JW Player for Flash

Release 4.7

www.longtailvideo.com

July 27, 2010

CONTENTS

CHAPTER

ONE

INTRODUCTION

The JW Player for Flash is the Internet's most popular and flexible media player. It supports playback of *any media type* the Adobe Flash Player can handle, both by using simple downloads, *HTTP Pseudostreaming* and *RTMP Streaming*.

The player supports various *playlist formats* and a wide range of *options* (flashvars) for changing its layout and behaviour. Embedding the player in a webpage *is a breeze*.

1.1 API

For javascript developers, the player features an extensive *API*. With this API, it is possible to both control the player (e.g. pause it) and respond to playback changes (e.g. when the video has ended).

API developers might find the *JW Player Architecture* interesting as well; it describes in broad strokes how the player is architected and initialized on startup.

1.2 Addons

Both the layout and the behaviour of the player can be extended with a range of so-called addons. These addons are available on the LongTail Video website. There are two categories:

- **Skins** drastically change the looks of the player. A wide range of professional-looking skins can be downloaded.
- **Plugins** extend the functionality of the player, e.g. in the areas of analytics, advertising or viral sharing. Plugins are loaded from our plugin repository, making them extremely easy to install.

It is possible to create your own skins and plugins using Adobe Flash and actionscript, but this is not covered by these documents. Instead, visit developer.longtailvideo.com to learn more and download the various SDKs we offer.

CHAPTER

TWO

EMBEDDING THE PLAYER

Like every other Flash object, the JW Player has to be embedded into the HTML of a webpage using specific embed codes. Overall, there are two methods for embedding Flash:

- Using a javascript (like SWFObject).
- Using a HTML tag (like *<embed>*).

We highly recommend the javascript method for Flash embedding. It can sniff if a browsers supports Flash, it ensures the player *API Reference* works and it avoids browser compatibility issues. Detailed instructions can be found below.

2.1 Upload

First, a primer on uploading. This may sound obvious, but for the JW Player to work on your website, you must upload the *player.swf* file from the download (or SVN checkout) to your webserver. If you want to play Youtube videos, you must also upload the **yt.swf** file - this is the bridge between the player and Youtube. No other files are needed.

Your *media files* and *playlists* can be hosted at any domain. Do note that *Crossdomain Security Restrictions* apply when loading these files from a different domain. In short, playing media files works, but loading playlists across domains will not work by default. Resolve this issue by hosting a *crossdomain.xml file*.

2.2 SWFObject

The preferred way to embed the JW Player on a webpage is javascript. There's a wide array of good, open source libraries available for doing so. We recommend **SWFObject**, the most widely used one. It has excellent documentation.

Before embedding any players on the page, make sure to include the *swfobject.js* script in the *<head>* of your HTML. You can download the script and host it yourself, or leverage the copy provided by Google:

```
<script type="text/javascript"
  src="http://ajax.googleapis.com/ajax/libs/swfobject/2.2/swfobject.js">
</script>
```

With the library set up, you can start embedding players. Here's an example:

```
<p id="container1">You don't have Flash ...</p>

<script type="text/javascript">
  var flashvars = { file:'/data/bbb.mp4',autostart:'true' };
```

```
  var params = { allowfullscreen:'true', allowscriptaccess:'always' };
  var attributes = { id:'player1', name:'player1' };

  swfobject.embedSWF('player.swf','container1','480','270','9.0.115','false',
    flashvars, params, attributes);
</script>
```

It's a fairly sizeable chunk of code that contains the embed *container*, *flashvars*, *params*, *attributes* and *instantiation*. Let's walk through them one by one:

- The *container* is the HTML element where the player will be placed into. It should be a block-level element, like a <p> or . If a user has a sufficient version of Flash, the text inside the container is removed and replaced by the videoplayer. Otherwise, the contents of the container will remain visible.
- The *flashvars* object lists your player *Configuration Options*. One option that should always be there is *file*, which points to the file to play. You can insert as many options as you want.
- The *params* object includes the Flash plugin parameters. The two parameters in the example (our recommendation) enable both the *fullscreen* and *javascript* functionality of Flash.
- The *attributes* object include the HTML attributes of the player. We recommend always (and only) setting an *id* and *name*, to the same value. This will be the *id* of the player instance if you use its *API Reference*.
- The *instantiation* is where all things come together and the actual player embedding takes place. These are all parameters of the SWFObject call:
  - The URL of the *player.swf*, relative to the page URL.
  - The ID of the container you want to embed the player into.
  - The width of the player, in pixels. Note the JW Player automatically stretches itself to fit.
  - The height of the player, in pixels. Note the JW Player automatically stretches itself to fit.
  - The required version of Flash. We highly recommend setting *9.0.115*. This is the first version that supports *MP4* and is currently installed at >95% of all computers. The only feature for which you might restricted to *10.0.0* is *RTMP dynamic streaming*.
  - The location of a Flash auto-upgrade script. We recommend to **not** use it. People that do not have Flash 9.0.115 either do not want or are not able (no admin rights) to upgrade.
  - Next, the *flashvars*, *params* and *attributes* are passed, in that order.

It is no problem to embed multiple players on a page. However, make sure to give each player instance a different container **id** and a different attributess **id** and **name**.

2.3 Embed tag

In cases where a javascript embed method is not possible (e.g. if your CMS does not allow including javascripts), the player can be embedded using plain HTML. There are various combinations of tags for embedding a SWF player:

- A single <*embed*> tag (for IE + other browsers).
- An <*object*> tag with nested <*embed*> tag (the first one for IE, the second for other browsers).
- An <*object*> tag with nested <*object*> tag (the first one for IE, the second for other browsers).

We recommend using the single <*embed*> tag. This works in all current-day browsers (including IE6) and provides the shortest codes. Here is an example embed code that does exactly the same as the SWFObject example above:

```
<embed
  flashvars="file=/data/bbb.mp4&autostart=true"
  allowfullscreen="true"
  allowscripaccess="always"
  id="player1"
  name="player1"
  src="player.swf"
  width="480"
  height="270"
/>
```

As you can see, most of the data of the SWFObject embed is also in here:

- The **container** is now the embed tag itself. The *fallback* text cannot be used anymore.
- The **flashvars** are merged into a single string, and loaded as an attribute. You should always concatenate the flashvars using so-called querystring parameter encoding: *flashvar1=value1&flashvar2=value2&....*
- The **params** each are individual attributes of the embed tag.
- The **attributes** also are individual attributes of the embed tag.
- The **instantiation** options (source, width, height) are attributes of the embed tag.

Note: As you can see, the Flash version reference is not in the embed tag: this is one of the drawbacks of this method: it's not possible to sniff for Flash and selectively hide it, e.g. if the flash version is not sufficient or if the device (iPad ...) doesn't support Flash.

CHAPTER

THREE

CONFIGURATION OPTIONS

Here's a list of all configuration options (flashvars) the player accepts. Options are entered in the *embed code* to set how the player looks and functions.

3.1 Encoding

First, a note on encoding. You must URL encode the three glyphs **? = &** inside flashvars, because of the way these flashvars are loaded into the player (as a querystring). The urlencoded values for these symbols are listed here:

- ? → %3F
- = → %3D
- & → %26

If, for example, your **file** flashvar is at the location *getplaylist.php?id=123&type=flv*, you must encode the option to:

```
getplaylist.php%3Fid%3D123%26type%3Dflv
```

The player will automatically URLdecode every option it receives.

3.2 File properties

These options set different properties of the mediafile to load (e.g. the source file, preview image or video title). Each of these option can be set for every entry in a *playlist*.

duration (0)
: Duration of the file in seconds. Set this to present the duration in the controlbar before the video starts. It can also be set to a shorter value than the actual file duration. The player will restrict playback to only that section.

file (undefined)
: Location of the file or playlist to play, e.g. *http://www.mywebsite.com/myvideo.mp4*.

image (undefined)
: Location of a preview (poster) image; shown in display before the video starts.

link (undefined)
: URL to an external page the display can link to (see *displayclick* below). Sharing - related plugins also use this link.

start (0)
Position in seconds where playback should start. This option works for *HTTP Pseudostreaming*, *RTMP Streaming* and the MP3 and Youtube *files*. It does not work for regular videos.

streamer (undefined)
Location of an RTMP or HTTP server instance to use for streaming. Can be an RTMP application or external PHP/ASP file. See *RTMP Streaming* and *HTTP Pseudostreaming*.

type (undefined)
Set this flashvar to tell the player in which format (regular/streaming) the player is. By default, the type is detected by the player based upon the file extension. If there is no suiteable extension, it can be manually set. The following media types are supported:

- **video**: progressively downloaded FLV / MP4 video, but also AAC audio. See *Media Support*.
- **sound**: progressively downloaded MP3 files. See *Media Support*.
- **image**: JPG/GIF/PNG images. See *Media Support*.
- **youtube**: videos from Youtube. See *Media Support*.
- **http**: FLV/MP4 videos using HTTP pseudo-streaming. See *HTTP Pseudostreaming*.
- **rtmp**: FLV/MP4/MP3 files or live streams using RTMP streaming. See *RTMP Streaming*.

3.3 Appearance

These flashvars control the looks of the player.

backcolor (ffffff)
background color of the controlbar and playlist. This is white by default.

controlbar (bottom)
Position of the controlbar. Can be set to *bottom*, *over* and *none*.

frontcolor (000000)
color of all icons and texts in the controlbar and playlist. Is black by default.

lightcolor (000000)
Color of an icon or text when you rollover it with the mouse. Is black by default.

logo (undefined)
Location of an external JPG, PNG or GIF image to show in a corner of the display. With the default skin, this is top-right, but every skin can freely place the logo.

playlist (none)
Position of the playlist. Can be set to **bottom**, **right**, **left**, **over** or **none**.

playlistsize (180)
When the playlist is positioned below the display, this option can be used to change its height. When the playlist lives left or right of the display, this option represents its width. In the other cases, this option isn't needed.

screencolor (000000)
Background color of the display. Is black by default.

skin (undefined)
Location of a so-called **skin**, an SWF file with the player graphics. Our addons repository contains a list of available skins.

The color flashvars need so-called hexadecimal values, as is common for web colors (e.g. ''FFCC00" for bright yellow).

3.4 Behaviour

These flashvars control the playback behaviour of the player.

autostart (false)
Set this to *true* to automatically start the player on load.

bufferlength (1)
Number of seconds of the file that has to be loaded before the player starts playback. Set this to a low value to enable instant-start (good for fast connections) and to a high value to get less mid-stream buffering (good for slow connections).

displayclick (play)
What to do when a user clicks the display. Can be:

- **play**: toggle playback
- **link**: jump to the URL set by the *link* flashvar.
- **none**: do nothing (the handcursor is also not shown).

dock (false)
set this to **true** to list plugin buttons in display. By default (*false*), plugin buttons are shown in the controlbar.

linktarget (_blank)
Browserframe where link from the display are opened in. Some possibilities are *_self* (same frame) or *_blank* (new browserwindow).

mute (false)
Mute the sounds on startup. Is saved in a cookie.

plugins (undefined)
A powerful feature, this is a comma-separated list of plugins to load (e.g. **hd,viral**). Plugins are separate SWF files that extend the functionality of the player, e.g. with advertising, analytics or viral sharing features. Visit our addons repository to browse the available plugins.

repeat (none)
What to do when the mediafile has ended. Has several options:

- **none**: do nothing (stop playback) whever a file is completed.
- **list**: play each file in the playlist once, stop at the end.
- **always**: continously play the file (or all files in the playlist).
- **single**: continously repeat the current file in the playlist.

shuffle (false)
Shuffle playback of playlist items. The player will randomly pick the items.

smoothing (true)
This sets the smoothing of videos, so you won't see blocks when a video is upscaled. Set this to **false** to disable the feature and get performance improvements with old computers / big files.

stretching (uniform)
Defines how to resize the poster image and video to fit the display. Can be:

- **none**: keep the original dimensions.
- **exactfit**: disproportionally stretch the video/image to exactly fit the display.
- **uniform**: stretch the image/video while maintaining its aspect ratio. There'll be black borders.
- **fill**: stretch the image/video while maintaining its aspect ratio, completely filling the display.

3.5 Config XML

All options can be listed in an XML file and then fed to the player with a single option:

config (undefined)
: location of a XML file with flashvars. Useful if you want to keep the actual embed codes short. Here's an example:

Here is an example of such an XML file:

```
<config>
  <file>files/bunny.mp4</file>
  <image>files/bunny.jpg</image>
  <repeat>true</repeat>
  <backcolor>333333</backcolor>
  <volume>40</volume>
  <controlbar>over</controlbar>
</config>
```

Options set in the embed code will overwrite those set in the config XML.

Note: Due to the *Crossdomain Security Restrictions* restrictions of Flash, you cannot load a config XML from one domain in a player on another domain. This issue can be circumvented by placing a *crossdomain.xml* file on the server that hosts your XML.

CHAPTER

FOUR

MEDIA SUPPORT

This page lists all media file formats the JW Player supports: video, sound, images and Youtube clips.

Single media files can be grouped using *playlists* and streamed over *http* or *rtmp* instead of downloaded. Both options do not change the set of supported media formats.

Note: The player always tries to recognize a file format by its extension. If no suitable extension is found, **the player will presume you want to load a playlist**! Work around this issue by setting the *type option*.

4.1 Video

The player supports video (*type=video*) in the following formats:

`H.264 ( .mp4, .mov, .f4v )`
: Video in either the MP4 or *Quicktime <http://en.wikipedia.org/wiki/Quicktime>* container format. These files must contain video encoded with the H.264 codec and audio encoded with the AAC codec. H264/AAC video is today's format of choice. It can also be played on a wide range of (mobile) devices.

  Note: If you cannot seek within an MP4 file be before it is completely downloaded, the cause of this problem is that the so-called MOOV atom (which contains the seeking information) is located at the end of your video. Check out this little application to parse your videos and fix it.

`FLV ( .flv )`
: Video in the Flash Video container format. These files can contain video encoded with both the ON2 VP6 codec and the Sorenson Spark codec. Audio must be in the MP3 codec. FLV is a slightly outdated format. It is also unique to Flash.

  Note: If the progress bar isn't running with your FLV file, or if your video dimensions are wrong, this means that your FLV file doesn't have metadata. Fix this by using the small tool from buraks.com.

`3GPP ( .3gp, .3g2 )`
: Video in the 3GPP container format. These files must contain video encoded with the H.263 codec and audio encoded with the AAC codec. Used widely for mobile phones because it is easy to decode. More and more devices switch to H264 though.

`AAC ( .aac, .m4a )`
: Audio encoded with the AAC codec. Indeed, this is not video! However, the player must use the **video** type to playback this audio, since the **sound** type only supports MP3. State of the art codec, widely supported.

4.2 Sound

The player supports sounds (*type=sound*) in the following formats:

MP3 (.mp3)
Audio encoded with the MP3 codec. Though not as good as AAC, MP3 is very widely used. It is also support by nearly any device that can play audio.

Note: If you encounter too fast or too slow playback of MP3 files, it contains variable bitrate encoding or unsupported sample frequencies (eg 48Khz). Please stick to constant bitrate encoding and 44 kHz. The free iTunes software has an MP3 encoder built-in.

4.3 Images

The player supports images (*type=image*) in the following formats:

JPEG (.jpg)
Images encoded with the JPEG algorythm. No transparency support.

PNG (.png)
Images encoded with the PNG algorythm. Supports transparency.

GIF (.gif)
Images encoded with the GIF algorythm. Supports transparency, but pixels can only be opaque or 100% transparent.

Note: The player does NOT support animated GIFs.

SWF (.swf)
Drawings/animations encoded in the Adobe Flash format. Supports transparency.

Note: Though SWF files load in the player, it is discouraged to use them. The player cannot read the duration and dimensions of SWF files. Custom scripts inside these SWF files might also interfere with (or break) playback.

4.4 Youtube

The player includes native support for playing back Youtube videos (*type=youtube*). Youtube playback is automatically enabled when the **file** option is assigned to the URL of a Youtube video (e.g. *http://www.youtube.com/watch?v=WuQnd3d9IuA*).

The player uses the official Youtube API for this functionality, so this is definitely not a hack. Youtube officially support playback of its content in third-party players like the JW Player.

The Youtube API is accessed through a bridge, the separate **yt.swf** file included in the player download.

Note: In order for Youtube videos to play, you must upload the *yt.swf* file to the same directory as the *player.swf*.

CHAPTER

FIVE

PLAYLIST SUPPORT

First, note that playlist XML files are subject to the *Crossdomain Security Restrictions* of Flash. This means that a videoplayer on one domain cannot load a playlist from another domain. It can be fixed by placing a *crossdomain.xml* file at the server the playlist is loaded from.

If your playlist and player.swf are hosted on the same domain, these restrictions don't apply.

5.1 Supported XML Formats

That said, the following playlist formats are supported:

- ASX feeds
- ATOM feeds with Media extensions
- RSS feeds with iTunes extensions and Media extensions
- XSPF feeds

Here is an overview of all the tags of each format the player processes, and the property in the JW Player playlist they correspond to:

JW Player	XSPF	RSS	itunes:	media:	ASX	ATOM
author	creator	(none)	author	credit	author	(none)
date	(none)	pubDate	(none)	(none)	(none)	published
description	annotation	description	summary	description	abstract	summary
duration	duration	(none)	duration	content	duration	(none)
file	location	enclosure	(none)	content	ref	(none)
link	info	link	(none)	(none)	moreinfo	link
image	image	(none)	(none)	thumbnail	(none)	(none)
start	(none)	(none)	(none)	(none)	starttime	(none)
streamer	(none)	(none)	(none)	(none)	(none)	(none)
tags	(none)	category	keywords	keywords	(none)	(none)
title	title	title	(none)	title	title	title
type	(none)	(none)	(none)	(none)	(none)	(none)

All **media:** tags can be embedded in a **media:group** element. A **media:content** element can also act as a container.

5.2 JWPlayer Namespace

In order to enable all JW Player file properties for all feed formats, the player contains a **jwplayer** namespace. By inserting this into your feed, properties that are not supported by the feed format itself (such as the **streamer**) can be

amended without breaking validation. Any of the entries listed in the above table can be inserted. Here's an example, of a video that uses *RTMP Streaming*:

```
<rss version="2.0" xmlns:jwplayer="http://developer.longtailvideo.com/">
  <channel>
    <title>Example RSS feed with jwplayer extensions</title>
    <item>
      <title>Big Buck Bunny</title>
      <jwplayer:file>videos/nPripu91-60830.mp4</jwplayer:file>
      <jwplayer:streamer>rtmp://myserver.com/myApp/</jwplayer:streamer>
      <jwplayer:duration>34</jwplayer:duration>
    </item>
  </channel>
</rss>
```

Pay attention to the top level tag, which describes the JW Player namespace with the xmlns attribute. This must be available in order to not break validity.

5.3 Mixing namespaces

You can mix **jwplayer** elements with both the regular elements of a feed and elements from the mRSS and iTunes extensions. If multiple elements match the same playlist entry, the elements will be prioritized:

- Elements that are defined by the feed format (e.g. the *enclosure* in RSS) get the lowest priority.
- Elements defined by the *itunes* namespace rank third.
- Element defined by the *media* namespace (e.g. *media:content*) rank second.
- Elements defined by the *jwplayer* extension always gets the highest priority.

This feature allows you to set, for example, a specific video version or streaming features for the JW Player, while other feed aggregators will pick the default content. In the above example feed, we could insert a regular *enclosure* element that points to a download of the video. This would make the feed useful for both the JW Player and text-oriented aggregators such as Feedburner.

CHAPTER

SIX

HTTP PSEUDOSTREAMING

Both MP4 and FLV videos can be played back with a mechanism called HTTP Pseudostreaming. This mechanism allows your viewers to seek to not-yet downloaded parts of a video. Youtube is an example site that offers this functionality. HTTP pseudostreaming is enabled by setting the *option type=http* in your player.

HTTP pseudostreaming combines the advantages of straight HTTP downloads (it passes any firewall, viewers on bad connections can simply wait for the download) with the ability to seek to non-downloaded parts. The only drawbacks of HTTP Pseudostreaming compared to Flash's official *RTMP Streaming* are its reduced security (HTTP is easier to sniff than RTMP) and long loading times when seeking in large videos (> 15 minutes).

HTTP Pseudostreaming should not be confused with HTTP Dynamic Streaming. The latter is a brand-new mechanism currently being developed by Adobe that works by chopping up the original video in so-called *chunks* of a few seconds each. The videoplayer seamlessly glues these chunks together again. This version of the JW Player does **not** support HTTP Dynamic Streaming.

6.1 Servers

HTTP Pseudostreaming does not work by default on any webserver. A serverside module is needed to enable it. Here are the two most widely used (and open source) modules for this:

- The H264 streaming module for Apache, Lighttpd, IIS and NginX. It supports MP4 videos.
- The FLV streaming module for Lighttpd. It supports FLV videos.

Several CDN's (Content Delivery Networks) support HTTP Pseudostreaming as well. We have done succesfull tests with Bitgravity, CDNetworks, Edgecast and Limelight.

Instead of using a serverside module, pseudostreaming can also be enabled by using a serverside script (in e.g. PHP or .NET). We do not advise this, since such a script consumes a lot of resources, has security implications and can only be used with FLV files. A much-used serverside script for pseudostreaming is Xmoov-PHP.

6.2 Mechanism

Under water, HTTP pseudostreaming works as follows:

When the video is initially loaded, the player reads and stores a list of *seekpoints* as part of the video's metadata. These seekpoints are offsets in the video (both in seconds and in bytes) at which a new *keyframe* starts. At these offsets, a request to the server can be made.

When a user seeks to a not-yet-downloaded part of the video, the player translates this seek to the nearest seekpoint. Next, the player does a request to the server, with the seekpoint offset as a parameter. For FLV videos, the offset is always provided in bytes:

```
http://www.mywebsite.com/videos/bbb.flv?start=219476905
```

For MP4 videos, the offset is always provided in seconds:

```
http://www.mywebsite.com/videos/bbb.mp4?starttime=30.4
```

The server will return the video, starting from the offset. Because the first frame in this video is a keyframe, the player is able to correctly load and play it. Should the server have returned the video from an arbitrary offset, the player would not be able to pick up the stream and the display would only show garbage.

Note: Some FLV encoders do not include seekpoints metadata when encoding videos. Without this data, HTTP Pseudostreaming will not work. If you suspect your videos to not have metadata, use our Metaviewer plugin to inspect the video. There should be a *seekpoints* or *keyframes* list. If it is not there, use the FLVMDI tool to parse your FLV videos and inject this metadata.

6.3 Startparam

When the player requests a video with an offset, it uses *start* as the offset parameter name for FLV videos and *starttime* as the offset parameter name for MP4 videos:

```
http://www.mywebsite.com/videos/bbb.flv?start=219476905
http://www.mywebsite.com/videos/bbb.mp4?starttime=30.4
```

These names are most widely used by serverside modules and CDNs. However, sometimes a CDN might use a different name for this parameter. In that case, use the option *http.startparam* to set a custom offset parameter name. Here are some examples of CDNs that use a different name:

- Bitgravity uses *http.startparam=apstart* for MP4 videos.
- Edgecast uses *http.startparam=ec_seek* for FLV videos.
- Limelight uses *http.startparam=fs* for FLV videos.

Here's what an example SWFObject *embed code* looks like when both HTTP Pseudostreaming and a custom start parameter is enabled:

```
<div id='container'>The player will be placed here</div>

<script type="text/javascript">
  swfobject.embedSWF('player.swf','container','480','270','9.0.115','false',{
    file:'http://bitcast-a.bitgravity.com/botr/bbb.mp4',
    type:'http',
    'http.startparam':'apstart'
  });
</script>
```

6.4 Playlists

HTTP Pseudostreaming can also be enabled in playlists, by leveraging the *JWPlayer namespace*. Both the *type* and *http.startparam* options can be set for every entry in a playlist. In this case, you don't have to set them in the embed code (just point the *file* to your playlist).

Here's an example, an RSS feed with a single video:

```
<rss version="2.0" xmlns:jwplayer="http://developer.longtailvideo.com/">
  <channel>
    <title>Playlist with HTTP Pseudostreaming</title>

    <item>
      <title>Big Buck Bunny</title>
      <description>Big Buck Bunny is a short animated film by the Blender Institute,
         part of the Blender Foundation.</description>
      <enclosure url="http://myserver.com/botr/bbb.mp4" type="video/mp4" length="3192846" />
      <jwplayer:type>http</jwplayer:type>
      <jwplayer:http.startparam>apstart</jwplayer:http.startparam>

    </item>
  </channel>
</rss>
```

Instead of the *enclosure* element, you can also use the *media:content* or *jwplayer:file* element. More info in *Playlist Support*.

Note: Do not forget the **xmlns** at the top of the feed. It is needed by the player (and any other feed reader you might use) to understand the *jwplayer:* elements.

6.5 Bitrate Switching

Like with *RTMP Streaming*, HTTP Pseudostreaming includes the ability to dynamically optimize the video quality for each individual viewer. We call this mechanism *bitrate switching*.

To use bitrate swiching, you need multiple copies of your MP4 or FLV video, each with a different quality (dimensions and bitrate). These multiple videos are loaded into the player using an mRSS playlist (see example below). The player recognizes the various *levels* of your video and automatically selects the highest quality one that:

- Fits the *bandwidth* of the server » client connection.
- Fits the *width* of the player's display (or, to be precise, is not more than 20% larger).

As a viewer continues to watch the video, the player re-examines its decision (and might switch) in response to certain events:

- On **startup**, immediately after it has calculated the bandwidth for the first time.
- On a **fullscreen** switch, since the *width* of the display then drastically changes. For example, when a viewer goes fullscreen and has sufficient bandwidth, the player might serve an HD version of the video.
- On every **seek** in the video. Since the player has to rebuffer-the stream anyway, it takes the opportunity to also check if bandwidth conditions have not changed.

Note that the player will not do a bandwidth switch if extreme bandwidth changes cause the video to re-buffer. In practice, we found such a heuristic to cause continous switching and an awful viewing experience.

6.5.1 Example

Here is an example bitrate switching playlist (only one item). Note that it is similar to a *regular* HTTP Pseudostreaming playlist, with the exception of the multiple video elements per item. The mRSS extension is the only way to provide these multiple elements including *bitrate* and *width* attributes:

```
<rss version="2.0" xmlns:media="http://search.yahoo.com/mrss/"
  xmlns:jwplayer="http://developer.longtailvideo.com/">
  <channel>
    <title>Playlist with HTTP Bitrate Switching</title>

    <item>
      <title>Big Buck Bunny</title>
      <description>Big Buck Bunny is a short animated film by the Blender Institute,
         part of the Blender Foundation.</description>
      <media:group>
        <media:content bitrate="1800" url="http://myserver.com/bbb-4.mp4"  width="1280" />
        <media:content bitrate="1100" url="http://myserver.com/bbb-3.mp4" width="720"/>
        <media:content bitrate="700" url="http://myserver.com/bbb-2.mp4" width="480" />
        <media:content bitrate="400" url="http://myserver.com/bbb-1.mp4" width="320" />
      </media:group>
      <jwplayer:type>http</jwplayer:type>
      <jwplayer:http.startparam>apstart</jwplayer:http.startparam>
    </item>

  </channel>
</rss>
```

Some hints:

- The *bitrate* attributes must be in kbps, as defined by the mRSS spec. The *width* attribute is in pixels.
- It is recommended to order the streams by quality, the best one at the beginning. Most RSS readers will pick this one.
- The four levels displayed in this feed are actually what we recommend for bitrate switching of widescreen MP4 videos. For 4:3 videos or FLV videos, you might want to increase the bitrates or decrease the dimensions a little.
- Some publishers only modify the bitrate when encoding multiple levels. The player can work with this, but modifying both the bitrate + dimensions allows for more variation between the levels (and re-use of videos, e.g. the smallest one for streaming to phones).
- The *media:group* element here is optional, but it organizes the video links a little.

CHAPTER

SEVEN

RTMP STREAMING

RTMP (Real Time Messaging Protocol) is a system for delivering on-demand and live media to Adobe Flash applications (like the JW Player). RTMP supports video in both FLV and H.264 (MP4/MOV/F4V) *format* and audio in both MP3 and AAC (M4A) format. RTMP offers several advantages over regular HTTP video downloads:

- RTMP can do live streaming - people can watch your video while it is being recorded.
- With RTMP, viewers can seek to not-yet-downloaded parts of a video. This is especially useful for longer-form content (> 10 minutes).
- Videos delivered over RTMP (and its secure brother, RTMPE) are harder to steal than videos delivered over regular HTTP.

However, do note that RTMP has its disadvantages too. Especially since the introduction of *HTTP Pseudostreaming* (which also offer seeking to not-yet-downloaded parts), RTMP is not the only option for professional video delivery. Some drawbacks to be aware of:

- RTMP is a different protocol than HTTP and is sent over a different port (1935 instead of 80). Therefore, RTMP is frequently blocked by (corporate) firewalls. This can be circumvented by using RTMPT (tunneled), but this comes at a performance cost (longer buffer times - 2x in our experience).
- RTMP is a *true* streaming protocol, which means that the bandwidth of the connection must always be larger than the datarate of the video. If the connection drops for just a few seconds, the stream will stutter. If the connection overall is just a little less than the video datarate, the video will not play at all. With *HTTP Pseudostreaming* on the other hand, people can simply wait until more of the video is downloaded.

The JW Player supports a wide array of features of the RTMP protocol.

7.1 Servers

In order to use RTMP, your webhoster or CDN needs to have a dedicated RTMP webserver installed. There are three offerings, each one of them supported by the JW Player:

- The Flash Media Server from Adobe is the de facto standard. Since Flash is also developed by Adobe, new video functionalities always find their way in FMS first.
- The Wowza Media Server from Wowza is a great alternative, because it includes support for other streaming protocols than RTMP (for e.g. Shoutcast, the iPhone or Silverlight).
- The Red5 Media Server is an open-source RTMP alternative. It lags in features (e.g. no dynamic streaming), but has an active and open community of developers.

RTMP servers are not solely used for one-to-many media streaming. They include support for such functionalities as video conferencing, document sharing and multiplayer games. Each of these functionalities is separately set up on the

server in what is called an *application*. Every application has its own URL (typically a subfolder of the root). For example, these might be the path to both an on-demand streaming and live streaming application on your webserver:

```
rtmp://www.myserver.com/ondemand/
rtmp://www.myserver.com/live/
```

7.2 Options

To play an RTMP stream in the player, both the *streamer* and *file options* must be set. The *streamer* is set to the server + path of your RTMP application. The *file* is set to the internal URL of video or audio file you want to stream. Here is an example *embed code*:

```
<div id='container'>The player will be placed here</div>

<script type="text/javascript">
  swfobject.embedSWF('player.swf','container','480','270','9.0.115','false',{
    file:'library/clip.mp4',
    streamer:'rtmp://www.myserver.com/ondemand/'
  });
</script>
```

Note that the documentation of RTMP servers tell you to set the *file* option in players like this:

- For FLV video: **file=clip** (without the *.flv* extension).
- For MP4 video: **file=mp4:clip.mp4** (with *mp4:* prefix).
- For MP3 audio: **file=mp3:song.mp3** (with *mp3:* prefix).
- For AAC audio: **file=mp4:song.aac** (with *mp4:* prefix).

You do not have to do this with the JW Player, since the player takes care of stripping the extension or adding the prefix. If you do add the prefix yourself, the player will recognize it and not modify the URL.

Additionally, the player will leave querystring variables (e.g. for certain CDN security mechanisms) untouched. It basically ignores everything after the **?** character.

Note: Because of the way options are loaded into Flash, it is not possible to use querystring delimiters (*?*, *=*, *&*) inside a single option. This issue can be circumvented by URL encoding these three characters. More info can be found in *Configuration Options*.

7.3 Playlists

RTMP streams can also be included in playlists, by leveraging the *JWPlayer namespace*. The *streamer* option should be set for every RTMP entry in a playlist. You don't have to set them in the embed code (just point the *file* option to your playlist).

Here's an example, an RSS feed with an RTMP video and audio clip:

```
<rss version="2.0" xmlns:jwplayer="http://developer.longtailvideo.com/">
  <channel>
    <title>Playlist with RTMP streams</title>

    <item>
      <title>Big Buck Bunny</title>
```

```
      <description>Big Buck Bunny is a short animated film by the Blender Institute,
         part of the Blender Foundation.</description>
      <enclosure url="files/bbb.mp4" type="video/mp4" length="3192846" />
      <jwplayer:streamer>rtmp://myserver.com/ondemand</jwplayer:streamer>
    </item>

    <item>
      <title>Big Buck Bunny (podcast)</title>
      <description>Big Buck Bunny is a short animated film by the Blender Institute,
         part of the Blender Foundation.</description>
      <enclosure url="files/bbb.mp3" type="audio/mp3" length="3192846" />
      <jwplayer:streamer>rtmp://myserver.com/ondemand</jwplayer:streamer>
    </item>

  </channel>
</rss>
```

Instead of the *enclosure* element, you can also use the *media:content* or *jwplayer:file* element. You could even set the *enclosure* to a regular http download of the video ánd *jwplayer:file* to the RTMP stream. That way, this single feed is useful for both regular RSS readers and the JW Player. More info in *Playlist Support*.

Note: Do not forget the **xmlns** at the top of the feed. It is needed by the player (and any other feed reader you might use) to understand the *jwplayer:* elements.

7.4 Live Streaming

A unique feature of RTMP is the ability to do live streaming, e.g. of presentations, concerts or sports events. Next to the player and an RTMP server, one then also needs a small tool to *ingest* (upload) the live video into the server. There's a bunch of such tools available, but the easiest to use is the (free) Flash Live Media Encoder. It is available for Windows and Mac.

A live stream can be embedded in the player using the same options as an on-demand stream. The only difference is that a live stream has no file extension. Example:

```
<div id='container'>The player will be placed here</div>

<script type="text/javascript">
  swfobject.embedSWF('player.swf','container','480','270','9.0.115','',{
    file:'livepresentation',
    streamer:'rtmp://www.myserver.com/live/'
  });
</script>
```

7.4.1 Subscribing

When streaming live streams using the Akamai or Limelight CDN, players cannot simply connect to the live stream. Instead, they have to *subscribe* to it, by sending an **FCSubscribe call** to the server. The JW Player includes support for this functionality. Simply add the *rtmp.subscribe=true* option to your embed code to enable:

```
<div id='container'>The player will be placed here</div>

<script type="text/javascript">
  swfobject.embedSWF('player.swf','container','480','270','9.0.115','false',{
```

```
    file:'livepresentation',
    streamer:'rtmp://www.myserver.com/live/',
    'rtmp.subscribe':'true'
  });
</script>
```

7.5 Dynamic Streaming

Like with *HTTP Pseudostreaming*, RTMP Streaming includes the ability to dynamically optimize the video quality for each individual viewer. Adobe calls this mechanism *dynamic streaming*. This functionality is supported for FMS 3.5+ and Wowza 2.0+.

To use dynamic streaming, you need multiple copies of your MP4 or FLV video, each with a different quality (dimensions and bitrate). These multiple videos are loaded into the player using an mRSS playlist (see example below). The player recognizes the various *levels* of your video and automatically selects the highest quality one that:

- Fits the *bandwidth* of the server » client connection.
- Fits the *width* of the player's display (or, to be precise, is not more than 20% larger).

As a viewer continues to watch the video, the player re-examines its decision (and might switch) in response to certain events:

- On a **bandwidth** increase or decrease - the bandwidth is re-calculated at an interval of 2 seconds.
- On a **resize** of the player. For example, when a viewer goes fullscreen and has sufficient bandwidth, the player might serve an HD version of the video.

A dynamic streaming switch is unobtrusive. There'll be no re-buffering or audible/visible hickup. It does take a few seconds for a switch to occur in response to a bandwidth change / player resize, since the server has to wait for a *keyframe* to do a smooth switch and the player always has a few seconds of the old stream in its buffer. To keep stream switches fast, make sure your videos are encoded with a small (2 to 4 seconds) keyframe interval.

Note: So far, we have not been able to combine dynamic streaming with live streaming. This functionality is highlighted in documentation from Adobe and Wowza, but in our tests we found that the bandwidth the player receives never exceeds the bandwidth of the level that currently plays. In other words: the player will never switch to a higher quality stream than the one it starts with. This seems to be a bug in the Flash plugin, since both FMS and Wowza have this issue.

7.5.1 Example

Here is an example dynamic streaming playlist (only one item). It is similar to a regular RTMP Streaming playlist, with the exception of the multiple video elements per item. The mRSS extension is the only way to provide these multiple elements including *bitrate* and *width* attributes:

```
<rss version="2.0" xmlns:media="http://search.yahoo.com/mrss/"
  xmlns:jwplayer="http://developer.longtailvideo.com/">
  <channel>
    <title>Playlist with RTMP Dynamic Streaming</title>

    <item>
      <title>Big Buck Bunny</title>
      <description>Big Buck Bunny is a short animated film by the Blender Institute,
         part of the Blender Foundation.</description>
      <media:group>
```

```
        <media:content bitrate="1800" url="videos/Qvxp3Jnv-4.mp4"  width="1280" />
        <media:content bitrate="1100" url="videos/Qvxp3Jnv-3.mp4" width="720"/>
        <media:content bitrate="700" url="videos/Qvxp3Jnv-2.mp4" width="480" />
        <media:content bitrate="400" url="videos/Qvxp3Jnv-1.mp4" width="320" />
      </media:group>
      <jwplayer:streamer>rtmp://www.myserver.com/ondemand/</jwplayer:streamer>
    </item>

  </channel>
</rss>
```

Some hints:

- The *bitrate* attributes must be in kbps, as defined by the mRSS spec. The *width* attribute is in pixels.
- It is recommended to order the streams by quality, the best one at the beginning.
- The four levels displayed in this feed are actually what we recommend for bitrate switching of widescreen MP4 videos. For 4:3 videos or FLV videos, you might want to increase the bitrates or decrease the dimensions a little.
- Some publishers only modify the bitrate when encoding multiple levels. The player can work with this, but modifying both the bitrate + dimensions allows for more variation between the levels (and re-use of videos, e.g. the smallest one for streaming to mobile phones).
- The *media:group* element here is optional, but it organizes the video links a little.

7.6 Load Balancing

For high-volume publishers who maintain several RTMP servers, the player supports load-balancing by means of an intermediate XML file. This is used by e.g. the Highwinds and VDO-X CDNs. Load balancing works like this:

- The player first requests the XML file (typically from single a *master* server).
- The server returns the XML file, which includes the location of the RTMP server to use (typically the server that's least busy).
- The player parses the XML file, connects to the server and starts the stream.

7.6.1 Example

Here's an example of such an XML file. It is in the SMIL format:

```
<smil>
  <head>
    <meta base="rtmp://server1234.mycdn.com/ondemand/" />
  </head>
  <body>
    <video src="library/myVideo.mp4" />
  </body>
</smil>
```

Here's an example embed code for enabling this functionality in the player. Note the *type=rtmp option* is needed in addition to *rtmp.loadbalance*, since otherwise the player thinks the XML file is a playlist.

```
<div id='container'>The player will be placed here</div>

<script type="text/javascript">
  swfobject.embedSWF('player.swf','container','480','270','9.0.115','false',{
    file:'http://www.mycdn.com/videos/myVideo.mp4.xml',
    type:'rtmp',
    'rtmp.loadbalance':'true'
  });
</script>
```

7.6.2 Playlists

RTMP Load balancing in playlists works in a similar fashion: the *type=rtmp* and *rtmp.loadbalance=true* options can be set for every entry in the playlist that uses loadbalancing. Here's an example with one item:

```
<rss version="2.0" xmlns:jwplayer="http://developer.longtailvideo.com/">
  <channel>
    <title>Playlist with RTMP loadbalancing</title>

    <item>
      <title>Big Buck Bunny (podcast)</title>
      <description>Big Buck Bunny is a short animated film by the Blender Institute,
         part of the Blender Foundation.</description>
      <enclosure url="http://www.mycdn.com/videos/bbb.mp3.xml" type="text/xml" length="185" />
      <jwplayer:type>rtmp</jwplayer:type>
      <jwplayer:rtmp.loadbalance>true</jwplayer:rtmp.loadbalance>
    </item>

  </channel>
</rss>
```

See the playlist section above for more information on format and element support.

Note: A combination of load balancing + dynamic streaming is not possible yet. We are working on such a functionality, which will be included in a future version of the player.

CHAPTER

EIGHT

API REFERENCE

The JW Player for Flash supports a flexible javascript API. It is possible to read the config/playlist variables of the player, send events to the player (e.g. to pause or load a new video) and listen (and respond) to player events. A small initialization routine is needed to connect your apps to the player.

8.1 Initialization

Please note that the player will **NOT** be available the instant your HML page is loaded and the first javascript is executed. The SWF file (40k) has to be loaded and instantiated first! You can catch this issue by defining a simple global javascript function. It is called *playerReady()* and every player that's succesfully instantiated will call it.

```
var player;
function playerReady(*object*) {
  alert('the player is ready');
  player = document.getElementById(object.id);
};
```

The *object* the player send to the function contain the following variables:

id
: ID of the player (the <*embed*> code) in the HTML DOM. Use it to get a reference to the player with *getElementById()*.

version
: Exact version of the player in MAJOR.MINOR.REVISION format *e.g. 4.7.1017*.

client
: Plugin version and platform the player uses, e.g. *FLASH WIN 10.0.47.0*.

If you are not interested in calling the player when the page is loading, you won't need the *playerReady()* function. You can then simply use the ID of the embed/object tag that embeds the player to get a reference. So for example with this embed tag:

```
<embed id="myplayer" name="myplayer" src="/upload/player.swf" width="400" height="200" />
```

You can get a pointer to the player with this line of code:

```
var player = document.getElementById('myplayer');
```

Note: Note you must add both the **id** and **name** attributes in the <*embed*> in order to get back an ID in all browsers.

8.2 Reading variables

There's two variable calls you can make through the API: *getConfig()* and *getPlaylist()*.

8.2.1 getConfig()

getConfig() returns an object with state variables of the player. For example, here we request the current audio volume, the current player width and the current playback state:

```
var volume = player.getConfig().volume;
var width = player.getConfig().width;
var state = player.getConfig().state;
```

Here's the full list of state variables:

bandwidth
: Current bandwidth of the player to the server, in kbps (e.g. *1431*). This is only available for videos, *HTTP Pseudostreaming* and *RTMP Streaming*.

fullscreen
: Current fullscreen state of the player, as boolean (e.g. *false*).

height
: Current height of the player, in pixels (e.g. *270*).

item
: Currently active (playing, paused) playlist item, as zero-index (e.g. *0*). Note that *0* means the first playlistitem is playing and *1* means the second one is playing.

level
: Currently active bitrate level, in case multipe bitrates are supplied to the player. This is only useful for *HTTP Pseudostreaming* and *RTMP Streaming*. Note that *0* always refers to the highest quality bitrate.

position
: current playback position, in seconds (e.g. *13.2*).

state
: Current playback state of the player, as an uppercase string. It can be one of the following:

  - *IDLE*: The current playlist item is not loading and not playing.
  - *BUFFERING*: the current playlistitem is loading. When sufficient data has loaded, it will automatically start playing.
  - *PLAYING*: the current playlist item is playing.
  - *PAUSED*: playback of the current playlistitem is not paused by the player.
  - *COMPLETED*: the current playlist item has completed playback. This state differs from the *IDLE* state in that the item is now already loaded.

mute
: Current audio mute state of the player, as boolean (e.g. *false*).

volume
: Current audio volume of the player, as a number from 0 to 100 (e.g. *90*).

width
: Current width of the player, in pixels (e.g. *480*).

Note: In fact, all the *Configuration Options* will be available in the response to *getConfig()*. In certain edge cases, this might be useful, e.g. when you want to know if the player did **autostart** or not.

8.2.2 getPlaylist()

getPlaylist() returns the current playlist of the player as an array. Each entry of this array is in turn again a hashmap with all the *playlist properties* the player recognizes. Here's a few examples:

```
var playlist = player.getPlaylist();
alert("There are " + playlist.length + " videos in the playlist");
alert("The title of the first entry is " + playlist[0].title);
alert("The poster image of the second entry is " + playlist[1].image);
alert("The media file of the third entry is " + playlist[2].file);
alert("The media type of the fourth entry is " + playlist[3].type);
```

8.3 Sending events

The player can be controlled from javascript by sending events (e.g. to pause it or change the volume). Sending events to the player is done through the *sendEvent()* call. Some of the event need a parameter and some don't. Here's a few examples:

```
// this will toggle playback.
player.sendEvent("play");
// this sets the volume to 90%
player.sendEvent("volume","true");
// This loads a new video in the player
player.sendEvent('load','http://content.bitsontherun.com/videos/nPripu9l-60830.mp4');
```

Here's the full list of events you can send, plus their parameters:

item (index:Number)
: Start playback of a specific item in the playlist. If *index* isn't set, the current playlistitem will start.

link (index:Number)
: Navigate to the *link* of a specific item in the playlist. If *index* is not set, the player will navigate to the link of the current playlistitem.

load (url:String)
: Load a new media file or playlist into the player. The *url* must always be sent.

  Note: Instead of a URL, it is also possible to send a full playlist object along with this call. For more info, see *Loading playlists* below.

mute (state:Boolean)
: Mute or unmute the player's sound. If the *state* is not set, muting will be toggled.

next
: Jump to the next entry in the playlist. No parameters.

play (state:Boolean)
: Play (set *state* to *true*) or pause (set *state* to *false*) playback. If the *state* is not set, the player will toggle playback.

prev
: Jump to the previous entry in the playlist. No parameters.

seek (position:Number)
Seek to a certain position in the currently playing media file. The *position* must be in seconds (e.g. *65* for one minute and five seconds).

Note: Seeking does not work if the player is in the *IDLE* state. Make sure to check the *state* variable before attempting to seek.

Additionally, for the *video* media type, the player can only seek to portions of the video that are already loaded. All other media types (*sound*, *image*, *youtube*, *http* and *rtmp* streaming) do not have this additional restriction.

stop
Stop playback of the current playlist entry and unload it. The player will revert to the *IDLE* state and the poster image will be shown. No parameters.

volume (percentage:Number)
Change the audio volume of the player to a certain percentage (e.g. *90*). If the player is muted, it will automatically be unmuted when a volume event is sent.

Note: Due to anti-phishing restrictions in the Adobe Flash runtime, it is not possible to enable/disable fullscreen playback of the player from javascript.

8.3.1 Loading playlists

The ***load*** event is mostly used for loading a single video or a single playlist into the player. This is done setting the second parameter of the event to the URL of the video:

```
player.sendEvent("load","http://www.mysite.com/video.mp4");
```

The player will inspect the URL, decide which *media format* or *playlist format* it is and setup its internal playlist accordingly.

It is also possible to load an already preformatted playlist in the player using this same **load** call. This is useful in situations where e.g. you want to create or manipulate the playlist in javascript.

The preformatted playlist should be an *array* that contains one or more *object* blocks. Each *object* block in turn contains multiple *playlistitem properties* as key:value pairs. Here are two examples of playlist loading; one with a single entry and one with multiple entries:

```
// Here we load a single-entry playlist.
// We set the video to RTMP streaming, plus we set the start position.
var list1 = new Array(
    { file:"video.mp4", type:"rtmp", start: 56, streamer: "rtmp://mysite.com/definst" }
);
player.sendEvent("load",list1);

// Here we load a full playlist.
// For each entry, we set a title.
var list2 = new Array(
    { file:"/static/bbb.mp4", type:"video", title: "Big Buck Bunny Trailer" },
    { file:"/static/ed.mp3", type:"sound", title: "Elephant's Dream Podcast" },
    { file:"/static/sintel.jpg", type:"image", title: "Sintel Movie Poster" }
);
player.sendEvent("load",list2);
```

Note: Both the **file** and the **type** properties are required for each playlistitem. If one of the two is missing, the playlistitem will not be loaded. The **type** property can be *video*, *sound*, *image*, *youtube*, *http* and *rtmp*.

8.4 Setting listeners

In order to let javascript respond to player updates, you can assign listener functions to various events the player fires. An example of such event is the *volume* one, when the volume of the player is changed. The player will call the listener function with one parameter, a *key:value* populated object that contains more info about the event.

Both the *Model* and the *Controller* of the player's *MVC structure* send events. You can subscribe to their events by resp. using the *addModelListener()* and *addControllerListener()* function. Here's a few examples:

```
function stateTracker(obj) {
   alert('the playback state is changed from '+obj.oldstate+' to '+obj.newstate);
};
player.addModelListener("state","stateTracker");

function volumeTracker(obj) {
   alert('the audio volume is changed to: '+obj.percentage'+ percent');
};
player.addControllerListener("volume","volumeTracker");
```

If you only need to listen to a certain event for a limited amount of time (or just once), use the *removeModelListener()* and removeControllerListener()* functions to unsubscribe your listener function. The syntax is exactly the same:

```
player.removeModelListener("state","stateTracker");
player.removeControllerListener("volume","volumeTracker");
```

Note: You MUST string representations of a function for the function parameter!

8.4.1 Model events

Here's an overview of all events the *Model* sends. Note that the data of every event contains the *id*, *version* and *client* parameters that are also sent on *playerReady()*.

error
Fired when a playback error occurs (e.g. when the video is not found or the stream is dropped). Data:

- *message* (String): the error message, e.g. *file not found* or *no suiteable playback codec found*.

loaded
Fired while the player is busy loading the currently playing media item. This event is never sent for *RTMP Streaming*, since that protocol does not preload content. Data:

- *loaded* (Number): the number of bytes of the media file that are currently loaded.
- *total* (Number): the total filesize of the media file, in bytes.
- *offset* (Number): the byte position of the media file at which loading started. This is always 0, except when using *HTTP Pseudostreaming*.

meta
Fired when metadata on the currently playing media file is received. The exact metadata that is sent with this event varies per individual media file. Here are some examples:

- *duration* (Number) : sent for *video*, *youtube*, *http* and *rtmp* media. In seconds.
- *height* (Number): sent for all media types, except for *youtube*. In pixels.
- *width* (Number): sent for all media types, except for *youtube*. In pixels.
- Codecs, framerate, seekpoints, channels: sent for *video*, *http* and *rtmp* media.

- TimedText, captions, cuepoints: additional metadata that is embedded at a certain position in the media file. Sent for *video*, *http* and *rtmp* media.
- ID3 info (genre, name, artist, track, year, comment): sent for MP3 files (the *sound* media type).

Note: Due to the *Crossdomain Security Restrictions* restrictions of Flash, you cannot load a ID3 data from an MP3 on one domain in a player on another domain. This issue can be circumvented by placing a *crossdomain.xml* file on the server that hosts your MP3s.

state
Fired when the playback state of the video changes. Data:

- *oldstate* ('IDLE','BUFFERING','PLAYING','PAUSED','COMPLETED'): the previous playback state.
- *newstate* ('IDLE','BUFFERING','PLAYING','PAUSED','COMPLETED'): the new playback state.

time
Fired when the playback position is changing (i.e. the media file is playing). It is fired with a resolution of 1/10 second, so there'll be a lot of events! Data:

- *duration* (Number): total duration of the media file in seconds, e.g. *150* for two and a half minutes.
- *position* (Number): current playback position in the file, in seconds.

8.4.2 Controller events

Here's an overview of all events the *Controller* sends. Note that the data of every event contains the *id*, *version* and *client* parameters that are also sent on *playerReady()*.

item
Fired when the player switches to a new playlist entry. The new item will immediately start playing. Data:

- *index* (Number): playlist index of the media file that starts playing.

mute
Fired when the player's audio is muted or unmuted. Data:

- *state* (Boolean): the new mute state. If *true*, the player is muted.

play
Fired when the player toggles playback (playing/paused). Data:

- *state* (Boolean): the new playback state. If *true*, the player plays. If *false*, the player pauses.

playlist
Fired when a new playlist (a single file is also pushed as a playlist!) has been loaded into the player. Data:

- *playlist* (Array): The new playlist. It has exactly the same structure as the return of the *getPlaylist()* call.

resize
Fired when the player is resized. This includes entering/leaving fullscreen mode. Data:

- *fullscreen* (Boolean): The new fullscreen state. If *true*, the player is in fullscreen.
- *height* (Number): The overall height of the player.
- *width* (Number): The overall width of the player.

seek
Fired when the player is seeking to a new position in the video/sound/image. Parameters:

- *position* (Number): the new position in the file, in seconds (e.g. *150* for two and a half minute).

stop
Fired when the player stops loading and playing. The playback state will turn to *IDLE* and the position of a video will be set to 0. No data.

volume
Fired when the volume level is changed. Data:

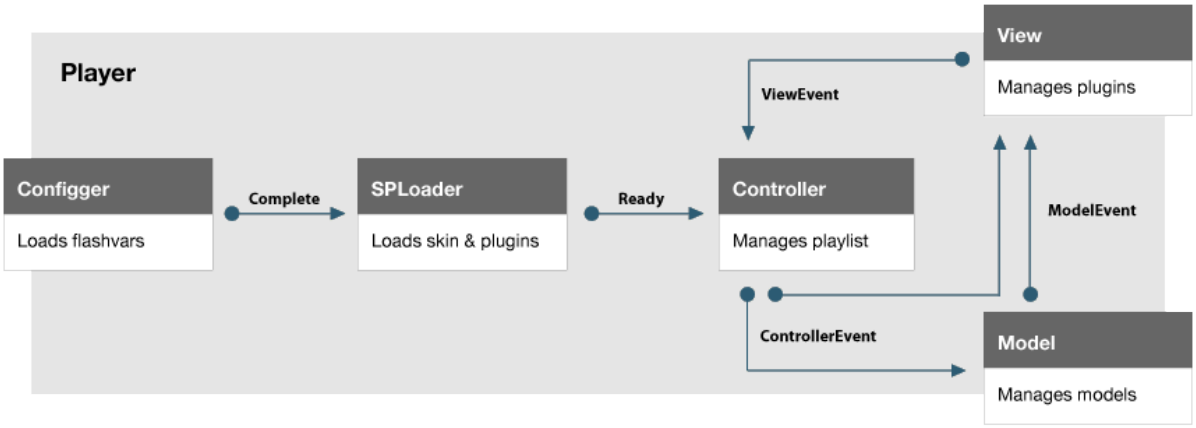
- *percentage* (Number): new volume percentage, from 0 to 100 (e.g. *90*).

CHAPTER

NINE

JW PLAYER ARCHITECTURE

This page gives a high level overview of the features and structure of the JW Player: its configuration through options, skinning and plugin loading, event structure, file loading and resizing process. Here's a simple schematic of the internal structure of the player:



9.1 Flashvars

First, all *Configuration Options* are loaded, by the **Configger** utility class. Options are used to change the layout and behaviour of the player. You also use options to load *files* or plugins into the player. Options are set by adding a *flashvars* attribute to the HTML code that embeds the player in your site.

9.2 Skinning

After the flashvars are loaded, the player proceeds to loading a skin, if you *have set one*. A skin is an external SWF file that provides a new look to the player. A number of skins is available as a free download in our addons repository.

9.3 Event structure

When the skin is loaded, the player will initialize its internal framework, the MVC triad. It splits up the functionality of the player in three sections:

The Model
It manages the playback of the videos and contains a string of classes that can each playback a specific *file* (e.g. *video*) or *rtmp* or *http* stream.

The View
It manages all user interfaces: the built-in components, the externally loaded plugins and the javascript API.

The Controller
It checks and forwards all directive from the *View* to the *Model*. It also manages the internal playlist, including such functionalities as shuffle and repeat.

9.4 Plugins

After the MVC framework is initialized, the plugins are being loaded (if the *plugins flashvar* is set). Plugins are standalone SWF files that add functionality to the player (e.g. for closed captioning or advertisements). There's a big number of plugins available in our repository.

Some plugins have their own flashvars (e.g. the *captions.file* flashvar for the *captions* plugin), to customize them or tell them which files to load. These flashvars (that always begin with the name of the plugin and a dot) were already loaded in step 1 of the player initialization, so plugins can immediately setup themselves.

9.5 PlayerReady

When all plugins are loaded the player is completely ready to start playback. At this point the player sends a ping to a *playerReady function* in javascript. When sent, the player is ready to receive events from and send events to javascript.

Next, the player will also send two events itself to get things started:

- The player will send a *LOAD event*, loading the video or playlist.
- The player sends a *RESIZE event*, so all plugins and components can resize themselves to the stage.

Both these events are explained in more detail below.

9.6 File loading

Because the JW Player handles a wide range of filetypes and formats, the file loading features a small decision list to determine which playback *model* to use:

1. First, the player checks if a *type option* is set (e.g. *type=rtmp*). If it is, the player loads the file and assigns the set playback type.
2. If there is no *type* option, the player looks at the extension of the *file* option. If it is a known media *format* (e.g. *.mp4**), the player will load the file and assigns the right playback type (e.g. *video*). If the extension is not a known media format (e.g. *.xml*), the player will presume the file is a playlist. It tries to load and parse the *playlist*.
3. When the playlist is loaded and parsed, the player repeats step 1 and 2 for every entry in the playlist. If any of these entries have no *type* option and no known extension, they are dropped.

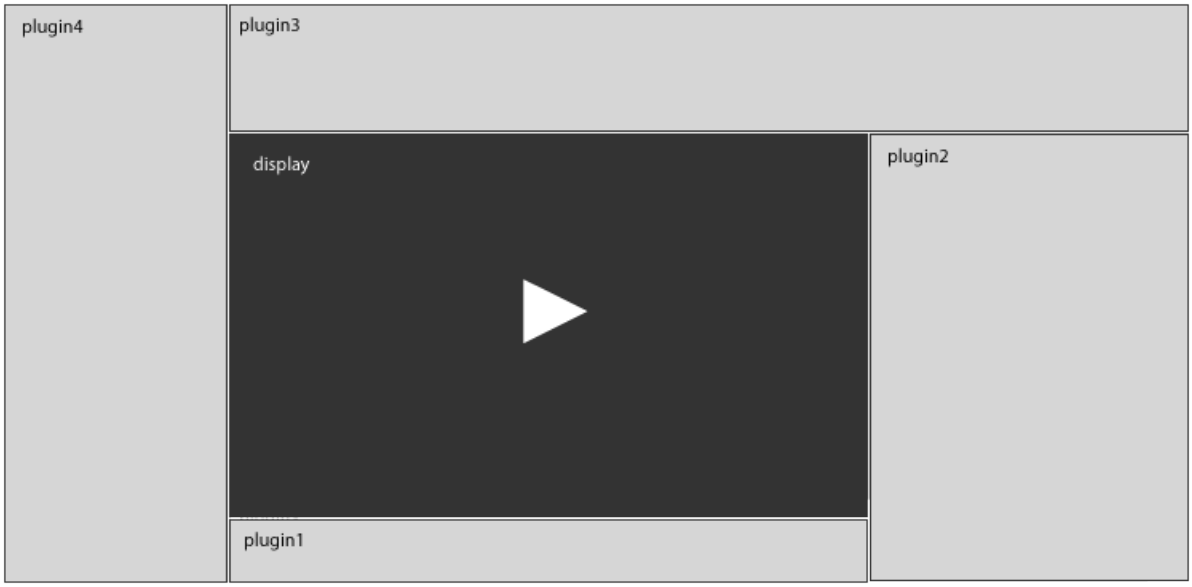
Next, if the file or playlist has loaded and the *autostart* option is turned on, the player will also immediately start playback.

9.7 Resizing

The JW Player resizes itself automatically to the dimensions of the Flash container in HTML. The resizing is managed like this:

- At the beginning of each resize operation, the *display* gets the entire stage.
- Next, the player walks through every plugin to see if it has a *size* and *position* flashvar set. For example, the *controlbar* has a default *bottom* position and *20* pixels size. The player adjusts the display dimensions for that.
- Some components (such as the *dock*) have their position set to *over*. The player will then simply set the dimensions of this plugin to match those of the display.
- When all component dimensions have been calculated, the player issues a *RESIZE event*. Next, it is up to each plugin then to position itself where the player wants it to be.

Here's an image that illustrates the resizing functionality. Next to the display, it contains for components/plugins that requested screen estate:



In fullscreen, the screen-division mechanism is not used. Instead, the ''display'' is given all screenspace. Only the components that are *over* the display will be visible. The *controlbar* will automatically be set to *over* in fullscreen mode.

CHAPTER

TEN

CROSSDOMAIN SECURITY RESTRICTIONS

The Adobe Flash Player contains a crossdomain security mechanism, similar to JavaScript's Cross-Site Scripting restrictions. Flash's security model denies certain operations on files that are loaded from a different domain than the *player.swf*. Roughly speaking, three basic operations are denied:

- Loading of data files (such as *Playlist Support* and the *config XML*).
- Loading of SWF files (such as *skins*).
- Accessing raw data of media files (such as *ID3 metadata*, sound waveform data or image bitmap data).

Generally, file loads (XML or SWF) will fail if there's no crossdomain access. Attempts to access or manipulate data (ID3, waveforms, bitmaps) will abort.

10.1 Crossdomain XML

Crossdomain security restrictions can be lifted by hosting a crossdomain.xml file on the server that contains the files. This file must be placed in the root of your (sub)domain, for example:

```
http://www.myserver.com/crossdomain.xml
http://videos.myserver.com/crossdomain.xml
```

Before the Flash Player attempts to load XML files, SWF files or raw data from any domain other than the one hosting the *player.swf*, it checks the remote site for the existence of such a *crossdomain.xml* file. If Flash finds it, and if the configuration permits external access of its data, then the data is loaded. The file is not loaded or the data is not shown.

10.1.1 Allow All Example

Here's an example of a *crossdomain.xml* that allows access to the domain's data from SWF files on any site:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="*" />
</cross-domain-policy>
```

Our *plugins.longtailvideo.com* domain includes such a crossdomain file, so players from any domain cal load the plugins hosted there.

Note that this example sets your server wide open. Any SWF file can load any data from your site, which might lead to sercurity issues.

10.1.2 Restrict Access Example

Here is another example *crossdomain.xml*, this time permitting SWF file access from only a number of domains:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="*.domain1.com"/>
  <allow-access-from domain="www.domain2.com"/>
</cross-domain-policy>
```

Note the use of the wildcard symbol: any subdomain from *domain1* can load data, whereas *domain2* is restricted to only the *www* subdomain.

Crossdomain policy files can even further finegrain access, e.g. to certain ports or HTTP headers. For a detailed overview, see Adobe's Crossdomain documentation.