Table of Contents

[The core concept 2](#_Toc295078245)

[Core design concepts of MediaChrome 3](#_Toc295078246)

[API Reference 4](#_Toc295078247)

[Song 4](#_Toc295078248)

[Important methods 4](#_Toc295078249)

[Artist 6](#_Toc295078250)

[Album 7](#_Toc295078251)

[IPlayEngine 8](#_Toc295078252)

[Properties 9](#_Toc295078253)

[Methods 11](#_Toc295078254)

[View Subsystem 17](#_Toc295078255)

[Case study 19](#_Toc295078256)

[*P* 19](#_Toc295078257)

[Existing solutions 19](#_Toc295078258)

[Why this solution? 19](#_Toc295078259)

[Fundamentals 20](#_Toc295078260)

[Layer 1: Preprocess 20](#_Toc295078261)

[Layer 2: Layout Elements 20](#_Toc295078262)

[Layer 3: Rendering Elements 20](#_Toc295078263)

[Active Rendering Layer 22](#_Toc295078264)

[1.2 Feed syndication 25](#_Toc295078265)

[View inflation 26](#_Toc295078266)

[Example 26](#_Toc295078267)

[Armature elements 28](#_Toc295078268)

[<view> 28](#_Toc295078269)

[<section> 28](#_Toc295078270)

[Layout elements 29](#_Toc295078271)

[<label> 29](#_Toc295078272)

[<img> 29](#_Toc295078273)

[<entry> 30](#_Toc295078274)

[<columnheader> 30](#_Toc295078275)

[<button> 31](#_Toc295078276)

[Code lifecycle 32](#_Toc295078277)

# The core concept

MediaChrome (abbreviated MC) is an framework and application aiming to provide a highly dynamic, extensive music service consumer application. MediaChrome architecture is based on an concept where the application acts as an adapter between music services, social networks and 3rd party plugin.

## Core design concepts of MediaChrome

* **One application, many services.** The avaibility ofmusic services and individual songs may change quickly. People today consuming music on many sources simultany, since track A is only available on Youtube, and can thus not be put on an playlist with music only available on Spotify. **MediaChrome** aims to workaround this by an **abstract media location system and IPlayEngine. Mediachrome makes a division between the physical location of an media file and playlist storage.**
* **Extensibility.** Regardless of the consumer service, people tend to create mashups dealing with services. MediaChrome architecture an abstraction layer (like Windows HAL) between the plugins (user scripts in views), social networks and music services. Thus, all music service compatible with MC will be able to interfere with existing user plugins today without any modification.

**The golden rule: Scripts should never request info from any music service/social network directly, it should always use MC’s wrappers to be dynamic.**

# API Reference

## Song

The song object is an representation of an song from any source. MC intend to let user **to seamless transfer song(s) between service** without loosing metadata. The first service lookup the song from it’s mapper (uris) by it’s internal API, and then create an instance of **Song** with the metadata applied. Then this instance is sent to the target service, which extract the information and recreate the song by looking up an apporiate song with similar metadata provided with the first instance, without knowing how service 1 works internally. This will let people to **transfer playlists between two different music service** and retain all songs whose available on the target service intact.

## Important methods

This functionality is uilitied by some of the vital functions that each service must implement in their derivation of IPlayEngine:

**List<Song>** RawFind(**Song** Song)

The code in this method should ask it’s service to lookup an song from it’s database by the metadata (title,artist,album) in the argument, and then return an new Song instance of the matched song, along with an corrensponding URI matching the target service.

**Song** ConvertSongFromLink(String URI)

MC pass an song URI which corresponds to an playable song in the service’s database, and expect to get an abstract Song back with it’s metadata for usage on another service. The programmer is obligated to extract the correct song from the URI according to the service’s system and then return an Song instance providing at least artist/album/title metadata so MC can pass it to another service seamless and recreate the song on the new service indepently of the first service.

|  |  |  |
| --- | --- | --- |
| Type | Method/Member | Description |
| String | Title | Title of the song |
| Artist | Artist | The artist of the song |
| Album | Album | The album of the song |
| String | AlbumName | An reference to Album’s name for simplicity. |
| IPlayEngine | Engine | The engine the song was derived from(spotify youtube) |
| String | Path | The path of the song, assigned to the service behind the engine. |
| String | ArtistURL | Used within mediachrome’s view system to create alias |
| String | Version | Some songs can provide version as an separate field. |
| String | Feature | Some songs can provide the featuring artist in an separate field |
| String | Contributing | Some songs can provide the contributing artist in an separate field |
| String | Composer | The composer of the artist |
| Float | Popularity | The popularity of the song |
| String | ID | The Identifier of the song, specific for the Engine song is residing on |
| String | Link | An link to the song |
|  |  |  |

## Artist

An artist object represents an artist, whose contains an list of Album objects.

|  |  |  |
| --- | --- | --- |
| Type | Method/Member | Description |
| Album[] | Albums | An array of album instances representing all albums the artist has |
| String | Link | The MC link to the artist |
| Name | Name | The name of the Artist |
| IPlayEngine | Engine | The engine the song was derived from(spotify youtube) |

## Album

Represents an album of the song.

|  |  |  |
| --- | --- | --- |
| Type | Method/Member | Description |
| Song [] | Songs | An array of all songs of an album |
| Artist | Artist | The artist of the album |
| String | Link | The link to the album |
| IPlayEngine | Engine | The engine the song was derived from(spotify youtube) |

## IPlayEngine

To make MC something useful, it uses media sources which implements a interface called “IPlayEngine”. It contains methods, which all must be implemented in an correct way in order to make a reliable system.

Any media source is installed locally as an .NET assembly written in any .NET language compiled to CIL and contains executable code organized by the interface specification to deal with an particular music service. The assembly may make use of API of other service which provides MediaChrome with the service’s functionality from MediaChromes media consumers.

### Properties

|  |  |  |
| --- | --- | --- |
| Type | Method/Member | Description |
| Song | CurrentSong | Gets the currently playing song |
| String | AudioSignature | Returns the base for the song urls (eg. spotify:track:\*) |
| String | Copyright | Returns the copyright holder of the assembly for the current implementation |
| String | Address | The address to the company issue the implementation |
| String | Company | Tells which company who has written the assembly |
| Uri | CompanyWebSite | The address to the company’s web site. |
| Uri | ServiceProvider | The URI to the service provider |
| Bool | LoggedIn | Gets if the user has logged in to the particular service |
| Bool | Streaming | Tells if music can be streamed from this source |
| Bool | DownloadStore | Tells if music can be purchased from the service. |
| List<Song> | Purchases | Returns a list of Songs of the purchases the logged in user has done |
| String | Address to an image which represents the logo of the service. | Returns the logo of the service. |
| System.Drawing.Icon | Icon | An reference to an Icon which are a graphical logo of the service. Uses to identify the service from the MC UI |
| String | Status | The status of the service right now. Should show “Loading” etc. if the service is trying to load an media. |
| Control | MediaControl | An Control for the service to provide service-specific integration with the user not able for other plugins to handle from other sides of MC system, such addons etc for th service. |
| Double | Duration | Duration of the current loaded media in double (measured in seconds) |
| Int | Position | Returns the position of the media right now |
| Bool | PlaylistLoaded | Returns if the playlist has been loaded |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### Methods

|  |  |  |
| --- | --- | --- |
| Return type | Signature |  |
| Void | Pause() | Pauses the current playing media. |
| Void | Load(**string** uri) | Load the media from the specified URI |
| Void | Play() | Play the media |
| Void | Stop() | Stop the media |
| Bool | Purchase(Song song) | Purchase an song from the provider, specified with the Song argument. Should return TRUE if success, false if failed. |
| Song | ConvertSongFromLink(string uri) | Lookup the song behind the service-specific URI (spotify:track:xxx for example) and convert it into an abstract representation instance of Song, so other media services can recreate the song. |
|  |  |  |

/// <summary>

/// Convert an URI to an instance of Song Object. Used in conjunction with AudioSignature

/// </summary>

/// <param name="URI">The resource URI</param>

/// <returns>A Song with properties beloning to the resource behind the URI</returns>

Song ConvertSongFromLink(String URI);

/// <summary>

/// Shows options dialogue.

/// </summary>

void ShowOptions();

/// <summary>

/// The company providing the service

/// </summary>

String Company { get; set; }

/// <summary>

/// The url to the company's website

/// </summary>

Uri CompanyWebSite { get; set; }

/// <summary>

/// The web resource of the service

/// </summary>

Uri ServiceUri { get; set; }

/// <summary>

/// Handles purchases of an song

/// </summary>

/// <param name="song">A song</param>

/// <remarks>Not implemented</remarks>

/// <returns>Whether the purchase was accepted or rejected by the merchant</returns>

bool Purchase(Song song);

/// <summary>

/// Gets and sets whether the user is logged in to the service

/// </summary>

bool LoggedIn { get; set; }

/// <summary>

/// Show the log in creditals. Log in is handled by each service

/// </summary>

void Login();

/// <summary>

/// Defines whether the music is streamed from an internet source

/// </summary>

/// <remarks>Not implemented</remarks>

bool Streaming { get; }

/// <summary>

/// Defines if purchases is available at the service

/// </summary>

/// <remarks>Not implemented</remarks>

///

bool DownloadStore { get; }

/// <summary>

/// Returns a list of songs ready for purchase

/// </summary>

List<Song> Purchases { get; }

/// <summary>

/// Method for logging out the user from the service

/// </summary>

void Logout();

/// <summary>

/// Image representing the service

/// </summary>

string Image

{

get;

}

/// <summary>

/// The icon for the Engine

/// </summary>

System.Drawing.Image Icon { get; }

/// <summary>

/// Get an artist by ID

/// </summary>

/// <param name="ID">The ID or name of the artist to retrieve, depending on the service</param>

/// <returns>An artist enclosed in an instance of MediaChrome's Artist class</returns>

Artist GetArtist(string ID);

/// <summary>

/// Find an artist

/// </summary>

/// <param name="Query">The search query.</param>

/// <returns>Return an array of the founded artist at the engine</returns>

Artist[] FindArtist(string Query);

/// <summary>

/// Get an album by artist

/// </summary>

/// <param name="artist">The artist for the query</param>

/// <param name="album">The name or ID of the album to locate, depending on the service</param>

/// <returns>An instance of the album enclosed into an Album class instance</returns>

/// <remarks>Used in conjunction with views frameworks</remarks>

Album GetAlbum(Artist artist, string album);

/// <summary>

/// Get an album

/// </summary>

/// <param name="album">The name or ID of the album to locate, depending on the service</param>

/// <returns>An instance of the album enclosed into an Album class instance</returns>

/// <remarks>Used in conjunction with views frameworks</remarks>

Album GetAlbum(string album);

/// <summary>

/// Find an album

/// </summary>

/// <param name="album"></param>

/// <returns></returns>

Album[] FindAlbum(string album);

event EventHandler PlaybackFinished;

/// <summary>

/// Gets whether the engine can handle playlists

/// </summary>

bool hasPlaylists {get;}

/// <summary>

/// The text status of the engine

/// </summary>

string Status { get; set; }

/// <summary>

/// Visible control for the specific engine

/// </summary>

Control MediaControl {get;}

/// <summary>

/// Denotes if the playback is ready

/// </summary>

bool Ready {get;set;}

/// <summary>

/// Duration of the current song

/// </summary>

double Duration { get; }

/// <summary>

/// The amount of files complete in an import

/// </summary>

int FilesCompleted {get;set;}

/// <summary>

/// Gets and sets if the playlists has been loaded

/// </summary>

bool PlaylistsLoaded { get; set; }

/// <summary>

/// Namespace of the IPlayEngine. Used for as an arbitrary

/// identifier for the engine. Must not contain any whitespaces

/// and not be modified at any point at the application cycle.

/// </summary>

string Namespace {get;}

/// <summary>

/// Title of the engine. Must not be modified upon runtime

/// </summary>

string Title {get;}

/// <summary>

/// Total local files

/// </summary>

int TotalFiles {get;set;}

/// <summary>

/// Query for songs on the service cloud

/// </summary>

/// <param name="Query">The textual query</param>

/// <returns></returns>

List<Song> Find(String Query);

/// <summary>

/// Import songs into the local library according to the specifications.

/// </summary>

/// <param name="songs"></param>

void SongImport(Song[] songs);

/// <summary>

/// Starts playing the current song

/// </summary>

void Play();

/// <summary>

/// The position of the song

/// </summary>

int Position { get; }

/// <summary>

/// Occurs when pausing

/// </summary>

void Pause();

/// <summary>

/// Occurs when stopping

/// </summary>

void Stop();

/// <summary>

/// Occurs when seeking

/// </summary>

/// <param name="pos">The new position of the media</param>

void Seek(double pos);

/// <summary>

/// Occurs when loading an song

/// </summary>

/// <param name="Content">The data uri associated with the song</param>

void Load(String Content);

/// <summary>

/// Import music to the local database

/// </summary>

/// <param name="Conn">SQLiteConnection instance for accessing the internal database</param>

/// <param name="RootDir">The file directory for the local files to recurse on</param>

List<Song> Import(string RootDir);

/// <summary>

/// Host form. Used by the runtime

/// </summary>

System.Windows.Forms.Form Host{get;set;}

/// <summary>

/// Unload the engine

/// </summary>

void Unload();

/// <summary>

/// Raw search songs from the service

/// </summary>

/// <returns>An list of songs</returns>

List<Song> Search();

/// <summary>

/// Used to do an fast extraction of a song instance from an song query

/// </summary>

/// <param name="\_Song">instance of song class without a real connection

/// to an existing service instance</param>

/// <returns></returns>

Song RawFind(Song \_Song);

//string RawFind(Song \_Song);

/// <summary>

/// Returns a list of playlists from the current service

/// </summary>

List<MediaChrome.Views.Playlist> Playlists {get;}

/// <summary>

/// Playlist-related functionality

/// </summary>

/// <returns>A instance of the playlist class with content from this source</returns>

Playlist ViewPlaylist(string Name,String PlsID);

/// <summary>

/// Creates an new playlist on the source.

/// </summary>

/// <param name="Name">The desired name of the playlist</param>

/// <returns>An instance of an playlist class representing the new playlist.</returns>

Playlist CreatePlaylist(String Name);

/// <summary>

/// Add an song to an playlist

/// </summary>

/// <param name="playlistID">The ID of the target playlist on the source</param>

/// <param name="\_Song">The instance of the song</param>

/// <param name="pos">The desired position on the playlist to add on. -1 indicates the last position</param>

void AddToPlaylist(string playlistID,MediaChrome.Song \_Song,int pos);

/// <summary>

/// Add an song to an playlist

/// </summary>

/// <param name="playlistID">The ID of the target playlist on the source</param>

/// <param name="\_Song">The instance of the song</param>

/// <param name="pos">The desired position on the playlist to add on. -1 indicates the last position</param>

void AddToPlaylist(Playlist pls, MediaChrome.Song \_Song, int pos);

/// <summary>

/// Remove a song at the specified index on the playlist.

/// </summary>

/// <param name="playlistID">The ID of the playlist, specific for the service</param>

/// <param name="pos">The index of the song which should be removed</param>

void RemoveFromPlaylist(string playlistID,int pos);

/// <summary>

/// Moves an entry through the playlist, the collection between the starting and end index.

/// </summary>

/// <param name="playlistID">The ID of the playlist</param>

/// <param name="entry">The Song to move</param>

/// <param name="startLoc">start index of the chunk to move</param>

/// <param name="endLoc">end segment of cnhunk</param>

void MoveSongPlaylist(string playlistID,Song entry,int startLoc,int endLoc);

// bool canSync {get;}

// void LoadSynchronized();

// bool OfflineService {get;}

/// <summary>

/// Returns the duration of the current song (in seconds)

/// </summary>

string Length { get; }

/// <summary>

/// Load an playlist and return the list. Obsolote.

/// </summary>

/// <param name="p"></param>

/// <param name="playlist"></param>

/// <returns></returns>

List<Song> LoadPlaylist(string p,ref Playlist playlist);

}

# View Subsystem

The view subsystem is an dynamic interface system to allow 3rd party to create mashups for MediaChrome.

## Case study

#### Purpose

The purpose of the Segurify view system is to create an streamlined, dynamic application user interface rendering engine.

### Existing solutions

A number of solutions are already existing for the purpose, which most are working at the w3 standard. These view system are based on an HTTP request layer and are them all not suitable for modern web application where the interface are regulary update in accordance with the changes in the content. Among such changes include adding items to lists, and the common solution for this is the well known and used Ajax solution.

### Why this solution?

We need an totally new solution for dynamic interface that are able to meet the common expections today of modern web application but at the same time has an limited effort at the computer. Our viewpoint is that the common layers has become to complex for webmasters, coders, authors to manage. To many aspects and layers are involved to create the end user experience of the web. For example, many primitive tasks has to be make possible with dealing with to many technologies together in an complex situations.

Even the most simple apporoach at the web, such managing blog posts and text form submission involves today an decent and complex relations between a wide amount of technology pardigams, which all has slightly different behavior, and make it slightly difficult to support and the variety of code and technology standard among the platforms (HTTP, javascript, HTML/XMKL, system) making primitive problems slightly difficult and time consuming to solve.

## Fundamentals

The Segurify rendering engine is divided into three levels.

Active Rendering Engine

Rendering Elements

Layout Markup

Preprocessor

User experience

### Layer 1: Preprocess

The preprocessing layer is compiling the view’s conditions. It are done by an bitwise preprocessing protocol which supports any preprocessing languages implementing an interface class called IScriptEngine.

The preprocessing is done **locally and not on the server.** The reason is to enasure the user has full control over the pre-rendering state. The purpose of preprocessor is to compile a set of layout markup which can make the layout rules.

**Preprocessing is one the client because these reasons:**

* **Client customization.** The preprocessor can customize the behavior in accordance with the client state, access by secured permissive system (measured for each app). This make interaction easier to manage.
* **Separate user interface and semantic information!** We propose web application should do the semantic communication with the client by feeds such Web Services, RSS or XML schemes, and not as with the todays sent of HTML. This will reduce costs for mobile users slightly and make an reduction of amount data traffic usage.

### Layer 2: Layout Elements

The preprocessor is rendering an preliminary set of XML elements called “Layout Elements”. It consists of an concept we’re familiar today, and has some characteristic found in XUL/HTML. The layout elements define the user experience.

### Layer 3: Rendering Elements

The next phase is the Rendering Elements. The Layout Elements are converted into Rendering Elements upon render time. The rendering elements has their distinct role at the end execution and will define how the layout will behave. The rendering elements are also more complex, and the reason to split up the rendering system to an additional abstraction layer is to make consistency of the rendering engine.

For example, rendering elements are specifying explicit how to behave and multiple rendering elements can together be a an result of an single layout element. Developers can also create new sets of rendering elements to custom Layout Elements but also inflate other sets of layout elements into another Layout and make them behave unique by instance parameters (xml attributes).

Another important aspect is that while Layout Elements has an specified set of attributes, rendering elements have an dictionary of arbitrary properties that can apply to all elements.

### Active Rendering Layer

The rendering engine is active in maintain state after loading. The engine will look for updated conditions and update all rendering elements with new properties or add elements previous not there by recurring interval.

The engine uses an synchronized buffer. When the runtime has downloaded and parsed an updated view, it associate it

Spotify Ultra uses a distinct View where all view consists of a mako like template. These template can be used to tweek the view to an particular query and all view is defining an particular namespace.

An preprocessing statement begins with an % throughout line until the line end or between <? ?> blocks. The preprocessing iself is done by the javascript engine jint. All views are an xml file divided into so called “section” statements and inside them resides some special kind of html like content.

<?

// Our classical statement hello world

var output = “Hello World!”;

// Repeat this five time

var count\_repeat = 5;

?>

<view>

<section name=”Overview”>

<text left=”20” top=”30”>@{output}</text>

% for(var i=0; i < count\_repeat; i++){

<text left=”20” top=”@TOP”>Item @{i}</text>

% }

</section>

</view>

Will generate this output

<view>

<section name=”Overview”>

<text left=”20” top=”30”>Hello World!</text>

<text left=”20” top=”@TOP”>Item 0</text>

<text left=”20” top=”@TOP”>Item 1</text>

<text left=”20” top=”@TOP”>Item 2</text>

<text left=”20” top=”@TOP”>Item 3</text>

</section>

</view>

## Feed syndication

The advantage is to be able to compile JSON statements directly into the asserted view with the function synchronize\_data(str json). We have an feed (feed.json) with following content on our wwwhost root at localhost:

[**http://localhost/feed.json**](http://localhost/feed.json)

{

"items":[

{

"title":"Hello World",

"author":"By Me",

},

],

]

View’s source code

<?

// Our classical statement hello world

var data = synchronize\_data(“http://localhost/feed.json”);

?>

<view>

<section name=”Overview”>

<text left=”20” top=”30”>@{output}</text>

% for(var i=0; i < data.feed.length; i++){ var item = data.feed[i];

<text left=”20” top=”@TOP”> @{item.title} by @{item.author}</text>

% }

</section>

</view>

When ready it looks like

<view>

<section name=”Overview”>

<text left=”20” top=”30”>@{output}</text>

<text left=”20” top=”@TOP”> Hello world by Me</text>

% }

</section>

</view>

## View inflation

Views can be used as component through an inflation system. As many items are used repeatedly, if there are some errors it could be exhausting to edit all files. Therefore there is an feature called “View inflation”.

This feature will allow the developer to develop components as own views, and then reuse the sub view. These components can have their own **parameter** which are set by the parent object by their attributes.

### Example

Album view is a common way to view songs. We put the album piece in own view, called album.xml and call this the album component. All xml attributes specified by the parent code is provided as an variable **arg\_{parameter}** for the child code. Notice this preprocessing is happening after the host preprocessing, so code between the host view and the inflated preprocessing **can not be shared.** The design reason of this is because those views should be treated as **components** and thus be independent of what context they’re put on. However @{variables} can define xml attributes for the child view before it are executed.

Album.xml

<?

/\*\*\*

\* Album view component. Not to use separetely

\*\*/

var album\_name = arg\_name;

var track\_count = arg\_track\_count;

?>

<view>

<section name="component">

<label title="@{album\_name}" height="23" top="@top" width="0"/>

<img src="http://farm1.static.flickr.com/2/1703693\_0412c29a4f\_m.jpg" width="128" height="128" top="@TOP" persistent="true"/>

<!-- An example tracklist -->

% for(var i=0; i < 10; i++){

<entry left="150" top="@TOP" height="18" author="Test" title="Item @{i}" collection="Test" album="test"/>

% }

</section>

</view>

This view is then put together in this file, artist view example:

<?

/\*\*

This is the view for the main view of Spotify Ultra

\*/

var current\_user = "drsounds";

var testdata = synchronize\_data("http://krakelin.com/sync.json");

?>

<view>

<section title="Overview">

<h1>Welcome to SpotifyUltra </h1>

<label left="20" top="@TOP" width="100" height="60">Welcome to Spotify Ultra. User is @{current\_user} </label>

<!-- Recent activity -->

<header width="320" height="20" top="@top" >Entries</header>

% for(var i=0; i < 10; i++){

**<inflate src="components/album.xml" track\_count="5" name="Album @{i} t"/>**

% }

</section>

</view>

Note the bold text. It are the inflation block. Pay attention to the concept where the attributes of the element is relative to the arg\_ variables we seen on previous page.

## Armature elements

Some elements is an armature for the view

### <view>

The root element. Should contains one or several **<section>** element defining an section of an view.

### <section>

<view>

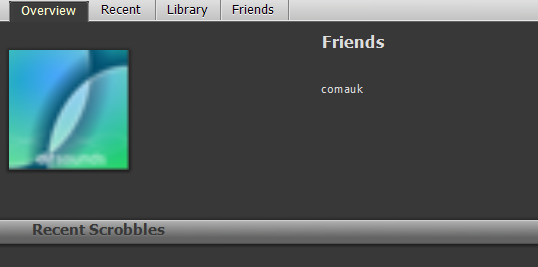
<section name=”Overview”>…</section>

<section name=”Recent”>…</section>

<section name=”Library>…</section>

<section name=”Friends”>…</section>

</view>



Defines an section of the view. All section can consist of an playlist.

|  |  |  |
| --- | --- | --- |
| Attribute | description | Default |
| Name :string | The name of the section | Overview |
| List :bool | Specifies if the section should have visible columnheaders. | false |

## Layout elements

Layout elements are used inside <section> elements to create interaction with the user.

### <label>

A label is used for outputting an string.

<label top=”@TOP” left=”10” height=”18”>Text</label>

|  |  |  |
| --- | --- | --- |
| Attribute | description | Default |
| Top | The y location of the label | @TOP |
| Left | The x location of the label | 0 |
| Width | The width of the label | 100 |
| Height | Height of the label | 10 |
| *Data* | The data of the label |  |
| Font | Typeface of the font |  |
| Size | Size of the string in pixels |  |
| Color | Color of the label |  |
|  |  |  |

### <img>

Used for displaying an picture.

<img src=http://source/coolimage.png" width=”320” height=”320” left=”10” top=”@TOP” />

**Note:** for consistency <image> tag is an substitute for the <img> tag

|  |  |  |
| --- | --- | --- |
| Attribute | description | Default value |
| Top | The y location of the image | @TOP |
| Left | The x location of the left | 0 |
| Width | The width of the label | 100 |
| Height | Height of the label | 10 |
| Shadow | Indicating if the image should have an drop shadow | False |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### <entry>

An playlist entry pointing to an audiotory resource for use with MediaChrome. Any view with an group of entry is considered as an **playlist** within **MediaChrome.** The **noelm** attribute must be set to true on all sub-level properties to prevent interfering with the height of the element.

<entry top=”@TOP” left=”10” width=”-1” height=”18”>

<title noelm=”true”>Title</title>

<artist noelm=”true”>Artist</artist>

</entry>

|  |  |  |
| --- | --- | --- |
| Attribute | description | Default value |
| Top | The y location of the entry | @TOP |
| Left | The x location of the entry | 0 |
| Width | The width of the entry | 100 |
| Height | Height of the entry | 10 |
| Uri | The media resource pointing on |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### <columnheader>

<columnheader size=”110” name=”artist”>

Columnheaders is mandatory when dealing with entries. Entries are labeled by implementing attributes or child elements named after the columnheaders. They are used in the order they were represented in markup.

|  |  |  |
| --- | --- | --- |
| Attribute | description | Default/proposed value |
| Size | The size of the column | 10 |
| Title | The title of the column | Title |

**Note:** You should always put an columnheader with the size=10 and name=” “ to make a space for the little speaker icon.

### <button>

Defines an push\_button. Used for extraordinary interaction with the user.

|  |  |  |
| --- | --- | --- |
| Attribute | description | Default/proposed value |
| Top | The y location of the entry | @TOP |
| Left | The x location of the entry | 0 |
| Width | The width of the entry | 100 |
| Height | Height of the entry | 10 |
| Onclick | Code to raise when clicking | *none* |
|  |  |  |

## Code lifecycle

In differ to other markup frameworks, the runtime stack where the preprocessor code was executed for rendering the view has not been disposed when the output was ready. Functions defined inside the preprocessor directive are sleeping, not dead and can be raised through calls by various events around the markup.

# Callbacks

Every view can accept some callbacks that are called from the presentation shell. These are:

**String** list\_delete(**string** uri)

Invokes when an user request an entry to be deleted. If the entry deletion is accepted, it should return “TRUE” otherwise “FALSE”.

# Method reference

Full method reference is provided in the excel file following the MediaChrome docs

# Skinning

MediaChrome supports skins throughout the application. A skin consists an folder with a set of bitmaps and an xml file called <skinName>.xml with a set of tags. All skins must use the xml namespace.

All skins are placed in an directory called ‘Skins’ in the application’s root folder. They should not be compressed in any way. An skin placed in the correct folder can be choosed at the preferences at the skin selection combobox.

**Sample skin ‘Spotify’s skin.xml**

<?xml version="1.0" encoding="ISO-8859-1"?>

<skin xmlns="http://skin.mediachrome.net/2011/skin-definition">

<title>Spotify skin</title>

<component rel="Board" BackColor="#383838" ForeColor="#EEEEEE" />

<component rel="Board#Alternative" BackColor="#2c2c2c" ForeColor="#aaaaaa" />

<component rel="Board#Selection" BackColor="#aaeeaa" ForeColor="#003300" />

<component rel="Board#Active" BackColor="#111111" ForeColor="#AAFFAA" />

<component rel="Board#Section" BackColor="#EEEEEE" ForeColor="#444444" />

<component rel="Board#TextFade" BackColor="#eeeeff" ForeColor="#333388" />

<component rel="Header" BackColor="#aaaaaa" BackgroundImage="~\top\_wall1.png" />

<component rel="Footer" BackColor="#aaaaaa" ForeColor="#000000" BackgroundImage="~\top\_wall1.png" />

<component rel="Toolbar" SeparatorImage="~\tab\_separator.png" BackColor="#aaaaaa" ForeCOlor="#FFFFFF" BackgroundImage="~\toolbar.png" />

<component rel="ActiveSection" BackColor="#aaaaaa" ForeColor="#FFFFFF" BackgroundImage="~\section.png" />

</skin>

## Tags

### <component>

Is an selector for an component class.

<component rel="Board#Alternative" BackColor="#2c2c2c" ForeColor="#aaaaaa" />

|  |  |  |
| --- | --- | --- |
| Attribute | description | Default value |
| Rel | The selector of elements to be skinned | <element>#<subelement> |
| BackColor | Background color of the element | 0 |
| ForeColor | Foreground (text color) of an element | 100 |
| BackgroundImage | Bitmap background of the element |  |
| SeparatorImage | Some elements uses the SeparatorImageBitmap to draw sepearators |  |

## Selectors used by mediachrome

~ is substituted in paths for the current skin directory.

|  |  |
| --- | --- |
| Selector | Description |
| Board | Color for the view area of the application |
| Board#Alternate | Style for the alternate entries on the views |
| Board#Section | Style for the active section tab |
| Board#Active | Style for an active (playing) entry |
| Header | Style for the header of the application |
| Footer | Style for the lower bar of the application |
| Toolbar | Style for the toolbar section of the view |
| Board#TextFade | Text fade for some element |

*NOTE: This text covers an feature not yet implemented, it are an proposed way to enabled end-users to customize MediaChrome’s functionality.*

# Installing and managing different music service

Music services in MediaChrome is managed by .NET assembling which contains an reference to MediaChrome core assemblies and an class implementing the “IPlayEngine” interface.

The engines are located in an folder named by the engine’s namespace inside an folder on C:\ root called MediaProviders.

Each mediaProvider (Media Engines) has an assembly, and own .NET bundles.

In each engine folder, there is an XML called ‘<namespace>.xml’ which contains important info about the engine. Sample xml file is here:

<?xml version="1.0" encoding="ISO-8859-1"?>

<engine type=”streaming”>

<title>Mp3</title>

<namespace>mp3</namespace>

<description><![CDATA[ Built in engine for MediaChrome, which allows playback of local MP3s]]></description>

<author>MediaChrome team</author>

<targetVersion>0.6</targetVersion>

</engine>

The target version element assigns which build of MediaChrome the plugin is compiled for. The type attribute on the root element defines an multi-valued enum of what type of consumtion the service allow. Valid types is streaming|purchasing. Currently only streaming services is planned, but this is for maintaining further implementations.

## Web repository

Engines can be downloaded from an web repository. An repository consist as on the harddrive, of an folder of folders containing the engines. They are fetched by FTP. In this example, we call an fictional music service “qmusic” to provide their functionality into MediaChrome:

Example:

ftp://ftp.mc.qmusic.com/

Lists available engines:

Qmusic/ 10,3 MB

qmusic.dll 10,1 MB r---

qmusic.xml 200 KB r---

The user selects the FTP address as the repository in the apporiate “Add service “ which lists up the service into an window, where he/she can select to install the service into MediaChrome. He can also decide to uninstall it and the user should be warned that service may contain malicious code and only trust serius sources. The xml file is used to present the user an description of the service before he/she installs it to his computer.

## Installing from HTTP repositories

An root folder of an http repository consists of an XML file called ‘mc\_repository’: