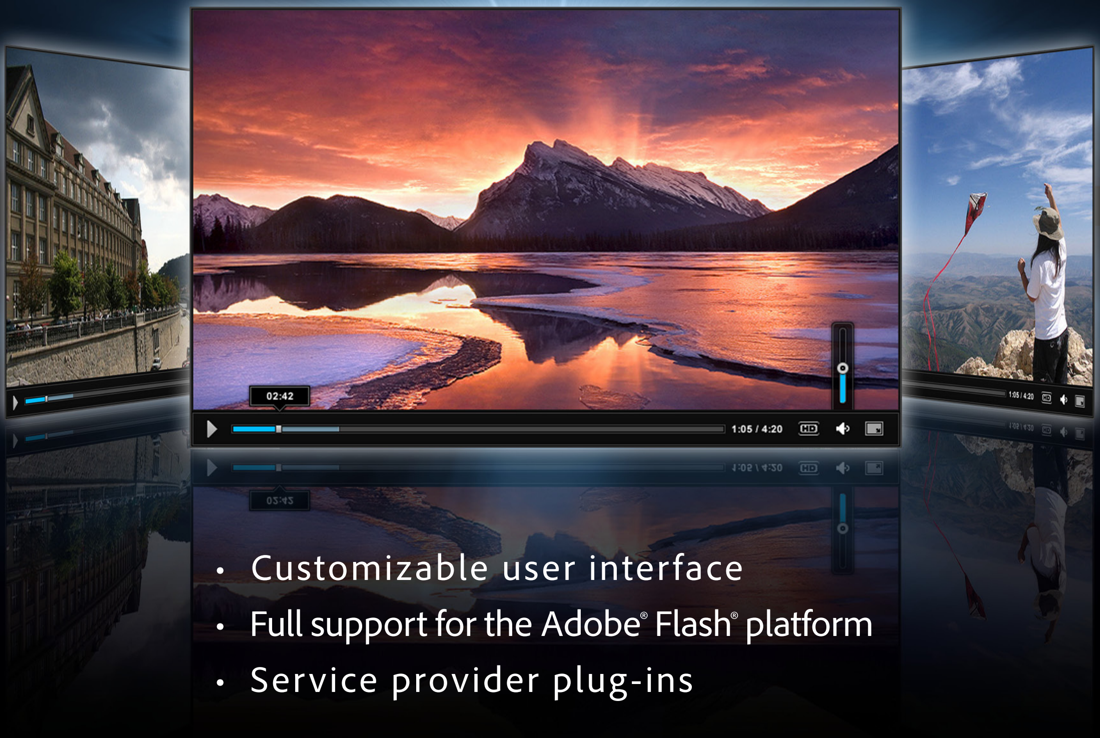
This DZone Refcard is brought to you by...

Adobe

open source
media framework



- Customizable user interface
- Full support for the Adobe® Flash® platform
- Service provider plug-ins

Simplified media player development

www.osmf.org

© 2010 Adobe Systems Incorporated. All rights reserved. Adobe, the Adobe logo, and Flash are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. All other trademarks are the property of their respective owners.

CONTENTS INCLUDE:

Open Source Media Framework: Building Simple Custom Video Players

By R Blank

ABOUT OSMF

The Open Source Media Framework (OSMF) is an Adobe-led, open-source ActionScript 3 coding framework for loading, playing and displaying media. The website is http://osmf.org.

Just as Adobe created the Flex Framework to standardize and expedite the creation of applications on the Flash Platform, OSMF is a means to standardize media handling in your Flash Platform experiences.

Why are the loading and playback of sounds, videos and images each handled so differently in AS3? Why is building a streaming video player so much more difficult than building a progressive video player? Why is controlling volume so non-intuitive?

With OSMF, all those annoying questions disappear, as we now have a simple, clean and standard way of working with all media inside of Flash.

Getting OSMF

OSMF is a framework—which means that you need to install the OSMF library if you wish to use it in your code.

Download the current version of OSMF (v1.5 at the time of authoring) from http://opensource.adobe.com/wiki/display/osmf/Downloads. The Source .zip from this page includes both the ActionScript source for OSMF, as well as the compiled SWC component for use in development. You may also download the ASDoc documentation for the OSMF classes at the same URL.

Installing OSMF

If you are using Flash Builder, add OSMF.swc to the libs folder in your project, or to your standard library paths. Please note that Flash Builder 4 ships with a version of OSMF.swc that you will likely want to remove prior to installing the new OSMF.swc into your library paths. To remove the default version of OSMF from your project, select 'Project > Properties' from the menu. Select 'Flex Build Path' from the menu on the left side of the dialog box, and then click on the the 'Library path' tab. Expand the tree branch for the version of the Flex Framework you are using, select the OSMF.swc and then click on 'Remove'. Click 'OK'.

If you are using Flash CS4 or Flash Professional CS5, then you will want to copy OSMF.swc to:

On Windows: \Program Files\Adobe\Adobe Flash CS[#]\Common\Configuration\ActionScript 3.0\libs
On Mac: /Applications/Adobe Flash CS[#]/Common/Configuration/ActionScript 3.0/libs

Testing OSMF Installation

Write these two lines of ActionScript and try to compile your project—if this code works, you have successfully installed OSMF:

```
import org.osmf.media.MediaPlayer;
var mediaPlayer : MediaPlayer = new MediaPlayer();
```

CAPABILITIES

Supported Media Formats

OSMF supports any type of media that can be loaded by Flash, including (streaming audio) mp3, AAC, Speex, and Nellymoser; (streaming video) FLV, F4V, MP4, MPEG-4: MP4, M4V, F4V, 3GPP; (audio) mp3; (video) FLV, F4V, MP4, MP4V-ES, M4V, 3GPP, 3GPP2, QuickTime; (images) PNG, GIF, or JPG; and SWF files.

Support for Standards

OSMF also comes packaged with support for media standards including Video Ad Serving Template (VAST), Media Abstract Sequencing Template (MAST), Media RSS (MRSS), Distribution Format Exchange Profile (DFXP), and Synchronized Multimedia Integration Language (SMIL).

Flash Player 10 or 10.1?

There are different versions of OSMF depending on whether you are publishing to Flash Player 10 or 10.1 (OSMF is unsupported in Flash Player 9 and earlier), and both are included in the standard OSMF download. Which do you want to use?

If you want to utilize HTTP Streaming, DRM, and Multicast, then you must use OSMF for FP 10.1—otherwise, the functionality of both versions is equivalent.

Plug-ins

OSMF includes a plug-in architecture, enabling you and other third party developers to create useful and reusable functionality for OSMF experiences. Due to space considerations, plug-ins are not covered in this Refcard. For more information on OSMF plug-ins, visit http://opensource.adobe.com/wiki/display/osmf/Plugins. For a list of available OSMF plug-ins (for example, to facilitate playing media from Akamai's CDN, or to implement Omniture tracking), visit http://osmf.org/partner.php.

Adobe
open source media framework
Customizable user interface
Full support for the Adobe Flash platform
Service provider plug-ins
Simplified media player development
www.osmf.org

STROBE MEDIA PLAYBACK AND FLASH MEDIA PLAYBACK

OSMF is a coding framework and includes no GUI or graphical options at all—OSMF is just logic. There are no OSMF-based components in Flash Professional, or the Flex Framework. If you want a quick video player, or a sample OSMF video player to use as a starting point, you can use Strobe Media Playback (SMP) and Flash Media Playback (FMP).

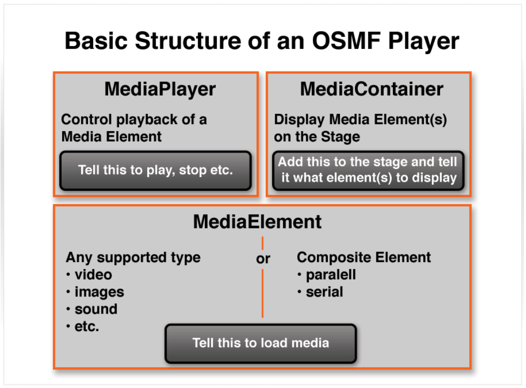
SMP is an open-source Flash video player, built on OSMF, that includes a full GUI, with elements like a play button, a volume slider and a progress bar. You can use SMP to get a kick-start working with OSMF, and then customize and redeploy SMP for your own uses. For more information on, and to download SMP, visit http://osmf.org/strobe_mediaplayback.html.

FMP is a version of SMP hosted by Adobe. Simply visit the FMP Configurator at http://www.osmf.org/configurator/fmp/, enter the location of your video file, copy the embed code, and paste it into any web page and voila!—you have an OSMF player in seconds. For more information on FMP, visit http://www.adobe.com/products/flashmediaplayback/.

Both SMP and FMP are easily skinnable, and playback options are rapidly customized through FlashVar parameters.

THE BASIC OSMF PLAYER

At the heart of any OSMF experience are three classes: the MediaElement, the MediaPlayer, and the MediaContainer.


Basic Structure of an OSMF Player
MediaPlayer
Control playback of a Media Element
Tell this to play, stop etc.
MediaContainer
Display Media Element(s) on the Stage
Add this to the stage and tell it what element(s) to display
MediaElement
Any supported type
• video
• images
• sound
• etc.
or
Composite Element
• paralell
• serial
Tell this to load media

MediaElement

The MediaElement is used to **load** media.

You generally would not use the MediaElement directly—there are several descendants of the MediaElement, including AudioElement (to load audio) and ImageElement (to load images). In this example, we will use the VideoElement (to load video). When creating a new VideoElement, you pass in a URLResource pointing to the video you wish to play, as in:

```
var videoElement = new VideoElement(new URLResource("my.flv"));
```

MediaPlayer

The MediaPlayer is used to play media stored in MediaElement instances.

The MediaPlayer has a media property; we use this to instruct the MediaPlayer which piece of media to playback. The MediaPlayer may be reused to play multiple pieces of content.

```
var mediaPlayer : MediaPlayer = new MediaPlayer();
mediaPlayer.media=videoElement;
```

MediaContainer

The MediaContainer is used to display media stored in MediaElement instances.

```
var container : MediaContainer = new MediaContainer();
```

As the MediaContainer makes our media visible to humans, it must reside in the Display List to be seen.

```
addChild(container);
```

Finally, we add our MediaElement to our MediaContainer by calling the addMediaElement method:

```
container.addMediaElement(videoElement);
```

Imports

Even if you are accustomed to coding on the timeline in Flash Professional, you still must import OSMF classes for them to be used. You can always reference the OSMF documentation (which can be downloaded as a .zip from http://opensource.adobe.com/wiki/display/osmf/Downloads) to determine the location of the classes you need to import. For this simple OSMF player, four imports are required.

Putting it all Together

These are the four imports and six lines of code in the basic OSMF video player:

```
import org.osmf.containers.MediaContainer;
import org.osmf.elements.VideoElement;
import org.osmf.media.MediaPlayer;
import org.osmf.media.URLResource;
var container : MediaContainer = new MediaContainer();
addChild(container);
var videoElement : VideoElement =new VideoElement(new URLResource("my.flv"));
container.addMediaElement(videoElement);
var mediaPlayer : MediaPlayer = new MediaPlayer();
mediaPlayer.media=videoElement;
```

MORE USEFUL STUFF

Thus far, we've covered the basics—which are obviously an important place to start. But the real benefits of working with OSMF become clear once you start doing more—and you see how easy and standardized it all is.

Controlling Media Playback

We use the MediaPlayer to control media playback, with three methods. The default behavior of the MediaPlayer is to automatically play loaded media. To pause the media, we tell the MediaPlayer to pause:

```
mediaPlayer.pause( );
```

Similarly, to play (resume) our media:

```
mediaPlayer.play( );
```

And to jump around within our media file, we call the seek() method, passing in as a parameter the position within our media (in seconds) to which we wish to seek:

```
mediaPlayer.seek( 5 );
```

Customizing Layout

By default, a MediaContainer instance will display any visual media at its default size (100%) and position (0,0). We use the LayoutMetadata (org.osmf.layout.LayoutMetadata) class to customize the appearance of visual media, by creating a LayoutMetadata object, defining its properties, and then attaching it to our MediaElement.

First, we import the LayoutMetadata class.

```
import org.osmf.layout.LayoutMetadata ;
```

Then, we create a LayoutMetadata object:

```
var layout : LayoutMetadata = new LayoutMetadata();
```

Next, we customize the properties of our LayoutMetadata object. Four common properties to set are width, height, x and y:

```
layout.width=640;
layout.height=480;
layout.x=20;
layout.y=40;
```

Finally, we need to link our LayoutMetadata with the MediaElement instance it is intended to influence:

```
mediaElement.addMetadata(LayoutMetadata.LAYOUT_NAMESPACE, layout);
```

Then, when our mediaElement is displayed in a MediaContainer, it will have our intentional, customized layout.

Key MediaPlayer Properties

The MediaPlayer includes several useful properties, which I've included here for reference:

MediaPlayer.duration
The duration (in seconds) of the playback length of the media loaded into the MediaElement playing back in this MediaPlayer instance (read-only).

MediaPlayer.volume
The volume (from 0 to 1) of the audio track of the MediaElement currently loaded into this MediaPlayer instance. By default, this is set to 1.

MediaPlayer.autoPlay
A boolean value determining whether or not to automatically begin playback on MediaElement instances loaded into this MediaPlayer instance. By default, this is true (playback begins immediately).

MediaPlayer.loop
A boolean value determining whether or not to automatically loop the playback of any MediaElement instances loaded into this MediaPlayer instance. When set to true, upon completion of playback, the MediaPlayer instance will rewind the loaded media and begin playback again. By default, this is set to false (media does not loop).

MediaPlayer.autoRewind
A boolean value determining whether or not this MediaPlayer instance will automatically rewind-and-pause when reaching the end of playback of any MediaElement instances loaded into it. By default this is set to true (will automatically return to the beginning of the media upon completion of playback). This property is ignored if the loop property is set to true.

Media Factories

Thus far, when we wanted to create a new MediaElement instance to load a video, we created a new VideoElement. But, of course, that code will now only work to load a video. Sometimes we want to write code more generically than that. What if we want to load an image instead? Or a sound file? Or some text? Do we have to re-write our code?

Actually, no. OSMF includes Media Factories—adorable, little assembly plants that can churn out any flavor of MediaElement instances. Just tell a Media Factory to load a file from a URL, and the factory will know exactly what type of MediaElement to create.

First, we import the DefaultMediaFactory class:

```
import org.osmf.media.DefaultMediaFactory ;
```

Then, we create a DefaultMediaFactory:

```
var mediaFactory : DefaultMediaFactory = new DefaultMediaFactory();
```

Finally, we tell the mediaFactory to produce a MediaElement from a URLResource, pointing to a file:

```
var mediaElement : MediaElement = mediaFactory.createMediaElement(new
URLResource("my.flv"));
```

You will note that our MediaElement is strong-typed to MediaElement, instead of VideoElement—because we do not necessarily know at author time, what type of MediaElement will be required.

COMPOSITE ELEMENTS

CompositeElements are complex types of the MediaElement class. CompositeElements encapsulate multiple MediaElement instances in a single object.

As with MediaElements, we do not create CompositeElements directly, but instead work with descendants of the CompositeElement class.

SerialElements are used to play multiple MediaElements **sequentially** (one after the other). SerialElements can be used to emulate playlists.

ParallelElements are used to play multiple MediaElements **concurrently** (at the same time). ParallelElements can be used for any number of purposes, but could, for example, be used to combine playback of images and music (a dynamic slideshow, for example).

Creating CompositeElements

Determine whether you want a SerialElement or a ParallelElement. As you work with both types of CompositeElements identically in ActionScript, for the purposes of this example, we will work with a SerialElement to play two videos sequentially.

First, we import the SerialElement class:

```
import org.osmf.elements.SerialElement ;
```

Then, we create a new SerialElement:

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

Populating CompositeElements

To populate our CompositeElement instance, we must first have our constituent MediaElement instances (in this case, we'll define two VideoElement instances for sequential playback).

```
var mediaElement1 : MediaElement = mediaFactory.createMediaElement(new
URLResource("some.flv"));
var mediaElement2 : MediaElement = mediaFactory.createMediaElement(new
URLResource("other.flv"));
```

Once we have our MediaElement instances, we can add them to our CompositeElement, using the addChild() method (in a SerialElement, the MediaElement instances are played in the order in which they are added to the SerialElement):

```
serialElement.addChild ( mediaElement1 ) ;
serialElement.addChild ( mediaElement2 ) ;
```

Playing CompositeElements

Once our CompositeElement is populated, it is ready for playback. We play CompositeElements **exactly the same way** as simple MediaElement instances. We simply set the media property of our MediaPlayer to point to our CompositeElement:

```
mediaPlayer.media = serialElement;
```

Displaying CompositeElements

We add a CompositeElement instance to the display the same way we reveal a MediaElement—by attaching it to a MediaContainer:

```
mediaContainer.addMediaElement(serialElement);
```

We do NOT customize the layout of CompositeElement instances. Recall, CompositeElements can support multiple different pieces of media, of different types and characteristics, alone or in parallel; thus you will want to control the layout of each MediaElement individually anyway.

To customize the layout of media stored in CompositeElements, we work with LayoutMetadata objects, and apply them to the individual MediaElement instances (as we learned above).

```
layout : LayoutMetadata = new LayoutMetadata();
layout.width = 320;
layout.height = 240;
layout.x = 320;
layout.y = 240;
mediaElement1.addMetadata(LayoutMetadata.LAYOUT_NAMESPACE, layout);
mediaElement2.addMetadata(LayoutMetadata.LAYOUT_NAMESPACE, layout);
```

STREAMING

Streaming media, unlike progressively loaded media, is delivered (or 'streamed') in a series of packets that are continually delivered to the user during playback. For this reason, unlike progressively delivered media, media files that are streamed are never actually stored on viewer computers. Streaming is a very effective way to deliver large audio and video files and, in particular, streaming facilitates rapid and immediate seeking (with progressive media, you can only seek to those portions of the media that have already downloaded).

If you are interested in working more with streaming, and you are on a Windows or Linux machine, you can install a local developer version of Flash Media Server (which you may download from http://adobe.com/products/flashmediaserver/). If you are working on Mac, or you wish to develop your FMS work on a live server, you can lease FMS inexpensively from http://Influxis.com.

URL Formatting for Streaming

Before we get into the code for streaming, if you aren't used to working with Flash Media Server, you may not be used to the rather awkward logic FMS requires with respect to streaming media file extensions.

If you are streaming an FLV, you strip the '.flv' when calling the file, as in:

```
rtmp://myFMS.com/appDirectory/my
```

Basic RTMP Streaming

OSMF makes streaming incredibly easy. If you return to our 'Basic OSMF Example, we can change the URL in this line of code:

```
var videoElement : VideoElement = new VideoElement(new URLResource("my.flv"));
```

to (assuming you had a Flash Media Server setup at myFMS.com):

```
var videoElement : VideoElement = new VideoElement(new URLResource("rtmp://myFMS.
com/appDirectory/my"));
```

And voila! Our player is now a streaming video player. Seriously, that's it. But it gets better.

HTTP Streaming

The above is an example of RTMP streaming—or streaming with the real-time media protocol (used by Adobe's Flash Media Server). With Flash Player 10.1, you can now stream your media without an RTMP server, straight from your web server, utilizing HTTP Streaming (assuming your web server is configured to support it, which the majority are). It's just what it sounds like—streaming video without Flash Media Server.

Can you guess how difficult the code is? Well, it's the same code as we've already worked with, but instead of pointing to an FLV, F4V, MP4 or MP3, we point to an F4M file, or a Flash Media Manifest file, which is an XML file containing information about a Flash media asset. For example:

```
var videoElement : VideoElement = new VideoElement(new URLResource("http://my.com/
my.f4m"));
```

For more information on F4M, reference the specification at http://opensource.adobe.com/wiki/display/osmf/Flash+Media+Manifest+File+Format+Specification.

Dynamic Streaming

Sometimes you want to prepare multiple versions of the same video, at different quality bitrates, so that viewers can enjoy the highest quality video possible, given their bandwidth.
And, because a viewer's bandwidth can change over time, you will want to query the active bandwidth repeatedly, and
change to a higher or lower quality video if the viewer's bandwidth has shifted.

This process is called Dynamic Streaming—delivering the highest quality video possible, at all times during the viewing experience, from a collection of videos encoded at different bitrates.

Though not quite as simple as basic streaming (after all, there's more information we have to setup to get this working), implementing dynamic streaming with OSMF is remarkably easy.

DynamicStreamingResource

In all of the prior examples, we have utilized a URLResource to wrap the URL to our media file, when creating our VideoElement, as in:

```
var videoElement : VideoElement = new VideoElement(new URLResource("my.flv"));
```

To implement dynamic streaming, we need to use a different class—the DynamicStreamingResource. When we create a DynamicStreamingResource, instead of pointing to a media file, we point to a Flash Media Server application directory.
First, we import the DynamicStreamingResource class:

```
import org.osmf.net.DynamicStreamingResource ;
```

Then, we create a new DynamicStreamingResource:

```
var resource : DynamicStreamingResource = new DynamicStreamingResource ( "rtmp://
myFMS.com/myAppDir" ) ;
```

DynamicStreamingItem

Our resource now points to an entire directory, instead of a single file. Next, we must populate our DynamicStreamingResource with the information pointing to our actual video files.

To do so, we create a DynamicStreamingItem instance for each bitrate we intend to support. When creating a DynamicStreamingItem, we specify a filename (respecting FMS file extension rules, noted above), and a minimum supported bandwidth (in kilobits-per-second, or kbps) required to view this version of the video:

```
new DynamicStreamingItem ( "my_high" , 1500 )
```

In our code, we will store these DynamicStreamingItem instances in a vector (vectors are similar to arrays, but the values stored in the indices of a vector must conform to the same datatype—in this case, all values stored in our vector will be DynamicStreamingItem instances).

For our example, we will support three separate bitrates in our player: 400kbps (low), 800kbps (medium) and 1.5mbps (high). To start, of course, we will need to import the DynamicStreamingItem class:

```
import org.osmf.net.DynamicStreamingItem ;
```

Then, we will create our vector of DynamicStreamingItems (with a fixed length of 3, because we wish to support precisely three bitrates):

```
var vector : Vector.<DynamicStreamingItem> = new Vector.<DynamicStreamingItem> (
3 ) ;
```

Next, we will populate our vector with the three DynamicStreamingItem instances.

```
vector [ 0 ] = new DynamicStreamingItem ( “my_high” , 1500 ) ;
vector [ 1 ] = new DynamicStreamingItem ( “my_low” , 400 ) ;
vector [ 2 ] = new DynamicStreamingItem ( “my_medium” , 800 ) ;
```

Note that the ordering of DynamicStreamingItems in the vector is irrelevant.

Then, before we move on, we ensure that the streamItems property of our DynamicStreamingResource points to our vector:

```
resource.streamItems = vector ;
```

Bringing it all Together

There may be some new and long class names involved, but building a dynamic streaming player with OSMF is remarkably easy—the code is very brief (only 12 lines in this example) and very similar to the basic six lines of code we used to progressively play a single video:

```
var player : MediaPlayer = new MediaPlayer ( ) ;
var container : MediaContainer = new MediaContainer ( ) ;
addChild ( container ) ;
var resource : DynamicStreamingResource = new DynamicStreamingResource ( “rtmp://
myFMS.com/myAppDir” ) ;
vector [ 0 ] = new DynamicStreamingItem ( “my_high” , 1500 ) ;
vector [ 1 ] = new DynamicStreamingItem ( “my_low” , 400 ) ;
vector [ 2 ] = new DynamicStreamingItem ( “my_medium” , 800 ) ;
resource.streamItems = vector ;
videoElement = new VideoElement( resource ) ;
player.media = videoElement ;
container.addMediaElement ( videoElement ) ;
```

Subclipping

When you stream with an RTMP server, it is easy to play segments from entire videos—referred to as subclipping, or playing clips within clips.

We do this with the StreamingURLResource, which we can use instead of a regular URLResource. A StreamingURLResource points only to a specific portion of a streamed video, determined by in and out points (measured in seconds), set when you create the instance:

```
new StreamingURLResource( url, streamType , clipStartTime , clipEndTime );
```

There are many potential uses for subclipping, but an obvious one is the insertion of interstitial advertising. Let’s say we wish to stream a one-minute clip, and progressively deliver a video ad half way through (at 30 seconds). This will require two subclips (aka StreamingURLResources) and a regular URLResource (for the ad), connected by a SerialElement (so that they play sequentially, rather than concurrently).

To start, we import the StreamingURLResource class:

```
import org.osmf.net.StreamingURLResource ;
```

Then we can write the following code:

```
var serialElement : SerialElement = new SerialElement();
var resource1 = new StreamingURLResource( “rtmp://myFMS.com/app/vid1” , null, 0,
30 );
serialElement.addChild( new VideoElement( resource1 ) );
var resource2 : URLResource = new URLResource ( “myAd.flv” ) ;
serialElement.addChild( new VideoElement( resource2 ) );
var resource3 = new StreamingURLResource( “rtmp://myFMS.com/app/vid1”, null, 30,
60 );
serialElement.addChild( new VideoElement( resource3 ) );
var mediaPlayer : MediaPlayer = new MediaPlayer( serialElement );
var container : MediaContainer = new MediaContainer();
addChild( container );
container.addMediaElement( serialElement );
```

KEY EVENTS

A huge amount of OSMF just works. But when you start building larger and customized experiences (for example, a playlist-driven media player, with a control bar to manage playback), you will need to write a bit of your own code, to handle and respond to events coming from OSMF. For example, you may wish to know when to enable a custom pause button, or to toggle its state to look like a play button; or perhaps you wish to build a custom progress bar, and need to track the position of the media.

OSMF is a rich framework, there is a lot you can do with it, and there are many events within OSMF you can listen to and exploit. However, to get started building custom controls, there are a few events—all coming from the MediaPlayer—that are the most important to learn about and work with.

Remember: MediaPlayer instances can be re-used to playback multiple MediaElement and CompositeElement instances, of multiple types—so these events can be dispatched during playback of a MediaElement, or in between playback of different MediaElement instances.

MediaErrorEvent

The MediaErrorEvent (org.osmf.events.MediaErrorEvent) is dispatched on a MediaElement when there is an error loading the specified media (such as if the specified video file does not exist). The MediaErrorEvent can also be heard on the MediaPlayer to which the MediaElement is associated—so you may listen for this event on either type of object.

The code to listen for, and trace out the details of, a MediaErrorEvent is:

```
mediaPlayer.addEventListener ( MediaErrorEvent.MEDIA_ERROR , _onMediaError ) ;
function _onMediaError ( evt : MediaErrorEvent ) : void
{
            trace ( “_onMediaError () , evt.error : “ + evt.error ) ;
}
```

TimeEvent

The TimeEvent (org.osmf.events.TimeEvent) is dispatched on MediaPlayer instances on three occasions:

TimeEvent.CURRENT_TIME_CHANGE is dispatched when the time property of the MediaPlayer has changed (e.g., your video has advanced, and you want to update a progress bar).

TimeEvent.COMPLETE is dispatched when the playback of the MediaElement currently playing back in the MediaPlayer is complete (e.g., when your video has ended).

TimeEvent.DURATION_CHANGE is dispatched when the duration of the MediaElement currently playing back in the MediaPlayer has changed, for example, when you swap out associated MediaElement instances (when you change which MediaElement instance is stored in the media property), or advance MediaElements in a SerialElement.

The code to listen for, and trace out the details of, a TimeEvent is:

```
mediaPlayer.addEventListener ( TimeEvent.CURRENT_TIME_CHANGE , onTimeEvent ) ;
mediaPlayer.addEventListener ( TimeEvent.COMPLETE , onTimeEvent ) ;
mediaPlayer.addEventListener ( TimeEvent.DURATION_CHANGE , onTimeEvent ) ;
function onTimeEvent ( evt : TimeEvent ) : void
{
              trace ( “onTimeEvent () , evt.name: “ + evt.name + “ , evt.time : “ +
evt.time ) ;
}
```

MediaPlayerCapabilityChangeEvent

The MediaPlayerCapabilityChangeEvent (org.osmf.events.MediaPlayerCapabilityChangeEvent) is dispatched on MediaPlayer instances when the capabilities of the instance have changed.

MediaPlayerCapabilityChangeEvent.CAN_PLAY_CHANGE is dispatched on MediaPlayer instances when the playable state of the media loaded into the associated MediaElement changes. The value is either true (e.g., when your video has loaded and can begin playback), or false (e.g., when your video can no longer be played).

MediaPlayerCapabilityChangeEvent.CAN_SEEK_CHANGE is dispatched on MediaPlayer instances when the seekable state of the media loaded into the associated MediaElement changes. The value is either true (the media is now seekable) or false (the media is not seekable).

```
mediaPlayer.addEventListener ( MediaPlayerCapabilityChangeEvent.CAN_LOAD_CHANGE ,
onCapabilityChange ) ;
mediaPlayer.addEventListener ( MediaPlayerCapabilityChangeEvent.CAN_PLAY_CHANGE ,
onCapabilityChange ) ;
mediaPlayer.addEventListener ( MediaPlayerCapabilityChangeEvent.CAN_SEEK_CHANGE ,
onCapabilityChange ) ;
function onCapabilityChange ( evt : MediaPlayerCapabilityChangeEvent ) : void
{
              trace ( “onCapabilityChange () , evt.enabled : “ + evt.enabled ) ;
}
```

MediaPlayerStateChangeEvent

The MediaPlayerStateChangeEvent (org.osmf.events.MediaPlayerStateChangeEvent) is very useful at informing us in changes to the playback state of the media controlled by MediaPlayer instances. To listen for this event on MediaPlayer instances, and trace out the useful information from the event, we can use the following code:

```
mediaPlayer.addEventListener ( MediaPlayerStateChangeEvent.MEDIA_PLAYER_STATE_
CHANGE , onMediaPlayerStateChange ) ;
function onMediaPlayerStateChange ( evt : MediaPlayerStateChangeEvent ) : void
{
              trace ( “onMediaPlayerStateChange () , evt.state : “ + evt.state ) ;
}
```

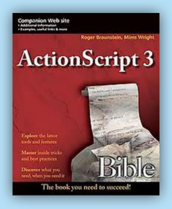
It then becomes important to know what the state property is set to on the MediaPlayerStateChangeEvent—so we know what state change led to the firing of this event. All possible state change values are available as public static constants on the MediaPlayerState class (org.osmf.media.MediaPlayerState).

If state =...	That Means...
MediaPlayerState.PLAYING	The media is playing.
MediaPlayerState.LOADING	The media is loading.
MediaPlayerState.BUFFERING	The media is buffering.
MediaPlayerState.UNINITIALIZED	The MediaPlayer is empty (there is no loaded media).
MediaPlayerState.READY	The media has completed loading and is ready for playback; or the media has completed playback, has rewound, and is ready for playback again.

ABOUT THE AUTHOR

R Blank is CTO of Almer/Blank, an Adobe Solution Partner based in Venice, California, that specializes in video and application development for the Flash platform. He is also the Training Director at the Rich Media Institute, the Adobe Authorized Training Center that he co-founded. As well, R serves on the faculty. His personal blog is RBlank.com.

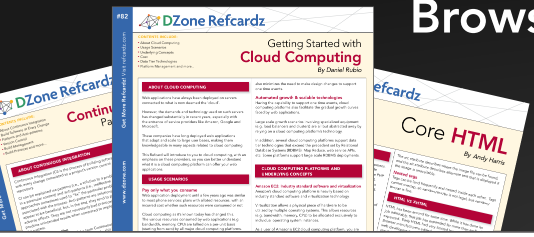
RECOMMENDED BOOKS



If you want to build interactive applications on the desktop, in the browser, or on mobile devices, this new edition of the ActionScript 3.0 Bible is all you need.

BUY NOW

books.dzone.com/books/actionscript3



Browse our collection of over 100 Free Cheat Sheets

Free PDF

Upcoming Refcardz

Network Security
ALM
Solr
Subversion

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more.
"DZone is a developer's dream," says PC Magazine.

DZone, Inc.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-75-2
ISBN-10: 1-934238-75-9

50795

9 781934 238752

$7.95

Copyright © 2010 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Version 1.0