource.getMetadataValue(NS_CONTROL_BAR_SETTINGS) as Metadata;

                result = settings != null;
            }

            return result;
        }

        private function mediaElementCreationCallback():MediaElement
        {
            controlBarElement = new ControlBarElement();

            updateControls();

            return controlBarElement;
        }
```

```
        private function mediaElementCreationNotificationCallback(target:MediaElement):void
        {
            // If the control bar has been created,
            // notify it about the just-created element.
            targetElement = target;

            updateControls();
        }

        private function updateControls():void
        {
            if (controlBarElement != null && targetElement != null && controlBarElement !=
targetElement)
            {
                controlBarElement.addReference(targetElement);
            }
        }
    }
}
```

ControlBarElement

The `ControlBarElement` class defines the control bar object, based on `MediaElement`. It exposes the control bar's `Sprite` by adding a `DisplayObjectTrait` to the media element.

```
package
{
    import org.osmf.chrome.assets.AssetsManager;
    import org.osmf.chrome.configuration.LayoutAttributesParser;
    import org.osmf.chrome.configuration.WidgetsParser;
    import org.osmf.chrome.widgets.Widget;
    import org.osmf.layout.LayoutMetadata;
    import org.osmf.media.MediaElement;
    import org.osmf.media.MediaResourceBase;
    import org.osmf.metadata.Metadata;
    import org.osmf.traits.DisplayObjectTrait;
    import org.osmf.traits.MediaTraitType;

    public class ControlBarElement extends MediaElement
    {
        // Embedded assets (see configuration.xml for their assignments):
        //

        [Embed(source="../assets/configuration.xml", mimeType="application/octet-stream")]
        private static const CONFIGURATION_XML:Class;

        [Embed(source="../assets/Standard0755.swf#Standard0755")]
        private static const DEFAULT_FONT:Class;

        [Embed(source="../assets/backDrop.png")]
        private static const BACKDROP:Class;

        [Embed(source="../assets/pause_disabled.png")]
        private static const PAUSE_DISABLED:Class;
        [Embed(source="../assets/pause_up.png")]
        private static const PAUSE_UP:Class;
```

```
        [Embed(source="../assets/pause_down.png")]
        private static const PAUSE_DOWN:Class;

        [Embed(source="../assets/stop_disabled.png")]
        private static const STOP_DISABLED:Class;
        [Embed(source="../assets/stop_up.png")]
        private static const STOP_UP:Class;
        [Embed(source="../assets/stop_down.png")]
        private static const STOP_DOWN:Class;

        [Embed(source="../assets/play_disabled.png")]
        private static const PLAY_DISABLED:Class;
        [Embed(source="../assets/play_up.png")]
        private static const PLAY_UP:Class;
        [Embed(source="../assets/play_down.png")]
        private static const PLAY_DOWN:Class;

        [Embed(source="../assets/scrubber_disabled.png")]
        private static const SCRUBBER_DISABLED:Class;
        [Embed(source="../assets/scrubber_up.png")]
        private static const SCRUBBER_UP:Class;
        [Embed(source="../assets/scrubber_down.png")]
        private static const SCRUBBER_DOWN:Class;
        [Embed(source="../assets/scrubBarTrack.png")]
        private static const SCRUB_BAR_TRACK:Class;

        // Public interface
        //

        public function addReference(target:MediaElement):void
        {
            if (this.target == null)
            {
                this.target = target;

                processTarget();
            }
        }

        private function processTarget():void
        {
            if (target != null && settings != null)
            {
                // We use the NS_CONTROL_BAR_TARGET namespaced metadata in order
                // to find out if the instantiated element is the element that our
                // control bar should control:
                var targetMetadata:Metadata =
target.getMetadata(ControlBarPlugin.NS_CONTROL_BAR_TARGET);
                if (targetMetadata)
                {
                    if  (   targetMetadata.getValue(ID) != null
                        &&  targetMetadata.getValue(ID) == settings.getValue(ID)
                        )
                    {
                        controlBar.media = target;
                    }
                }
```

```
            }
        }

        // Overrides
        //

        override public function set resource(value:MediaResourceBase):void
        {
            // Right after the media factory has instantiated us, it will set the
            // resource that it used to do so. We look the NS_CONTROL_BAR_SETTINGS
            // namespaced metadata, and retain it as our settings record
            // (containing only one field: "ID" that tells us the ID of the media
            // element that we should be controlling):
            if (value != null)
            {
                settings
                    = value.getMetadataValue(ControlBarPlugin.NS_CONTROL_BAR_SETTINGS) as
Metadata;

                processTarget();
            }

            super.resource = value;
        }

        override protected function setupTraits():void
        {
            // Set up a control bar using the ChromeLibrary:
            setupControlBar();

            // Use the control bar's layout metadata as the element's layout metadata:
            var layoutMetadata:LayoutMetadata = new LayoutMetadata();
            LayoutAttributesParser.parse(controlBar.configuration, layoutMetadata);
            addMetadata(LayoutMetadata.LAYOUT_NAMESPACE, layoutMetadata);

            // Signal that this media element is viewable: create a DisplayObjectTrait.
            // Assign controlBar (which is a Sprite) to be our view's displayObject.
            // Additionally, use its current width and height for the trait's mediaWidth
            // and mediaHeight properties:
            viewable = new DisplayObjectTrait(controlBar, controlBar.measuredWidth,
controlBar.measuredHeight);
            // Add the trait:
            addTrait(MediaTraitType.DISPLAY_OBJECT, viewable);

            controlBar.measure();

            super.setupTraits();
        }

        // Internals
        //

        private function setupControlBar():void
        {
            try
            {
                var configuration:XML = XML(new CONFIGURATION_XML());
```

```
                var assetsManager:AssetsManager = new AssetsManager();
                assetsManager.addConfigurationAssets(configuration);
                assetsManager.load();

                var widgetsParser:WidgetsParser = new WidgetsParser()
                widgetsParser.parse(configuration.widgets.*, assetsManager);

                controlBar = widgetsParser.getWidget("controlBar");
            }
            catch (error:Error)
            {
                trace("WARNING: failed setting up control bar:", error.message);
            }
        }

        private var settings:Metadata;

        private var target:MediaElement;
        private var controlBar:Widget;
        private var viewable:DisplayObjectTrait;

        /* static */

        private static const ID:String = "ID";
    }
}
```

ControlBarPluginSample

The OSMF player sample application consists of two classes. The first contains much of the standard code that is required in setting up any OSMF player application, and is omitted here. It's available from the OSMF public code repository.

The second class, called `ControlBarPluginSample`, is the main tester application and is located with the OSMF sample plugins.

Using `OSMFConfiguration`, the `MediaFactory.loadPlugin` method loads ControlBarPlugin.swf. Meanwhile, a parallel media element is set up that acts as the root of the aggregate media that is played back. The first child is then added, a video that plays an OSMF logo animation. Once the plug-in is loaded, the second element is added: a control bar media element, instantiated by the code on ControlBarPlugin.swf. This application uses `ControlBarPlugin` as a static plug-in for ease of debugging.

When the plug-in loaded, it registers itself with the media factory. When the factory is asked to create an element for the given resource, it goes over all types and plug-ins that registered with it. Ultimately, this results in the `canHandleResourceCallback` method in `ControlBarPlugin.as` being invoked. The implementation of that method returns `true` if it finds the resource to have metadata that goes by the namespace defined in `NS_CONTROL_BAR_SETTINGS`. When this method returns `true`, the framework asks the plug-in to instantiate a media element.

When the media element is instantiated, the plug-in is informed through the `mediaElementCreationNotificationFunction`. The control bar plug-in's implementation of this function calls `addReference` to associate the created element with the control bar. But it only does so if the element has metadata with a namespace URL that matches the namespace defined in `NS_CONTROL_BAR_TARGET`.

The last step in matching the control bar to the element it controls takes place inside the control bar element's `addReference` method. If there is an element with the same ID as the element's own ID from `NS_CONTROL_BAR_SETTINGS`, the element is assigned as the control bar's target element.

```
package
{
    import flash.display.Sprite;

    import org.osmf.elements.ParallelElement;
    import org.osmf.events.MediaFactoryEvent;
    import org.osmf.layout.HorizontalAlign;
    import org.osmf.layout.LayoutMetadata;
    import org.osmf.layout.VerticalAlign;
    import org.osmf.media.*;
    import org.osmf.metadata.Metadata;

    [SWF(width="640", height="360", backgroundColor="0x000000",frameRate="25")]
    public class ControlBarPluginSample extends Sprite
    {
        public function ControlBarPluginSample()
        {
            // Construct an OSMFConfiguration helper class:
            osmf = new OSMFConfiguration();

            // Construct the main element to play back. This is a
            // parallel element, that holds the main content to
            // play back, and the control bar (from a plug-in) as its
            // children:
            osmf.mediaElement = constructRootElement();
            osmf.view = this;

            // Add event listeners to the plug-in manager to receive
            // a heads-up when the control bar plug-in finishes loading:
            osmf.factory.addEventListener(MediaFactoryEvent.PLUGIN_LOAD, onPluginLoaded);
            osmf.factory.addEventListener(MediaFactoryEvent.PLUGIN_LOAD_ERROR,
onPluginLoadError);

            // Ask the plug-in manager to load the control bar plug-in:
            osmf.factory.loadPlugin(pluginResource);
        }

        // Internals
        //

        private var osmf:OSMFConfiguration;
        private var rootElement:ParallelElement;

        private function onPluginLoaded(event:MediaFactoryEvent):void
        {
            // The plugin loaded successfully. We can now construct a control
            // bar media element, and add it as a child to the root parallel element:
            rootElement.addChild(constructControlBarElement());
        }

        private function onPluginLoadError(event:MediaFactoryEvent):void
        {
            trace("ERROR: the control bar plugin failed to load.");
```

```
        }

        private function constructRootElement():MediaElement
        {
            // Construct a parallel media element to hold the main content,
            // and later on, the control bar.
            rootElement = new ParallelElement();
            rootElement.addChild(constructVideoElement());

            // Use the layout api to set the parallel element's width and
            // height. Make it the same size as the stage currently is:
            var rootElementLayout:LayoutMetadata = new LayoutMetadata();
            rootElement.addMetadata(LayoutMetadata.LAYOUT_NAMESPACE, rootElementLayout);
            rootElementLayout.width = stage.stageWidth;
            rootElementLayout.height = stage.stageHeight;

            return rootElement;
        }

        private function constructVideoElement():MediaElement
        {
            // Construct a metadata object that we can append to the video's collection
            // of metadata. The control bar plug-in uses the metadata to identify
            // the video element as its target:
            var controlBarTarget:Metadata = new Metadata();
            controlBarTarget.addValue(ID, "mainContent");

            // Construct a video element:
            var video:MediaElement = osmf.factory.createMediaElement(new
URLResource(VIDEO_URL));

            // Add the metadata to the video's metadata:
            video.addMetadata(ControlBarPlugin.NS_CONTROL_BAR_TARGET, controlBarTarget);

            return video;
        }

        private function constructControlBarElement():MediaElement
        {
            // Construct a metadata object to send to the media factory on
            // requesting a control bar element to be instantiated. The factory
            // uses it to parameterize the element. Specifically, the ID field
            // tells the plug-in what the ID of the content it should control is:
            var controlBarSettings:Metadata = new Metadata();
            controlBarSettings.addValue(ID, "mainContent");

            // Add the metadata to an otherwise empty media resource object:
            var resource:MediaResourceBase = new MediaResourceBase();
            resource.addMetadataValue(ControlBarPlugin.NS_CONTROL_BAR_SETTINGS,
controlBarSettings);

            // Request the media factory to construct a control bar element. The
            // factory knows a control bar element is requested by inspecting
            // the resource's metadata (and encountering a metadata object of namespace
            // NS_CONTROL_BAR_SETTINGS there):
            var controlBar:MediaElement = osmf.factory.createMediaElement(resource);
```

```
            // Set some layout properties on the control bar. Specifically, have it
            // appear at the bottom of the parallel element, horizontally centered:
            var layout:LayoutMetadata = controlBar.getMetadata(LayoutMetadata.LAYOUT_NAMESPACE)
as LayoutMetadata;
            if (layout == null)
            {
                layout = new LayoutMetadata();
                controlBar.addMetadata(LayoutMetadata.LAYOUT_NAMESPACE, layout);
            }
            layout.verticalAlign = VerticalAlign.BOTTOM;
            layout.horizontalAlign = HorizontalAlign.CENTER;

            // Make sure that the element shows over the video: elements with a
            // higher-order number set are placed higher in the display list:
            layout.index = 1;

            return controlBar;
        }

        /* static */

        private static const VIDEO_URL:String
            = "http://mediapm.edgesuite.net/osmf/content/test/logo_animated.flv";

        private static var ID:String = "ID";

        // Comment out to load the plug-in for a SWF (instead of using static linking, for testing):
        //private static const pluginResource:URLResource = new
URLResource("http://mediapm.edgesuite.net/osmf/swf/ControlBarPlugin.swf");

        private static const pluginResource:PluginInfoResource = new PluginInfoResource(new
ControlBarPlugin().pluginInfo);
    }
}
```