Plex Plug-in Framework Documentation

Release 2.1.1

Plex, Inc.

May 25, 2011

CONTENTS

CHAPTER

ONE

INTRODUCTION

The Plex Plug-in Framework provides a simple yet powerful platform that enables developers to easily create plug-ins for Plex Media Server. The framework consists of an API containing a set of tightly integrated methods, and a small runtime environment that manages the execution of a plug-in's code.

The framework API builds upon a great foundation provided by the Python programming language, and concentrates on making the areas of Python most frequently used when developing plug-ins even easier to access. The API handles a lot of the boilerplate code internally, allowing the developer to focus on functionality, and the runtime environment manages most of the interaction between plug-ins and the media server.

Plug-ins created using the Plex Plug-in Framework are typically very small and lightweight, and can achieve a great deal of functionality in hardly any time. You'll be amazed by what you can accomplish!

CHAPTER

TWO

HOW PLUG-INS WORK

Each media server plug-in runs in its' own process, in a separate Python instance. This ensures that plug-ins can't interfere with each other, or with the operation of the server. Plug-ins communicate with the media server over sockets and act like mini HTTP servers, receiving standard HTTP requests and responding with XML-formatted data. This process is completely transparent to the plug-in developer - the framework will handle receiving requests, routing them to the correct place within a plug-in, and sending a response back to the server.

Plug-in's don't behave quite like normal Python programs. A plug-in isn't a program in its' own right - the framework's runtime environment is the program, which dynamically loads the plug-in code as part of the initialization routine. The runtime handles common tasks like run loop management, request handling, logging and data storage, and calls specific functions in the plug-in when necessary.

2.1 Differences between plug-ins and regular Python programs

Because media server plug-ins are a little different to regular Python programs, they can behave strangely in certain situations. There are a few key areas where regular Python rules do not apply that you should be aware of.

2.1.1 Code structure

Because plug-ins are executed within the framework's runtime environment, the code structure is more strict than in a normal Python program. All of the plug-in's code needs to be contained within functions. Although it is generally safe to initialize simple variables with types like `str` (http://docs.python.org/library/functions.html#str), `int` (http://docs.python.org/library/functions.html#int) or `bool` (http://docs.python.org/library/functions.html#bool) outside a function, anything more complicated (especially involving framework functions) may produce unexpected results or fail completely. This is because the runtime environment needs to be fully initialized before using any of the framework methods, and code in plug-ins that exists outside a function is executed as soon as the plug-in is loaded. Any code you want to execute when the plug-in

loads should be put inside the `Start()` (page 13) function. More information can be found in the *functions* section.

2.1.2 The framework APIs

One of the biggest hurdles for existing Python developers to overcome is breaking old habits when it comes to developing plug-ins. For instance, many developers may already be familiar with using the `urllib` (http://docs.python.org/library/urllib.html#module-urllib) and `urllib2` (http://docs.python.org/library/urllib2.html#module-urllib2) modules to make HTTP requests, but all of the code required to perform a request can be replaced by calling a single method in the `HTTP` (page 69) API. This also has other benefits, like automatic cookie handling and the option to cache the server's response.

While creating the framework APIs, a great deal of care was taken to make plug-in development as easy as possible. If you take the time to familiarize yourself with how the framework operates and the features it provides, you'll become much more productive when writing plug-ins.

2.1.3 Plug-in execution

Another side effect of the plug-in architecture is that plug-ins are incapable of running by themselves. Because they rely heavily on the framework APIs and the runtime environment, plug-ins must be started by the media server.

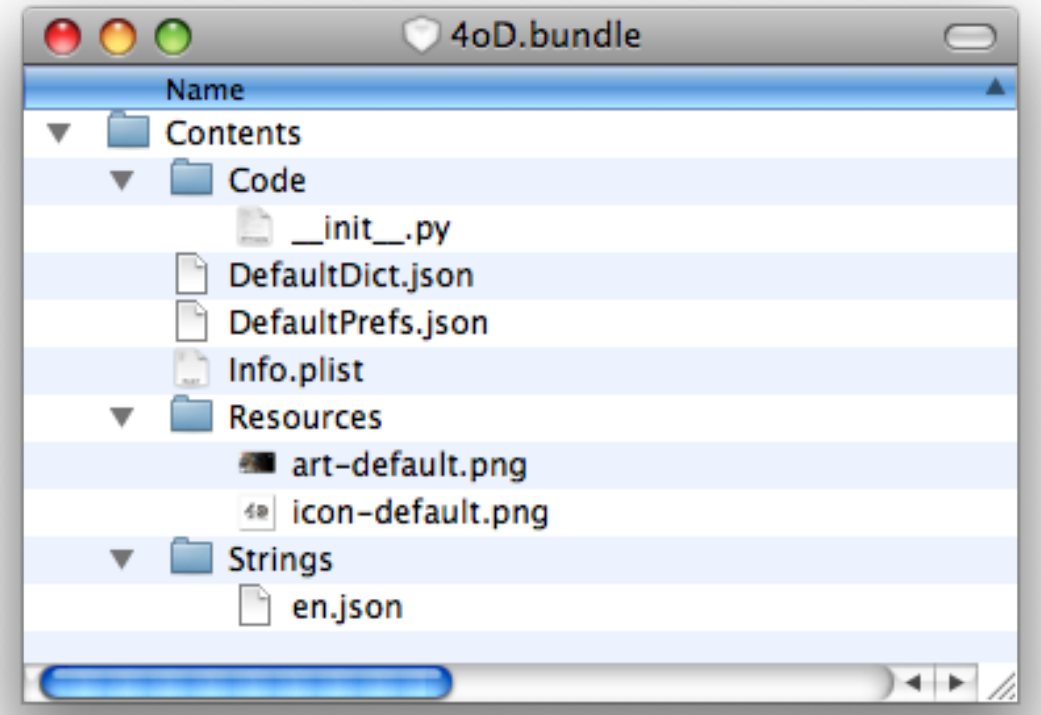
CHAPTER

THREE

BUNDLES

Each plug-in's code and support files are stored in a self-contained bundle. A bundle is a specially organized directory with a *.bundle* extension. Mac OS X treats bundles as if they were files, allowing you to manipulate them as such in Finder. To view the contents of a bundle, right-click on it & select *"Show Package Contents"*. On Windows or Linux, bundles appear as regular folders.

Here's an example of the contents of a bundle:



The bundle directory itself contains only one item: the *Contents* directory. This directory contains all files and other directories belonging to the plug-in. The `Contents` directory should include the following items:

3.1 Configuration files

The framework uses certain configuration files to provide information about the bundle, set up defaults and alter how the runtime environment operates.

3.1.1 The `Info.plist` file

The `Info.plist` file contains information in the Apple property list format about the bundle's contents. This information is used by the framework when loading the plug-in.

Note: This file is required.

Here is an example `Info.plist` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.
<plist version="1.0">
<dict>
     <key>CFBundleIdentifier</key>
     <string>com.plexapp.plugins.amt</string>
     <key>PlexAudioCodec</key>
     <array>
             <string>AAC</string>
     </array>
     <key>PlexClientPlatforms</key>
     <string>MacOSX,iOS,Android,LGTV</string>
     <key>PlexFrameworkVersion</key>
     <string>2</string>
     <key>PlexMediaContainer</key>
     <array>
             <string>M4V</string>
             <string>MOV</string>
     </array>
     <key>PlexVideoCodec</key>
     <array>
             <string>H.264</string>
     </array>
     <key>PlexURLServices</key>
     <dict>
             <key>Apple Movie Trailers</key>
             <dict>
                     <key>URLPattern</key>
                <string>http://.*apple\.com/.*</string>
                <key>Identifier</key>
                <string>com.plexapp.plugins.amt</string>
             </dict>
     </dict>
</dict>
```

```
</plist>
```

The following properties may be defined:

CFBundleIdentifier
: A string property containing a unique identifier for the plug-in. This should be in the reverse DNS format (e.g. `com.mysite.myplugin`).

PlexFrameworkVersion
: A string property specifying the bundle's target framework version. This should be set to `2` to target the current framework version.

PlexPluginClass
: A string plug-in identifying the class of the bundle. This should be set to either `Content` or `Agent`.

PlexClientPlatforms
: A string specifying a comma-separated list of client platforms supported by the bundle's contents. The list should contain constants from `ClientPlatform` (page 88).

PlexURLServices

PlexSearchServices

PlexRelatedContentServices
: These properties describe the services available within the bundle. More information about these keys can be found *here* (page 25).

Note: These properties are optional.

PlexMediaContainer
: An array of strings identifying the containers returned by the plug-in. The array should contain constants from `Container` (page 89).

Note: This property is optional.

PlexAudioCodec
: An array of strings identifying the containers returned by the plug-in. The array should contain constants from `AudioCodec` (page 89).

Note: This property is optional.

PlexVideoCodec
: An array of strings identifying the containers returned by the plug-in. The array should contain constants from `VideoCodec` (page 89).

Note: This property is optional.

3.1.2 The `DefaultPrefs.json` file

`DefaultPrefs.json` is a JSON-formatted file containing a set of preferences used by the plug-in, and the default values to use if the user hasn't specified their own.

More information about the preferences file format can be found *here* (page 85).

3.2 Directories

The remaining files inside the bundle are separated into specific directories, depending on their purpose.

3.2.1 The `Code` directory

The `Code` directory contains all of the plug-in's source code files. Third-party code should not be included here, only files created by the developer.

The `__init__.py` file

The `__init__.py` file is the main source code file of the plug-in. It is responsible for any initialization required, and loading any extra source code files.

See Also:

The `Start()` (page 13) function The predefined framework function which can be used to initialize your plugin.

3.2.2 The `Services` directory

The `Services` directory contains source code files for the plug-in's services.

More information about services can be found *here* (page 25).

3.2.3 The `Resources` directory

The `Resources` directory contains any resource files required by the plug-in, like icons or background images.

Information about loading these resource files in plug-in code is provided in the `Resource` (page 66) API reference.

3.2.4 The `Strings` directory

The `Strings` directory contains JSON-formatted text files for each language the plug-in supports. String files should have a `.json` extension and be named according to the ISO specification for language codes (e.g. `en.json`). The name may also include an optional country code (e.g. `en-us.json`).

More information about string files can be found in the `Locale` (page 79) API reference.

CHAPTER

FOUR

CHANNEL PLUG-INS

Channel plug-ins are the plug-ins users interact with most frequently. They integrate into the Plex client user interface, providing a new branch of media for the user to explore. Developers can easily create plug-ins that provide a hierarchy of media for the user to explore.

4.1 Getting started with channel development

4.1.1 What are channels?

Channels are plug-ins that provide playable content to clients via the media server. They can extend the server's media tree to provide access to almost any type of content available online, and present a rich, browsable hierarchy to the user.

4.1.2 Understanding contexts

Before starting to write code, the developer should understand the concept of **contexts** within the runtime environment. A context is an encapsulation of certain state information. When your plug-in first starts, a single context is created - this is referred to as the **global context**. As a plug-in receives requests, the framework creates a new context for each one - these are referred to as **request contexts**.

Many of the framework's APIs are context aware, meaning that they are able to act on information stored in the current context. What this means for the developer is that they need to do much less work to effectively support multiple parallel requests. The framework will ensure that API calls from plug-in code return values based on the correct context, affecting all manner of things from string localization, to user preferences, to supported object types.

There is only one caveat to this method - global objects cannot be modified from request contexts. Since plug-ins can potentially serve thousands of simultaneous requests, a global object could be modified by one request while it was still in use by another. Rather than use global objects, the developer should attempt to maintain state by passing arguments between functions, or using one of the data storage APIs provided by the framework.

4.1.3 Configuring the Info.plist file

To indicate that the bundle contains a channel plug-in, the *PlexPluginClass* key should be set to `Content` in the `Info.plist` file.

4.1.4 Adding a prefix

Channel plug-ins add themselves to the user interface by first defining a prefix. Once a prefix has been registered, the plug-in effectively 'owns' all requests received by the media server beginning with that prefix.

To register a prefix, the developer must define a function to be the main prefix handler using the `@handler` (page 47) decorator:

```
@handler('/video/example', 'Example')
def Main():
  pass
```

Any incoming requests with the `/video/example` prefix will now be routed to this plug-in, and requests matching the path exactly will be directed to this function, and the channel will appear as a new icon in each supported client's Channels menu.

4.1.5 Creating a menu hierarchy

To create the beginning of a browsable tree of metadata items, the developer needs to create an `ObjectContainer` (page 52), populate it with `objects` and return it to the client. The objects in this container can point to other functions within the plug-in, allowing the developer to define a rich navigation hierarchy.

The simplest and most common use of function callbacks is to connect a `DirectoryObject` (page 62) to a function that will generate another `ObjectContainer` (page 52) (providing a new level of navigation) using the `Callback()` (page 48) function:

```
def Main():
  oc = ObjectContainer(
    objects = [
      DirectoryObject(
        key = Callback(SecondMenu),
        title = "Example Directory"
      )
    ]
  )
  return oc

def SecondMenu():
  oc = ObjectContainer(
    ...
  )
  return oc
```

The `Callback()` (page 48) function can also be used to return image data for thumbnails or background art, or provide data for a media object (usually by redirecting to the real source).

4.1.6 Predefined functions

The framework reserves a few predefined function names. The developer can implement these functions if they wish to. They will be called when specific events occur.

`Start``()`
: This function is called when the plug-in first starts. It can be used to perform extra initialisation tasks such as configuring the environment and setting default attributes.

`ValidatePrefs``()`
: This function is called when the user modifies their preferences. The developer can check the newly provided values to ensure they are correct (e.g. attempting a login to validate a username and password), and optionally return a `MessageContainer` to display any error information to the user.

`SetRating`(*key*, *rating*)
: This function is called when the user sets the rating of a metadata item returned by the plug-in. The *key* argument will be equal to the value of the item's `rating_key` attribute.

4.2 Site Configurations

Note: Playback of Flash or Silverlight content using site configurations is currently only supported on Mac OS X.

4.2.1 Overview

A Site Configuration file allows a Flash or Silverlight player to be played within a web browser.

Pretend the you could fire up a web browser, go to your favorite video site and draw a rectangle around just the video player. The video player would then be zoomed in to fill the screen and all of other web page junk is just thrown away so that you can watch the video without any distractions.

Site Configurations are complements to `WebVideoItem()` in the plugin framework. They act as handlers to the urls that the WebVideoItem() passes out.

When you should NOT create a Site Config

If you have access to the video stream directly, then you should not use a site configuration file.

You will have to do some sleuthing to see if you can get access to the .flv, .mp4 or RTMP streams directly. You can use tools like the Net inspector in Firebug or you can sometimes find the stream URL passed in as a parameter to the player if you view the HTML source.

Some formats will just not work. RTMPE (note the E) or other formats which have DRM will have to be played with the flash/silverlight player and warrant the creation of a Site Configuration file

Where to put them

Site configurations can go one of two places.

1. **(recommended)** Inside of your plugin bundle file `MyPlugin.bundle/Contents/Site Configurations/yourfile.xml`
2. `~/Library/Application Support/Plex Media Server/Site Configurations/yourfile.xml`

4.2.2 Gathering Player Information

When creating site configs, often you will need to have access to information such as:

1. Url's of the webpage / player
2. Height and Width of the video player
3. X/Y coordinates of buttons on the player
4. X/Y coordinates of other elements / colors on the screen

You'll use this information later on to do things like "click on a button" or "find out what color the pixel is here"

This information can be a real pain to get without some tools and techniques to help you out. Here are just a few.

URLs

You will need two urls saved away somewhere for future reference. Later on, you will use these URLs in order to create two regular expressions for the <*site*> (page 100) tag

The first URL you will need is the URL of the page which contains the Flash or Silverlight player. You can simply copy and paste the URL from your browser URL bar.

The other URL you will need is the URL of the Flash or Silverlight player. You can use the "View Source" function of your web browser and then search for the URL that is used in the <embed> tag for the plugin.

Now that you have these URL's save them away in a text file for later use.

Player Size and Coordinates

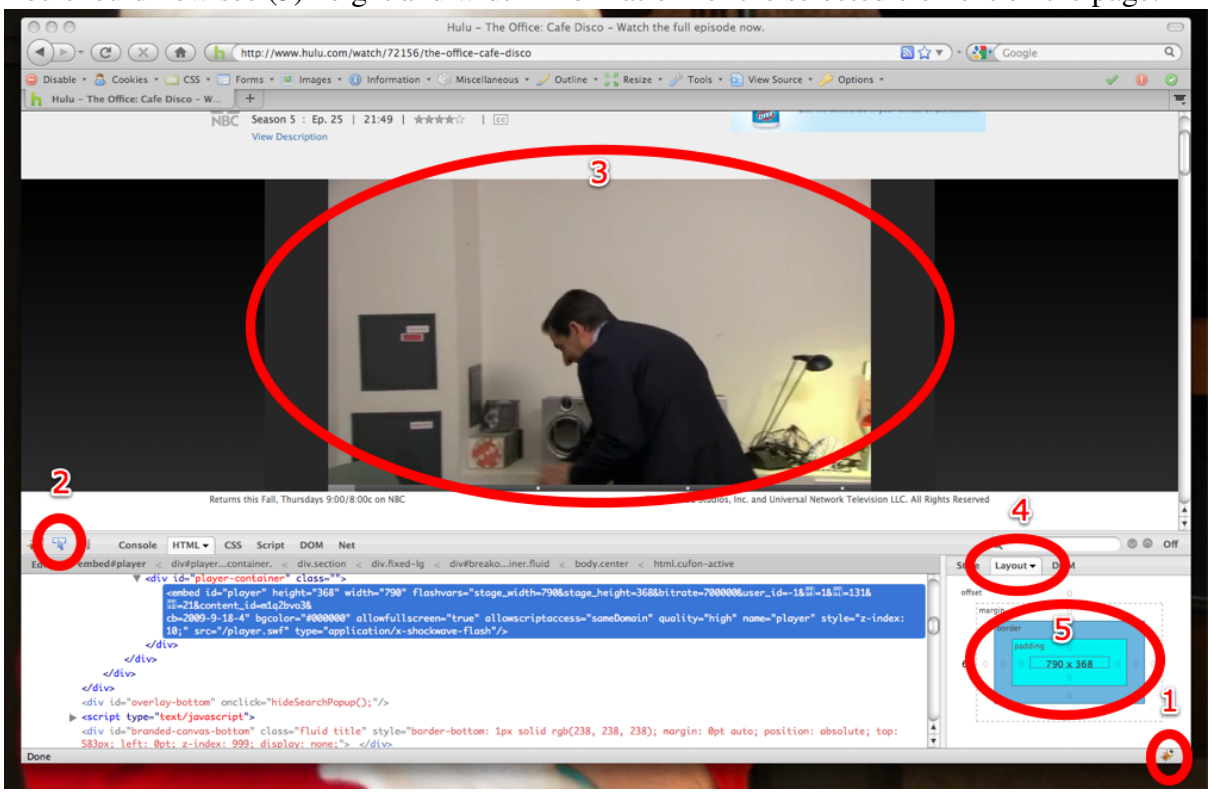
Firebug

Install Firebug (http://getfirebug.com/) for Firefox (http://getfirefox.com) and then simply visit the webpage containing the video player.

From here, (1) open Firebug and (2) click on the "Inspect" icon and then click on the (3) flash player in the page.

Information about this element list CSS style information now shows on the right hand side. Click on the (4) "Layout" tab over on the right.

You should now see (5) height and width information for the selected element on the page.



Free Ruler

Another excellent tool to use (OSX) is called Free Ruler (http://www.pascal.com/software/freeruler/).

Download and install this program and you will be able to overlay the horizontal and vertical rulers over the top/left edge of the flash player. With FreeRuler in the foreground, you'll be able to hit `Command-C` to copy the x/y coordinates of the current mouse position so that you can use this information later on with tags such as *<click>* (page 93) and *<move>* (page 96).

Colors

You can use the *DigitlColor Meter* tool which comes with OSX to find out the hex color value of any pixel on the screen.

4.2.3 The XML File

The <site> Element

Site configuration files all start out with the `<site>` tag along with the proper `<xml? ..` definition. A bare bones example looks like so:

```
<site  site="http://www.southparkstudios.com/episodes"
       plugin="http://media.mtvnservices.com/mgid.*southparkstudios.com"
       initialState="playing"
       version="1.0">

       <!-- ... -->

</site>
```

The `site` and `plugin` attributes are actually regular expressions which match URL's

The `site` regular expression must match the URL for the webpage which contains the flash or silverlight plugin. This regular expression is scanned whenever a `WebVideoItem` url is played.

If another site configuration file has a url which also matches the url, then the regular expression which is the most specific (longer) wins and the site configuration file containing that regular expression is run.

The `plugin` regular expression must match the url for the embedded player. This URL is usually found in the `<embed>` tag of the web page.

The `initialState` attribute tells the player to begin the the *<state>* (page 100) tag with the corresponding `name` attribute. More on *States and Events* (page 18)

Seekbars

A seekbar is a representation of your progress through a video. It shows you information such as:

1. How far through the video you are
2. Control for skip forward / skip backward

Some seekbars are graphical in nature only. For these, use the 'simple' or 'thumb' value for the `type` attribute on the *<seekbar>* (page 99) tag.

Some Flash/Silverlight players expose hooks in their players which allow javascript to actually control and retrieve information from the player. This is an advanced topic and you would use 'javascript' for the `type` attribute on the *<seekbar>* (page 99) tag.

Simple

This is a seekbar which shows progrss as a bar going from left to right across the player. It does NOT have a "thumb" for you to grab/drag in order to skip to a different place in the video.

Here is an example:

```
<seekbar type="simple">
  <start x="63" y="290" />
  <end x="388" y="290" />
  <played>
    <color rgb="4f4f4f" />
    <color rgb="616161" />
    </played>
</seekbar>
```

This example says the following in plain english.

The seekbar on this player starts at (63,290) and ends at (388,290). These coordinates are not relative to the actual web page, but of the actual flash/silverlight player. We create a line created by "drawing" from <start> to <end>. If, under this line, we see pixels colored #4f4f4f or #616161, then we'll consider that portion played. Once we find a color that does NOT match, we'll stop looking.

The site config then does the math for the portion of played / unplayed and now knows how to do a few things:

1. where to click the mouse when you skip ahead or skip back
2. how to update the progress bar in Plex

Thumb

This is a seekbar which has a "thumb" for you to grab and drag around to set the position of the video. It is very similar to a 'simple' seekbar but the way the colors are interpreted is a litte different.

Here is an example:

```
<seekbar type="thumb">
 <start x="63" y="290" />
 <end x="388" y="290" />
 <played>
    <color rgb="d3d3d3" />
 </played>
</seekbar>
```

This example says the following in plain english.

The seekbar on this player starts at (63,290) and ends at (388,290). These coordinates are not relative to the actual web page, but of the actual flash/silverlight player. We create a line created by "drawing" from <start> to <end>. If, under this line we see the color #d3d3d3, then we immediately stop and record that position as the current playing position.

Javascript

As mentioned above, some video players have exposed javascript API's for their video players. For those sites that have done this, you can access the values from those functions and pass them back over to Plex.

```
<seekbar type="javascript">
  <percentComplete equals="$.currentTime()/$.duration() * 100.0" />
  <bigStep minus="$.seek($.currentTime() - 30)" plus="$.seek($.currentTime() + 3
  <smallStep minus="$.seek($.currentTime() - 10)" plus="$.seek($.currentTime() +
  <end condition="$.currentTime() > 0 &amp;amp;&amp;amp; $.currentTime() > $.duration()"
</seekbar>
```

There are a few things that should be mentioned about the example above.

First, you'll notice the `$` variable. This is a special variable that Plex creates when the page is loaded. It is equal to the DOM node of the flash player which was identified when the site configuration file was loaded.

It is somewhat equivalant to Plex having done this for you where the element in question is the actual embeded SWF file:

```
$ = document.getElementById(...)
```

States and Events

The web video player allows you to define states of being as well as events and actions that are valid during those states.

You will be using events to detect a change and you'll then take "action". Taking action typically involves also moving into another state, which is aware of other events and actions.

We'll go into detail about the <*action*> (page 91) and <*event*> (page 95) tags later on but here are a few examples of the terms 'state', 'event' and 'action'

Examples of *states*:

1. Playing the video
2. Video is paused
3. An Ad is playing
4. Video is buffereing
5. Video is done playing

Examples of *events*:

1. User pressed the button named pause
2. Noticed a certain color or colors somewhere on the screen

Examples of *actions*:

1. Click a button

2. Move to another state

Lets take a look at a full example:

```
<site  site="http://www.southparkstudios.com/episodes"
       plugin="http://media.mtvnservices.com/mgid.*southparkstudios.com"
       initialState="playing"
       version="1.0">
 <!-- PLAYING -->
 <state name="playing">
  <event>
   <condition>
    <command name="pause" />
   </condition>
   <action>
    <click x="15" y="304" />
    <goto state="paused" />
   </action>
  </event>
 </state>
 <!-- PAUSED -->
 <state name="paused">
  <event>
   <condition>
    <command name="play" />
   </condition>
   <action>
    <click x="15" y="304" />
    <goto state="playing" />
   </action>
  </event>
 </state>
</site>
```

Here is what this file is saying.

We've found a SWF file and we are going to enter the initial state with the name of "playing". If the user causes the command named "pause" to run (could be keyboard, mouse, remote control etc), then we're going to click on the screen at (15,304) and then go to the "paused" state. Then the user runs the command named "play" and we'll click on (15,304) again and then enter the "playing" state.

Note that the `name` attribute of the *<state>* (page 100) tag doesn't inherently mean anything. It is simply a label so that we can *<goto>* (page 96) it and reference it with `initialState`. We could have called the "playing" state "foo" and the . Plex Media Server does not care.

The "play" in *<command>* (page 94) tag DOES matter but we will dive into that later.

Each state may have 0 or more *<event>* (page 95) tags.

Event Conditions

Each <event> (page 95) tag has one <condition> (page 93) tag and one <action> (page 91) tag inside.

Conditions are then built up through a combination of logic tags and conditionals. Specifics on the usage of each tag is detailed in the documentation for each tag individually.

Logic Tags:

- <and> (page 92)
- <or> (page 97)
- <not> (page 97)

Conditional Tags:

- <color> (page 93)
- <command> (page 94)
- <condition> (page 93) (special case. see: .. *Named Condition Definitions* (page 21))
- <frameLoaded> (page 95)
- <javascript> (page 96)
- <pref> (page 99)
- <title> (page 101)
- <url> (page 102)

Example:

```
<state name="playing">
 <event>
  <condition>
   <and> <!-- logic tag -->
     <command name="pause" /> <!-- conditional -->
     <or> <!-- logic tag -->
       <color x="45" y="45" rgb="ffffff"/> <!-- conditional -->
       <color x="46" y="45" rgb="ffffff"/> <!-- conditional -->
     </or>
   </and>
  </condition>
  <action>
   <click x="15" y="304" />
   <goto state="paused" />
  </action>
 </event>
</state>
```

Event Actions

As mentioned previously, each <*event*> (page 95) tag has one <*condition*> (page 93) tag and one <*action*> (page 91) tag inside.

These <*action*> (page 91) tags can then have one or more actions as children. They are executed in the order that they are defined from top to bottom.

Specifics on the usage of each action is detailed in the documentation for each tag individually.

Action Tags:

- <*click*> (page 93)
- <*goto*> (page 96)
- <*lockPlugin*> (page 96)
- <*move*> (page 96)
- <*pause*> (page 98)
- <*run*> (page 99)
- <*type*> (page 101)
- <*visit*> (page 102)

Named Condition Definitions

The <*condition*> (page 93) tag is used in one of two ways. As an event condition (see: *Event Conditions* (page 20)) or in this case, as a definition of a named condition which can be called out and used in event conditions.

Think of these as condition "functions" that can be re-used in more than one <*state*> (page 100).

The following example defined a named conditional, 'buffering_colors_showing' and then calls out to that condition later on in the "playing" state:

```
<site  site="http://www.southparkstudios.com/episodes"
       plugin="http://media.mtvnservices.com/mgid.*southparkstudios.com"
       initialState="playing"
       version="1.0">

 <!-- NAMED CONDITIONAL DEFINITION -->
 <condition name="buffering_colors_showing">
   <and>
     <color x="1" y="1" rgb="ffffff"/>
     <color x="2" y="2" rgb="ffffff"/>
   </and>
 </condition>

 <state name="playing">
  <event>
```

```
   <condition>
    <and>
       <command name="pause" />
       <not>
         <condition name="buffering_colors_showing"/>
       </not>
    </and>
   </condition>
   <action>
    <click x="15" y="304" />
    <goto state="paused" />
   </action>
  </event>
 </state>

 <!-- ... -->

</site>
```

As you can see, this can make for powerful condition reuse especially when you combine named conditions with logic tags such as <*or*> (page 97).

Advanced

User Agents

Some sites have strange user agent requirements or block requests based on the user agent.

You can specify a user agent using the `agent` attribute of the <*site*> (page 100) tag.

You can randomize the user agent by setting `randomAgent` to `true` in the <*site*> (page 100) tag.

TODO: what is the default user agent?

Access to Preferences

There are several ways you can use preferences.

In conditionals, you can test to see if a preference key exists by using the <*pref*> (page 99) conditional tag like so:

```
<condition>
  <pref name="username" exists="true"/>
</condition>
```

The value of preference variables are also available for use in other tags as well. For instance, when filling out a form in the site config, you can type the value of a pref using the <*type*> (page 101) tag like so:

```
<type text="${username}"/>
```

You can also use preference variable replacements in javascript expressions and several other places like so:

```
<run script="doLogin('${username}','${password}');"/>
```

Preference variable replacement works in the following elements/attributes:

1. <*setCookie*> (page 99) `value` attribute
2. <*run*> (page 99) `script` attribute
3. <*type*> (page 101) `text` attribute
4. <*smallStep*> (page 101) `plus` and `minus` attributes
5. <*bigStep*> (page 92) `plus` and `minus` attributes
6. <*end*> (page 95) `condition` attribute
7. <*javascript*> (page 96) `script` attribute

Changed in version Plex: 0.8.3 <*setCookie*> (page 99) `value` attribute can accept javascriptChanged in version Plex: 0.8.3 <*javascript*> (page 96) `script` attribute can accept javascript

---

Note: When referencing preference variables using javascript, the preference value is returned as a string. This included 'bool' preferences so for instance expect the string 'true' and not the javascript value of *true* when referencing 'bool' preferences.

Setting Cookies

You can set cookies for plugins too! Just take a look at the <*setCookie*> (page 99) tag documentation for more details.

Filling Out Forms

This can be tricky and you'll need to use a combination of techniques to do this including detecting if a form is present, clicking on form elements, filling them in etc.

here is an example from the mlb.xml site configuration file:

```
<!-- login prompt -->
<state name="login">
  <event>
    <condition>
      <and>
        <condition name="prompting-for-login"/>
        <pref name="login" exists="true"/>
        <pref name="password" exists="true"/>
```

```
      </and>
    </condition>
    <action>
      <!-- set focus -->
      <click x="116" y="409"/>
      <pause time="10"/>

      <!-- "double click" the email input to select all -->
      <click x="116" y="409"/>
      <click x="116" y="409"/>
      <pause time="100"/>
      <type text="${login}"/>

      <!-- click the password input -->
      <pause time="10"/>
      <click x="116" y="439"/>
      <pause time="100"/>
      <type text="${password}"/>

      <!-- submit the form -->
      <pause time="100"/>
      <type key="13"/>

      <!-- wait for the form to animate closed -->
      <pause time="2000"/>

      <goto state="playing"/>
    </action>
  </event>
</state>
```

Javascript Execution

Javascript can be run in two places.

Actions

If you would like to run javascript inside of an action, you'll need to use the <*run*> (page 99) tag.

ex.

```
<run script="myVariable=10;"/>
```

See the syntax for the <*run*> (page 99) tag for more details.

Conditions

If you would like to test a javascript expression for truth inside of a <*condition*> (page 93), then you'll need to use the <*javascript*> (page 96) tag.

example assuming `myVariable` is 10 :

```
<javascript script="myVariable" matches="10"/> <!-- true condition -->
<javascript script="myVariable == 10 ? 1 : 0" matches="1"/> <!-- true condition
<javascript script="myVariable == 99 ? 1 : 0" matches="1"/> <!-- false condition
```

4.3 Services

The Plex development platform provides a great deal of power and flexibility for developing self-contained channel plug-ins. However, these plug-ins can (and often do) present very different media hierarchies and handle media from a wide range of sources. Because of this inconsistency, it is difficult to treat the content as anything other than a navigable tree of media - there is no standard way to obtain meaningful information about a specific piece of media, find a piece of media in a predictable manner, or establish relationships between multiple pieces of media, restricting what channel plug-ins are capable of.

In order to solve this limitation, the plug-in framework includes the concept of "services". A service consists of a set of functions that work together to perform a specific task in a well-defined manner. Because the behaviour of these services is always consistent between plug-ins, a lot of new and interesting functionality can be leveraged, allowing much deeper integration into the Plex ecosystem - media easily be searched for or linked with related content, and additional features will be enabled in future. The code is often simpler and easier to reuse too!

Three types of service are currently available:

4.3.1 URL Services

URL services are responsible for taking a website's URL and returning standardized Plex metadata objects. For example, given the URL "http://youtube.com/watch?v=vzKbhxY1eQU", the URL service would return a video clip object with appropriate title, summary, and other relevant details.

The magic of a URL service is its ability to convert arbitrarily formatted web content into a normalized schema for a piece of media, with all the richness of the Plex library metadata (e.g. tags, cast, media format/codec information, etc.).

Defining a URL service

To add a URL service to a plug-in, the developer just to make a small addition to the plug-in's Info.plist file. The following example illustrates a definition for a URL service handling content from YouTube:

```
<key>PlexURLServices</key>
<dict>
   <key>YouTube</key>
   <dict>
      <key>Identifier</key>
      <string>com.plexapp.plugins.youtube</string>
```

```
      <key>URLPattern</key>
      <string>http://.*youtube\.com/(watch)?\?v=</string>
      <key>TestURLs</key>
      <array>
         <string>http://youtube.com/watch?v=vzKbhxY1eQU</string>
      </array>
   </dict>
</dict>
```

`PlexURLServices` is a dictionary that can define multiple URL services. Services must be uniquely named.

Each service must have a `URLPattern` defined. The value should be a regular expression that matches URLs that can be handled by the service. Plex uses this expression to determine which service should be used to handle a given URL.

The `Identifier` of the service must be a globally unique string, and cannot be shared with any other URL service, even ones in different bundles. While optional, this string must be provided in order to link a URL service with a related content service.

Additionally, each service can optionally define an array of `TestURLs`. This array should contain a list of URLs that the service is capable of handling. Plex's automated testing system will use these URLs to verify that the service is functioning correctly and generate an alert if any problems occur. Developers are strongly encouraged to add a list of testable URLs to allow errors to be caught quickly.

Creating a source file

Once a service has been defined in the Info.plist file, the developer needs to add a file containing the service's source code. The file should be named `ServiceCode.pys` (the .pys extension indicates that the file contains service code) and added to a subdirectory with the same name as the key used in the service definition. Continuing the YouTube example above, the path of the file would be as follows:

```
Contents/URL Services/YouTube/ServiceCode.pys
```

Writing the code

URL services should define the following functions:

`MetadataObjectForURL`(*url*)

This function should create and return a metadata object (for example, a VideoClipObject) and populate it with metadata from the given URL. Only the metadata should be added here - the object's `key` and `rating_key` properties will be synthesised based on the URL.

Note: Although the object's media items can be created here, this isn't necessary. If no items are provided, they will be populated using the function defined below.

Parameters **url** (*str* (http://docs.python.org/library/functions.html#str)) – The URL of the web page to use as a source for generating the metadata object.

Returns A metadata object representing the media available at the given URL.

MediaObjectsForURL(*url*)

This function should create and return a list of media objects and part objects representing the media available at the given URL. Callbacks may be used if obtaining the final media location requires additional computation.

Note: This function is expected to execute and return very quickly, as it could be called several times for different URLs when building a container. As such, developers should avoid making any API calls that could delay execution (e.g. HTTP requests).

Parameters **url** (*str* (http://docs.python.org/library/functions.html#str)) – The URL of the web page to use as a source for generating the metadata object.

Returns Media objects representing the media available at the given URL.

Return type list

NormalizeURL(*url*)

This function should return a "normalised" version of the given URL. Plex uses the URL as a unique identifier for a given piece of content, but the same media will often appear at different URLs, even simply due to additional query string arguments. For example, the following URLs all point to the same video:

- http://www.youtube.com/watch?v=dQw4w9WgXcQ
- http://www.youtube.com/?v=dQw4w9WgXcQ
- http://youtube.com/watch?v=dQw4w9WgXcQ
- http://www.youtube.com/watch?v=dQw4w9WgXcQ&feature=list_related&playnext=1&list=ML

In this case, the normalised version of this URL would be:

- http://youtube.com/?v=dQw4w9WgXcQ

This function is called automatically before passing the URL to the above two functions. They will always receive normalised URLs.

Note: This function is expected to execute and return very quickly, as it could be called several times for different URLs when building a container. As such, developers should avoid making any API calls that could delay execution (e.g. HTTP requests).

Note: This function is optional. If each piece of media is only accessible via a single URL, the developer may omit this function.

Parameters **url** (*str* (http://docs.python.org/library/functions.html#str)) – The URL to be normalised.

Returns A normalised version of the URL.

Return type str

Testing the service

The simplest way of testing a URL service is to use the built-in lookup function. Using the example URL given above, the service could be tested by making the following HTTP request:

```
$ curl "http://localhost:32400/system/services/url/lookup?url=http%3A%2F%2Fwww.y

<?xml version='1.0' encoding='utf-8'?>
<MediaContainer size="1" identifier="com.plexapp.system" mediaTagPrefix="/system
  <Video url="http://youtube.com/?v=dQw4w9WgXcQ" key="/system/services/url/looku
    <Media indirect="1">
      <Part file="" key="/:/plugins/com.plexapp.system/urlservice_function/Y2VyZ
    </Media>
    <Media indirect="1">
      <Part file="" key="/:/plugins/com.plexapp.system/urlservice_function/Y2VyZ
    </Media>
    <Media indirect="1">
      <Part file="" key="/:/plugins/com.plexapp.system/urlservice_function/Y2VyZ
    </Media>
    <Genre tag="Music"/>
    <Tag tag="Rick"/>
    <Tag tag="Astley"/>
    <Tag tag="Sony"/>
    <Tag tag="BMG"/>
    <Tag tag="Music"/>
    <Tag tag="UK"/>
    <Tag tag="Pop"/>
  </Video>
</MediaContainer>
```

Calling the service from plug-in code

The URL service can be invoked using the Service API. However, this is rarely the best course of action. Using the URL service to generate a full metadata object is often slow, and since plug-ins are usually providing a large container of items to be returned to a client, calling the service for each item in the container would cause an unacceptable delay for the user.

The most beneficial use of URL services within plug-ins is to use them to generate the media objects, but not the metadata. Since creating the media objects should always be fast, and

the plug-in will usually have access to the metadata from the web page used to generate the container, this hybrid approach offers the best of both worlds - fast execution and use of the extra features enabled by using the service.

In order to simplify implementation of this common case, the framework will automatically call the URL service to populate the list of media items and set the metadata object's `key` and `rating_key` properties if the developer sets the metadata object's `url` property:

```
video = VideoClipObject(
  title = video_title,
  summary = video_summary,
  originally_available_at = video_date,
  rating = video_rating,
  url = 'http://www.youtube.com/watch?v=dQw4w9WgXcQ'
)
```

4.3.2 Search Services

Search services are responsible for accepting a search query and returning a container of metadata objects that match the query. They are far simpler than URL services, and usually benefit from being able to reuse some of the service code. All installed search services are able to contribute to results from Plex's universal search feature.

Defining a search service

Like URL services, search services are defined in the plug-in's Info.plist file. The following example illustrates the definition of the VideoSurf search service:

```
<key>PlexSearchServices</key>
<dict>
   <key>VideoSurf</key>
   <dict>
      <key>Identifier</key>
      <string>com.plexapp.search.videosurf</value>
   </dict>
</dict>
```

As with URL services, `PlexSearchServices` is a dictionary that can contain multiple search services. The key of each item should be a name for the service unique to the bundle. The value of each item should be a dictionary containing information about the service.

Currently the only item stored in this dictionary is `Identifier`. This should be a globally unique identifier for the search service - no two search services (even in separate bundles) may share an identifier. Search services may share an identifier with an URL service or related content service.

Creating a source file

Similarly, the developer needs to add a file containing the service's source code after defining the service in the Info.plist file. The file should be named `ServiceCode.pys` (the .pys extension indicates that the file contains service code) and added to a subdirectory with the same name as the key used in the service definition. Continuing the VideoSurf example above, the path of the file would be as follows:

```
Contents/Search Services/VideoSurf/ServiceCode.pys
```

Writing the code

Search services should define the following function:

Search(*query*)

This function should accept a user-entered query and return a container of relevant results. Mixed object types can be added to the container if necessary - the client is responsible for separating them into relevant groups.

In addition to the standard metadata properties, the `source_title` property can be set on the returned objects if required. This will be displayed by the client to indicate the result's source, and is useful for services that return results from multiple sources (like VideoSurf). If no source title is given, the service's name is used.

If the items to be returned can be handled by a URL service, it can be invoked by setting the `url` property of the objects.

Parameters **query** (*str* (http://docs.python.org/library/functions.html#str)) – The search query entered by the user.

Returns A container of items matching the given query.

Return type `ObjectContainer` (page 52)

Testing the service

The simplest way of testing a search service is to use the built-in search function. Using the example service above and the query *"dog"*, the service could be tested by making the following HTTP request:

```
$ curl "http://localhost:32400/system/services/search?identifier=com.plexapp.sea

<?xml version='1.0' encoding='utf-8'?>
<MediaContainer size="20" identifier="com.plexapp.system" mediaTagPrefix="/syste
  <Video url="http://www.youtube.com/watch?v=BqeJlOWdyLs" key="/system/services/
    <Media indirect="1">
      <Part file="" key="/:/plugins/com.plexapp.system/urlservice_function/Y2VyZ
    </Media>
    <Media indirect="1">
      <Part file="" key="/:/plugins/com.plexapp.system/urlservice_function/Y2VyZ
    </Media>
```

```
    <Media indirect="1">
      <Part file="" key="/:/plugins/com.plexapp.system/urlservice_function/Y2VyZ
    </Media>
  </Video>

  ...

</MediaContainer>
```

4.3.3 Related Content Services

Related content services are similar to search services - they are responsible for accepting a metadata object and returning a new container of metadata objects that are related to the one provided. Like search services, they too can usually benefit from being able to reuse some of the URL service code.

Defining a related content service

Like other services, related content services are defined in the plug-in's Info.plist file. The following example illustrates the definition of a related content service for YouTube:

```
<key>PlexRelatedContentServices</key>
<dict>
   <key>YouTube</key>
   <dict>
      <key>Identifier</key>
      <string>com.plexapp.search.videosurf</value>
   </dict>
</dict>
```

As with other services, `PlexRelatedContentServices` is a dictionary that can contain multiple related content services. The key of each item should be a name for the service unique to the bundle. The value of each item should be a dictionary containing information about the service.

Currently the only item stored in this dictionary is `Identifier`. This should be a globally unique identifier for the related content service - no two related content services (even in separate bundles) may share an identifier. Each related content services must be associated with a URL service by assigning a common identifier to both. This allows Plex to select related content for an item based on its URL.

Creating a source file

The developer needs to add a file containing the service's source code after defining the service in the Info.plist file. The file should be named `ServiceCode.pys` (the .pys extension indicates that the file contains service code) and added to a subdirectory with the same name as the

key used in the service definition. Continuing the YouTube example above, the path of the file would be as follows:

```
Contents/Related Content Services/YouTube/ServiceCode.pys
```

Writing the code

Search services should define the following function:

RelatedContentForMetadata(*metadata*)

This function should accept a metadata object and return a container of related items. Mixed object types can be added to the container if necessary - the client is responsible for separating them into relevant groups.

In addition to the standard metadata properties, the `source_title` property can be set on the returned objects if required. This will be displayed by the client to indicate the result's source, and is useful for services that return results from multiple sources (like VideoSurf). If no source title is given, the service's name is used.

If the items to be returned can be handled by a URL service, it can be invoked by setting the `url` property of the objects.

Parameters **metadata** – The metadata object to present related content for.

Returns A container of items related to the given metadata object.

Return type `ObjectContainer` (page 52)

Testing the service

The simplest way of testing a related content service is to use the built-in lookup function. Using the example service above and the sample URL `http://www.youtube.com/watch?v=dQw4w9WgXcQ`, the service could be tested by making the following HTTP request:

```
$ curl "http://localhost:32400/system/services/relatedcontent/lookup?url=http%3A
```

CHAPTER

FIVE

AGENT PLUG-INS

Agent plug-ins mostly operate behind the scenes, hardly ever requiring any user interaction. However, they are just as important as channel plug-ins. Agent plug-ins provide search results and metadata for Plex's media library system. As with channel plug-ins, they can be easily created and extended by developers, allowing metadata from almost any source to be added to media in the library.

5.1 Getting started with agent development

5.1.1 What are agents?

Agents are plug-ins that provide search results used for matching media, and metadata for the media once it's been matched. While they use the same framework as channel plug-ins, and as such have access to the same APIs, their runtime operation is slightly different.

Agent plug-ins do not need to register prefixes or define a navigation hierarchy. Instead they must define a class capable of performing the functions required to integrate with Plex's metadata retrieval process. Once the concepts behind the operation of agent plug-ins are understood, agent development is very simple.

5.1.2 Understanding the metadata retrieval process

Rather than communicating with agent plug-ins directly, the media server issues agent requests via a service provided by the framework. This allows many requests to be queued and processed asynchronously.

When a new piece of media is found, the media server will issue a search to attempt to match the media with a known piece of metadata online. Agents will be called depending on the configuration of the section containing the media. The primary agent for the section will be called and provided with a set of hints that the media server was able to extract from the file's name and embedded metadata. It is the responsibility of the agent to return a set of search results that are potential matches for the media. The best match will be automatically selected by the media server, but the list may also be presented to the user if they need to manually override the automatic match. This method is called synchronously - the media server will wait

for it to complete so results can be selected or displayed, and as such it should return as quickly as possible.

Once a piece of media has been matched successfully, the media server issues an update request to the agents. The update methods of the agents associated with the library section are called, and passed a metadata object. It is then the agent's responsibility to fill in any available metadata that hasn't already been populated. The update method may be used to update an existing metadata object - it is the developer's responsibility to check the object's contents and avoid downloading metadata that's already there (especially images), as this can place undue burden on web servers and degrade the media server's performance. This method is called asynchronously. There is no limit on the amount of time an agent can take to download metadata, but the developer should attempt to make their method complete as quickly as possible in order to improve the user experience.

After the update methods in all required agents have completed, the framework's agent service creates a composite version of the metadata object including as much data as possible from the user's selected sources, using the priority order they specified. The combined metadata object is then loaded by the media server and used to populate the library database.

5.1.3 Configuring the Info.plist file

To indicate that the bundle contains an agent plug-in, the *PlexPluginClass* key should be set to `Agent` in the `Info.plist` file.

5.1.4 Defining an agent class

This is an example of an agent class definition:

```
class MyAgent(Agent.Movies):

  name = 'My Agent'
  languages = [
    Locale.Language.English,
  ]
  primary_provider = True
  fallback_agent = False
  accepts_from = None
  contributes_to = None


  def search(self, results, media, lang, manual):
    ...

  def update(self, metadata, media, lang, force)
    ...
```

Agents should inherit from either `Agent.Movies`, `Agent.TV_Shows`, `Agent.Artist` or `Agent.Album`.

The following class attributes may be defined:

name
A string defining the name of the agent for display in the GUI. *This attribute is required.*

languages
A list of strings defining the languages supported by the agent. These values should be taken from the constants defined in the `Locale` (page 79) API. *This attribute is required.*

primary_provider
A boolean value defining whether the agent is a primary metadata provider or not. Primary providers can be selected as the main source of metadata for a particular media type. If an agent is secondary (`primary_provider` is set to False) it will only be able to contribute to data provided by another primary agent. *This attribute is required*

fallback_agent
A string containing the identifier of another agent to use as a fallback. If none of the matches returned by an agent are a close enough match to the given set of hints, this fallback agent will be called to attempt to find a better match. *This attribute is optional.*

accepts_from
A list of strings containing the identifiers of agents that can contribute secondary data to primary data provided by this agent. *This attribute is optional.*

contributes_to
A list of strings containing the identifiers of primary agents that the agent can contribute secondary data to. *This attribute is optional.*

Note: The list of available contributors for a particular agent is a composite of the `accepts_from` and `contributes_to` attributes of the various agents involved, and is computed at runtime.

The `search` and `update` methods are detailed in the following sections.

5.2 Searching for results to provide matches for media

When the media server needs an agent to perform a search, it calls the agent's `search` method:

search(*self*, *results*, *media*, *lang*, *manual*)

Parameters

- **self** – A reference to the instance of the agent class.
- **results** (`ObjectContainer` (page 52)) – An empty container that the developer should populate with potential matches.
- **media** (`Media` (page 36)) – An object containing hints to be used when performing the search.
- **lang** (*str* (http://docs.python.org/library/functions.html#str)) – A string identifying the user's currently selected language. This will be one of the constants added to the agent's `languages` attribute.

- **manual** (*bool* (http://docs.python.org/library/functions.html#bool)) – A boolean value identifying whether the search was issued automatically during scanning, or manually by the user (in order to fix an incorrect match)

class Media

The media object provided to the `search` method provides the developer with all the hints found by the media server while scanning for media. For media that contains several individual playable items (e.g. albums or TV shows), the hints for the most recent item are provided.

primary_metadata

If the search is being called in a secondary agent, the metadata object from the primary agent will be provided here. If the search is being called in a primary agent, this attribute will be `None`.

primary_agent

If the search is being called in a secondary agent, the identifier of the primary agent will be provided here. If the search is being called in a primary agent, this attribute will be `None`.

filename

The name of the media file on disk.

name

A string identifying the name of the item, extracted from its' filename or embedded metadata.

Note: This attribute is only available for Movie and TV Show metadata searches.

---

openSubtitlesHash

A string identifying the hash of the file, as used by OpenSubtitles.org.

Note: This attribute is only available for Movie and TV Show metadata searches.

---

year

The year associated with the item, extracted from its' filename or embedded metadata.

Note: This attribute is only available for Movie and TV Show metadata searches.

---

duration

The duration of the media, extracted from the video file.

Note: This attribute is only available for Movie and TV Show metadata searches.

---

show

The name of the show the episode belongs to.

Note: This attribute is only available for TV Show metadata searches.

season
The season of the show the episode belongs to.

Note: This attribute is only available for TV Show metadata searches.

episode
The episode number.

Note: This attribute is only available for TV Show metadata searches.

artist
The name of the artist.

Note: This attribute is only available for Artist and Album metadata searches.

album
The name of the album.

Note: This attribute is only available for Artist and Album metadata searches.

track
The name of the track.

Note: This attribute is only available for Artist and Album metadata searches.

index
The position of the track in the album.

Note: This attribute is only available for Album metadata searches.

The media object also provides access to the media hierarchy identified by the scanner and stored in the database. This tree contains more comprehensive information about all items belonging to this piece of media and their locations on disk.

The tree contains the media item & media part structure used by Plex's library system and the `Object` classes. These items and parts can be accessed using Python's standard item getting syntax. For example, to access the first part of the first item in the tree:

```
media.items[0].parts[0]
```

The tree can also be used to access the sub-items of the media (e.g. episodes and albums). The syntax for accessing these items is relatively straightforward and intuitive.

For example, to access the first part of the first item of episode 1 of season 1:

```
media.seasons[1].episodes[1].items[0].parts[0]
```

Note: As shown above, the media items and media parts always start with an index of `0`. Seasons and episodes will always use the canonical number as the key.

---

class MediaPart

file
A string identifying path to the media file on disk.

openSubtitlesHash
A string identifying the hash of the file, as used by OpenSubtitles.org.

Note: This attribute is only available for Movie and TV Show metadata searches.

The developer should use the hints provided by the `media` object to add a series of search result objects to the provided `results` object. Results should be instances of the `MetadataSearchResult` (page 38) class.

class MetadataSearchResult
The `MetadataSearchResult` class includes the following attributes. Each attribute can be set by passing a keyword argument to the constructor, or setting the attribute directly on the object after it has been created.

id
A string that uniquely identifies the metadata. This can be in any format. The provided value will be passed to the `update` method if the metadata needs to be downloaded, so the developer should ensure that the value can be used later to access the metadata without the provided hints.

name
A string defining the name of the matched metadata item. This will be displayed to the user if they choose to manually match a piece of media.

year
An integer defining the year associated with the matched metadata item. This will be displayed to the user if they choose to manually match a piece of media, and can be helpful for identifying the correct item when two similarly or identically named results are returned.

score
An integer defining how close the result matches the provided hints. This should be a value between `0` and `100`, with `100` being considered an exact match. Results with a score of `85` or greater are considered "good enough" for automatic matching, with the highest-scoring result being selected by default.

lang
A string defining the language of the metadata that would be returned by the given

result. This should be equal to one of the constants defined in the `Locale` (page 79) API.

5.3 Adding metadata to media

Once an item has been successfully matched, it is added to the update queue. As the framework processes queued items, it calls the `update` method of the relevant agents.

`update(self, metadata, media, lang)::`

Parameters

- **self** – A reference to the instance of the agent class.
- **metadata** – A pre-initialized metadata object if this is the first time the item is being updated, or the existing metadata object if the item is being refreshed.
- **media** (`Media` (page 36)) – An object containing information about the media hierarchy in the database.
- **lang** (*str* (http://docs.python.org/library/functions.html#str)) – A string identifying which language should be used for the metadata. This will be one of the constants defined in the agent's `languages` attribute.
- **force** (*bool* (http://docs.python.org/library/functions.html#bool)) – A boolean value identifying whether the user forced a full refresh of the metadata. If this argument is `True`, all metadata should be refreshed, regardless of whether it has been populated previously.

Note: The `Media` (page 36) object provided to the update function exposes the media tree in the database, but does not provide any of the extra hints used in the search function.

When called, the function will be passed an instance of a metadata model class (detailed in the next topic). If the item has been updated previously, this object will contain all the data previously added by the agent. If this is the first time the update function has been called for this metadata item, a new, empty object will be passed.

It is the developer's responsibility to check for existing content stored within the object, and avoid populating data that is already present. This is especially important when storing image or audio data, as downloading this unnecessarily can place undue burden on the servers used by the agent and slow down the metadata updating process.

5.4 Metadata Model Classes

The model classes used to provide metadata objects to the `update` method share many common attributes with the framework's `Object` API. These classes are never instantiated directly

by the developer; they are generated automatically and passed to the `update` method as the `metadata` argument.

The attributes supported by each class are listed below.

class Movie

Represents a movie (e.g. a theatrical release, independent film, home movie, etc.)

genres
A set of strings specifying the movie's genre.

tags
A set of strings specifying the movie's tags.

collections
A set of strings specifying the movie's genre.

duration
An integer specifying the duration of the movie, in milliseconds.

rating
A float between 0 and 10 specifying the movie's rating.

original_title
A string specifying the movie's original title.

title
A string specifying the movie's title.

year
An integer specifying the movie's release year.

originally_available_at
A `date` object specifying the movie's original release date.

studio
A string specifying the movie's studio.

tagline
A string specifying the movie's tagline.

summary
A string specifying the movie's summary.

trivia
A string containing trivia about the movie.

quotes
A string containing memorable quotes from the movie.

content_rating
A string specifying the movie's content rating.

content_rating_age
A string specifying the minumum age for viewers of the movie.

writers
A set of strings specifying the movie's writers.

directors
A set of strings specifying the movie's directors.

producers
A set of strings specifying the movie's producers.

countries
A set of strings specifying the countries involved in the production of the movie.

posters
A container of proxy objects representing the movie's posters. See below for information about proxy objects.

art
A container of proxy objects representing the movie's background art. See below for information about proxy objects.

themes
A container of proxy objects representing the movie's theme music. See below for information about proxy objects.

class Episode
Represents an episode of a TV show or other episodic content.

title
A string specifying the episode's title.

summary
A string specifying the episode's summary.

originally_available_at
A `date` object specifying when the episode originally aired.

rating
An integer attribute with a valuebetween 0 and 10 specifying the episode's rating.

writers
A set of strings specifying the episode's writers.

directors
A set of strings specifying the episode's directors.

producers
A set of strings specifying the episode's producers.

guest_stars
A set of strings specifying the episode's guest stars.

absolute_index
An integer specifying the absolute index of the episode within the entire series.

thumbs
A container of proxy objects representing the episode's thumbnail images. See

below for information about proxy objects.

duration
An integer specifying the duration of the episode, in milliseconds.

class Season
Represents a season of a TV show.

summary
A string specifying the season's summary.

posters
A container of proxy objects representing the season's posters. See below for information about proxy objects.

banners
A container of proxy objects representing the season's banner images. See below for information about proxy objects.

episodes
A map of Episode (page 41) objects.

class TV_Show
Represents a TV show, or the top-level of other episodic content.

genres
A set of strings specifying the show's genres.

tags
A set of strings specifying the show's tags.

collections
A set of strings specifying the show's collections.

duration
An integer specifying the approximate duration of each episode in the show, in milliseconds.

rating
A float between 0 and 10 specifying the show's rating.

title
A string specifying the show's title.

summary
A string specifying the show's summary.

originally_available_at
A date object specifying the date the show originally started airing,

content_rating
A string specifying the show's content rating.

studio
A string specifying the studio that produced the show.

countries
A set of strings specifying the countries involved in the production of the show.

posters
A container of proxy objects representing the show's posters. See below for information about proxy objects.

banners
A container of proxy objects representing the show's banner images. See below for information about proxy objects.

art
A container of proxy objects representing the show's banner images. See below for information about proxy objects.

themes
A container of proxy objects representing the show's theme music. See below for information about proxy objects.

class Artist
Represents an artist or group.

genres
A set of strings specifying the artist's genres.

tags
A set of strings specifying the artist's tags.

collections
A set of strings specifying the collections the artist belongs to.

rating
A float between 0 and 10 specifying the artist's rating.

title
A string specifying the artist's name.

summary
A string specifying the artist's biography.

posters
A container of proxy objects representing the artist's posters. See below for information about proxy objects.

art
A container of proxy objects representing the artist's background art. See below for information about proxy objects.

themes
A container of proxy objects representing the artist's theme music. See below for information about proxy objects.

class Album
Represents a music album.

genres
A list of strings specifying the album's genres.

tags
A list of strings specifying the album's tags.

collections
..todo:: Describe

rating
A float between 0 and 10 specifying the album's rating.

original_title
A string specifying the album's original title.

title
A string specifying the album's title.

summary
A string specifying the album's summary.

studio
A string specifying the album's studio.

originally_available_at
A `date` object specifying the album's original release date.

producers
A list of strings specifying the album's producers.

countries
A list of strings specifying the countries involved in the production of the album.

posters
A container of proxy objects representing the album's covers. See below for information about proxy objects.

tracks
A map of `Track` (page 44) objects representing the album's tracks.

class Track
Represents an audio track (e.g. music, audiobook, podcast, etc.)

name
A string specifying the track's name.

5.4.1 Proxy objects

Proxy objects are used for associating image and audio files with metadata.

Metadata objects will often have many posters, background images or theme music tracks available, but the user will usually only require one of each. This makes downloading the data for each possible choice time-consuming and places undue burden on the servers providing the data.

To remedy this, the framework uses proxy objects to reduce the amount of data that needs to be downloaded during the initial metadata update. There are two types of proxy available:

`Proxy.`**`Preview`**(*data*, *sort_order=None*)
: This proxy should be used when a preview version of the final piece of media is available, e.g. a thumbnail image for a high-resolution poster, or a short clip of a theme music file. When the user selects a piece of media referred to by a preview proxy, the media server will automatically download the final piece of media for use by clients.

`Proxy.`**`Media`**(*data*, *sort_order=None*)
: This proxy should be used when no preview version of the media is available, i.e. the data is the real media file data.

Using proxies is simple. The proxy should be assigned to a proxy container attribute, using the media's full URL as the key:

```
metadata.posters[full_poster_url] = Proxy.Preview(poster_thumbnail_data)
```

The `sort_order` attribute can be set to specify the order in which the possible choices are presented to the user. Items with lower sort order values are listed first.

CHAPTER

SIX

API REFERENCE

Contents:

6.1 Standard API

6.1.1 Controlling the runtime environment

All plug-in code executes inside a runtime environment provided by the framework. This performs a variety of tasks automatically, including communicating with the media server, creating contexts in which to handle incoming requests, and formatting responses for consumption by clients.

The framework provides a number of APIs for controlling the runtime environment and accessing contextual request information.

Global functions

The following global functions are provided:

handler(*prefix*, *name*, *thumb="icon-default.png"*, *art="art-default.png"*)

This function should always be used as a decorator. It enables the developer to register a function in their code as a prefix handler. Any subsequent requests with a matching prefix will be directed to that plug-in by the media server:

```
@handler('/video/example', 'Example')
def Main():
  pass
```

The decorated function definition should not include any required arguments.

Parameters

- **prefix** (*str* (http://docs.python.org/library/functions.html#str)) – The prefix at which to register the plug-in. All plug-ins should be registered under one of the four main top-level directories: `video`, `music`, `photos` or `applications`.

- **name** – The title of the registered prefix, to be displayed in the client user interface. This can be either a string, or the key of a localized string (see `Locale` (page 79) for more information).

route(*path*, *method='GET'*)

This function should always be used as a decorator. It allows functions to be assigned routes under a registered prefix (e.g. `/video/example/myRoute`). This enables the developer to give their plug-in a REST-like API with very little effort.

Parameters

- **path** (*str* (http://docs.python.org/library/functions.html#str)) – The path for the new route.
- **method** (*str* (http://docs.python.org/library/functions.html#str)) – The HTTP method the route should be assigned to, either `GET` or `PUT`.

Route paths can include keyword arguments enclosed in braces that will be passed to the function when it is executed:

```
@route('/video/example/myRoute/{x}')
def MyFunction(x):
  print x
```

Callback(*f*, *ext=None*, ***kwargs*)

Generates a callback path for the given function. A route will be used if one is available, otherwise an internal callback URL will be generated. Since these are not easily readable or modifiable, the developer should consider adding routes to their plug-in's functions if this is important. If a request is made for the returned URL, the given function will be executed with the arguments provided.

Here is a short example of using a callback, using classes explained in the next section:

```
@handler('/video/example', 'Example')
def FunctionA():
  oc = ObjectContainer(
    objects = [
      DirectoryObject(
        key = Callback(FunctionB, x='abc'),
        title = "Callback Example"
      )
    ]
  )
  return oc

def FunctionB(x):
  print 'FunctionB called:', x
  return None
```

indirect()

This function should only be used as a decorator. It is used to specify that a function does not return data directly, but instead returns an `ObjectContainer` (page 52) with

a single item referring to the final location of the data. This is useful when the server the stream resides on requires a specific user agent string or cookies before it will serve the media, and these can't be computed easily or quickly.

Using the decorator is simple:

```
@indirect
@route('/video/example/myRoute/play')
def Play(x):
  ...
```

The framework will automatically adjust generated callback paths and the XML returned to clients to indicate that the function does not return a direct response.

Plugin

The Plugin API includes methods and attributes for interacting with the global plug-in object, which provides view group management and manages the registration and calling of prefix of route handler functions.

Plugin.**Identifier**
A read-only attribute containing the identifier of the plug-in.

Return type str

Plugin.**Nice**(*value*)
Alters the plug-in's 'niceness' level, which affects how system resources are allocated. The higher the value, the fewer resources will be given to this plug-in, allowing other plug-ins and applications to make use of them instead.

The value should be between 0 (the default) and 20.

Parameters **value** (*int* (http://docs.python.org/library/functions.html#int)) – The level of 'niceness' to apply

Plugin.**Prefixes**
Returns a list of all prefixes currently registered by the plug-in.

rtype list

Plugin.**ViewGroups**

Plugin.**AddViewGroup**(*name*, *viewMode='List'*, *mediaType='items'*, *type=None*, *menu=None*, *cols=None*, *rows=None*, *thumb=None*, *summary=None*)

Plugin.**Traceback**(*msg='Traceback'*)

Route

The Route module provides additional methods for working with routes. While the *route* decorator above is sufficient for most uses, it is sometimes useful to be able to have more control over the routing system.

Route.**Connect**(*path*, *f*, *method='GET'*, ***kwargs*)
Provides equivalent functionality to the *route* decorator, but allows instance methods of objects to be added as routes as well as unbound functions.

Platform

The Platform API provides information about the server platform the plug-in is currently running on. While almost all plug-in code can be used cross-platform safely, it is sometimes desirable to perform platform-specific checks if, for example, the code needs to call external binaries only available on certain platforms, or the plug-in relies on a feature not supported by all versions of the media server.

Platform.**OS**
Reports the current server's operating system.

Returns The current platform; either *MacOSX*, *Windows* or *Linux*.

Return type str

Platform.**CPU**
Reports the current server's CPU architecture.

Returns The current CPU architecture; either *i386*, *MIPS*, *mips64* or *armv5tel*.

Return type str

Platform.**HasSilverlight**
Reports whether the server supports playback of Silverlight video content.

Return type bool

Request

The Request API provides access to the HTTP request currently being handled by the plug-in.

Note: These methods can only be used inside a request context.

Request.**Headers**
A dictionary of the HTTP headers received by the plug-in for the current request.

Return type dict

Response

The Response API allows the developer to modify the HTTP response that will be returned to the client. This is unnecessary for most plug-ins, but in some cases it can be useful to be able to specify additional response data.

Note: These methods can only be used inside a request context.

`Response.`**`Headers`**
A dictionary of keys and values that will be returned to the client as HTTP headers:

```
HTTP.Headers['MyHeader'] = 'MyValue'
```

`Response.`**`Status`**
An integer value specifying the HTTP status code (http://en.wikipedia.org/wiki/List_of_HTTP_status_codes) of the response.

Client

The Client API provides information about the client currently accessing the plug-in, allowing the developer to offer content selectively based on the capabilities of the device.

Note: These methods can only be used inside a request context.

`Client.`**`Platform`**
Reports the platform of the client currently accessing the plug-in.

Returns The client platform; one of the constants defined in `ClientPlatform` (page 88).

Return type str

`Client.`**`Protocols`**
Reports the protocols supported by the client currently accessing the plug-in.

Returns A list of protocols supported by the client, containing constants defined in `Protocol` (page 88).

Return type list

6.1.2 Objects

The framework's class library makes returning content to the media server or a client very simple and easy to understand. Using the classes provided, the developer can construct rich trees of objects using the same metadata types as Plex's own library system. The runtime environment will automatically convert the returned objects into the media server's standard XML format.

Object classes

Each object class includes a number of attributes. Attributes for each object can be set in two ways; either as part of the constructor, or by setting the attribute individually later on:

```
movie = MovieObject(
  title = "Movie Title",
  studio = "Movie Studio"
)

movie.tagline = "Movie Tagline"
```

Additionally, attributes can be set on the classes themselves. Any objects created afterwards will inherit the default attribute from the class:

```
MovieObject.studio = "Movie Studio"
```

The following classes are available:

- ObjectContainer (page 52)
- MovieObject (page 54)
- VideoClipObject (page 56)
- EpisodeObject (page 56)
- SeasonObject (page 57)
- TVShowObject (page 58)
- ArtistObject (page 59)
- AlbumObject (page 59)
- TrackObject (page 60)
- DirectoryObject (page 62)
- PopupDirectoryObject (page 62)
- InputDirectoryObject (page 63)
- PrefsObject (page 63)
- MediaObject (page 64)
- PartObject (page 65)

The attributes supported by each class are listed below.

Containers

class ObjectContainer(***kwargs*)

A container for other objects. *ObjectContainer* is the type most frequently returned to other applications. It provides clients with an ordered list of items in response to a request.

view_group

A string specifying the name of the view group the client should use when displaying the container's contents. This should be the name of a group previously registered with Plugin.AddViewGroup() (page 49).

content
Identifies the type of the objects inside the container. This attribute should be set to one of the container type constants identified here. ..todo:: Link to container types

art
A string specifying an image resource that should be used as the container's background art.

title1
A string specifying the first title to display in the user interface.

title2
A string specifying the second title to display in the user interface.

http_cookies
A string specifying any HTTP cookies that need to be passed to the server when attempting to play items from the container.

user_agent
A string specifying the user agent header that needs to be sent to the server when attempting to play items from the container.

no_history
A boolean specifying whether the container should be added to the client's history stack. For example, if Container *B* in the sequence `A => B => C` had *no_cache* set to `True`, navigating back from *C* would take the user directly to *A*.

replace_parent
A boolean specifying whether the container should replace the previous container in the client's history stack. For example, if Container *C* in the sequence `A => B => C` had *replace_parent* set to `True`, navigating back from *C* would take the user directly to *A*.

no_cache
A boolean indicating whether the container can be cached or not. Under normal circumstances, the client will cache a container for use when navigating back up the directory tree. If *no_cache* is set to `True`, the container will be requested again when navigating back.

mixed_parents
A boolean indicating that the objects in the container do not share a common parent object (for example, tracks in a playlist).

header

message
The *header* and *message* attributes are used in conjunction. They instruct the client to isplay a message dialog on loading the container, where *header* is the message dialog's title and *message* is the body.

add(*obj*)
Adds the object *obj* to the container. The container can also be populated by passing a list of objects to the constructor as the *objects* argument:

```
oc = ObjectContainer(
  objects = [
    MovieObject(
      title = "Movie"
    ),
    VideoClipObject(
      title = "Video Clip"
    )
  ]
)
```

len(container)
Containers can be passed to the `len()` (http://docs.python.org/library/functions.html#len) function to get the number of objects they contain.

Metadata objects

class MovieObject(***kwargs*)
Represents a movie (e.g. a theatrical release, independent film, home movie, etc.)

url
A string specifying the URL of the movie. If a URL service that matches the given URL is available, the *key* and *rating_key* attributes will be set and the list of media objects will be generated automatically.

key
A string specifying the path to the movie's full metadata object. This is usually a function callback generated using `Callback()` (page 48). The function should return an *ObjectContainer* containing a single metadata object with the maximum amount of metadata available.

Note: If the *url* attribute is set (invoking a URL service), the *key* attribute is set automatically.

---

rating_key
A string specifying a unique identifier for the movie. This unique value is used by the media server for maintaining play counts and providing other advanced features.

Note: If the *url* attribute is set (invoking a URL service), the *rating_key* attribute is set automatically.

---

genres
A list of strings specifying the movie's genre.

tags
A list of strings specifying the movie's tags.

duration
An integer specifying the duration of the movie, in milliseconds.

rating
A float between 0 and 10 specifying the movie's rating.

original_title
A string specifying the movie's original title.

source_title
A string specifying the source of the movie (e.g. `Netflix` or `YouTube`)

title
A string specifying the movie's title.

year
An integer specifying the movie's release year.

originally_available_at
A `date` object specifying the movie's original release date.

studio
A string specifying the movie's studio.

tagline
A string specifying the movie's tagline.

summary
A string specifying the movie's summary.

trivia
A string containing trivia about the movie.

quotes
A string containing memorable quotes from the movie.

content_rating
A string specifying the movie's content rating.

content_rating_age
A string specifying the minumum age for viewers of the movie.

writers
A list of strings specifying the movie's writers.

directors
A list of strings specifying the movie's directors.

producers
A list of strings specifying the movie's producers.

countries
A list of strings specifying the countries involved in the production of the movie.

thumb
A string specifying an image resource to use as the movie's thumbnail.

art
A string specifying an image resource to use as the movie's background art.

add(*obj*)
Adds the MediaObject (page 64) instance *obj* to the movie's item list. The items can also be populated by passing a list of objects to the constructor as the *items* argument:

```
m = MovieObject(
   items = [
      MediaObject(...)
   ]
)
```

class VideoClipObject(***kwargs*)
Represents a video clip (e.g. a YouTube or Vimeo video).

VideoClipObject has the same attributes as *MovieObject*.

class EpisodeObject(***kwargs*)
Represents an episode of a TV show or other episodic content.

url
A string specifying the URL of the movie. If a URL service that matches the given URL is available, the *key* and *rating_key* attributes will be set and the list of media objects will be generated automatically.

rating_key
A string specifying a unique identifier for the episode. This unique value is used by the media server for maintaining play counts and providing other advanced features.

Note: If the *url* attribute is set (invoking a URL service), the *rating_key* attribute is set automatically.

---

title
A string specifying the episode's title.

summary
A string specifying the episode's summary.

originally_available_at
A `date` object specifying when the episode originally aired.

rating
An integer attribute with a valuebetween 0 and 10 specifying the episode's rating.

writers
A list of strings specifying the episode's writers.

directors
A list of strings specifying the episode's directors.

producers
A list of strings specifying the episode's producers.

guest_stars
A list of strings specifying the episode's guest stars.

absolute_index
An integer specifying the absolute index of the episode within the entire series.

key
A string specifying the path to the episode's full metadata object. This is usually a function callback generated using `Callback()` (page 48). The function should return an *ObjectContainer* containing a single metadata object with the maximum amount of metadata available.

Note: If the *url* attribute is set (invoking a URL service), the *key* attribute is set automatically.

---

show
A string identifying the show the episode belongs to.

season
An integer identifying the season the episode belongs to.

thumb
A string specifying an image resource to use as the episode's thumbnail.

art
A string specifying an image resource to use as the episode's background art.

source_title
A string specifying the source of the episode (e.g. `Netflix` or `YouTube`)

duration
An integer specifying the duration of the episode, in milliseconds.

add(*obj*)
Adds the `MediaObject` (page 64) instance *obj* to the episode's item list. The items can also be populated by passing a list of objects to the constructor as the *items* argument:

```
obj = EpisodeObject(
   items = [
      MediaObject(...)
   ]
)
```

class **SeasonObject**(***kwargs*)
Represents a season of a TV show.

summary
A string specifying the season's summary.

key
A string specifying the path to a container representing the season's content. This is usually a function callback generated using `Callback()` (page 48).

rating_key
A string specifying a unique identifier for the season. This unique value is used by

the media server for maintaining play counts and providing other advanced features.

index
An integer specifying the season's index.

title
A string specifying the title to display for the season in the user interface.

show
A string specifying the show the season belongs to.

episode_count
An integer specifying the number of episodes in the season.

source_title
A string specifying the source of the season (e.g. `Netflix` or `YouTube`)

thumb
A string specifying an image resource to use as the season's thumbnail.

art
A string specifying an image resource to use as the season's background art.

class TVShowObject (***kwargs*)
Represents a TV show, or the top-level of other episodic content.

key
A string specifying the path to a container representing the show's content. This is usually a function callback generated using `Callback()` (page 48).

rating_key
A string specifying a unique identifier for the show. This unique value is used by the media server for maintaining play counts and providing other advanced features.

genres
A list of attributes specifying the show's genres.

tags
A list of attributes specifying the show's tags.

duration
An integer specifying the approximate duration of each episode in the show, in milliseconds.

rating
A float between 0 and 10 specifying the show's rating.

source_title
A string specifying the source of the show (e.g. `Netflix` or `YouTube`)

title
A string specifying the show's title.

summary
A string specifying the show's summary.

originally_available_at
A `date` object specifying the date the show originally started airing,

content_rating
A string specifying the show's content rating.

studio
A string specifying the studio that produced the show.

countries
A list of strings specifying the countries involved in the production of the show.

thumb
A string specifying an image resource to use as the show's thumbnail.

art
A string specifying an image resource to use as the show's background art.

episode_count
An integer specifying the TV show's total number of episodes.

class ArtistObject (***kwargs*)
Represents an artist or group.

key
A string specifying the path to a container representing the artist's content. This is usually a function callback generated using `Callback()` (page 48).

genres
A list of strings specifying the artist's genres.

tags
A list of strings specifying the artist's tags.

duration
An integer specifying the total duration of the artist's albums, in milliseconds.

rating
A float between 0 and 10 specifying the artist's rating.

source_title
A string specifying the source of the artist (e.g. `Rhapsody` or `Spotify`)

title
A string specifying the artist's name.

summary
A string specifing the artist's biography.

thumb
A string specifying an image resource to use as the artist's thumbnail.

art
A string specifying an image resource to use as the artist's background art.

track_count
An integer specifying the total number of tracks available for the artist.

class AlbumObject(***kwargs*)

Represents a music album.

key
A string specifying the path to a container representing the album's tracks. This is usually a function callback generated using `Callback()` (page 48).

genres
A list of strings specifying the album's genres.

tags
A list of strings specifying the album's tags.

duration
An integer specifying the duration of the album, in milliseconds.

rating
A float between 0 and 10 specifying the album's rating.

original_title
A string specifying the album's original title.

source_title
A string specifying the source of the album (e.g. `Rhapsody` or `Spotify`)

artist
A string specifying the artist the album belongs to.

title
A string specifying the album's title.

summary
A string specifying the album's summary.

studio
A string specifying the album's studio.

originally_available_at
A `date` object specifying the album's original release date.

producers
A list of strings specifying the album's producers.

countries
A list of strings specifying the countries involved in the production of the album.

track_count
An integer value specifying the number of tracks the album contains.

thumb
A string specifying an image resource to use as the album's thumbnail.

art
A string specifying an image resource to use as the album's background art.

class TrackObject(***kwargs*)

Represents an audio track (e.g. music, audiobook, podcast, etc.)

url
A string specifying the URL of the track. If a URL service that matches the given URL is available, the *key* and *rating_key* attributes will be set and the list of media objects will be generated automatically.

key
A string specifying the path to the track's full metadata object. This is usually a function callback generated using `Callback()` (page 48). The function should return an *ObjectContainer* containing a single metadata object with the maximum amount of metadata available.

Note: If the *url* attribute is set (invoking a URL service), the *key* attribute is set automatically.

---

title
A string specifying the track's title.

rating_key
A string specifying a unique identifier for the track. This unique value is used by the media server for maintaining play counts and providing other advanced features.

Note: If the *url* attribute is set (invoking a URL service), the *rating_key* attribute is set automatically.

---

index
An integer specifying the track's index in the parent album.

artist
A string specifying the artist the track belongs to.

album
A string specifying the album the track belongs to.

genres
A list of strings specifying the track's genres.

tags
A list of strings specifying the track's tags.

duration
An integer specifying the duration of the track, in milliseconds.

rating
A float between 0 and 10 specifying the track's rating.

source_title
A string specifying the source of the track (e.g. `Rhapsody` or `Spotify`)

thumb
A string specifying an image resource to use as the track's thumbnail.

art
A string specifying an image resource to use as the track's background art.

add(*obj*)
Adds the MediaObject (page 64) instance *obj* to the track's item list. The items can also be populated by passing a list of objects to the constructor as the *items* argument:

```
obj = TrackObject(
   items = [
      MediaObject(...)
   ]
)
```

Generic browsing objects

class DirectoryObject(***kwargs*)
Represents a generic container of objects. Directory objects are usually used when creating a navigation hierarchy.

key
A string specifying the path to a container representing the directory's content. This is usually a function callback generated using Callback() (page 48).

title
A string specifying the directory's title.

tagline
A string specifying the directory's tagline.

summary
A string specifying the directory's summary.

thumb
A string specifying an image resource to use as the directory's thumbnail.

art
A string specifying an image resource to use as the directory's background art.

duration
An integer specifying the duration of the objects provided by the directory, in milliseconds.

class PopupDirectoryObject(***kwargs*)
Similar to *DirectoryObject* above. The only difference is in the presentation of the directory's content on the client - *PopupDirectoryObjects* are presented as a pop-up menu where possible, and are not added to the client's history stack.

key
A string specifying the path to a container representing the directory's content. This is usually a function callback generated using Callback() (page 48).

title
A string specifying the directory's title.

tagline
A string specifying the directory's tagline.

summary
A string specifying the directory's summary.

thumb
A string specifying an image resource to use as the directory's thumbnail.

art
A string specifying an image resource to use as the directory's background art.

duration
An integer specifying the duration of the objects provided by the directory, in milliseconds.

class `InputDirectoryObject` (***kwargs*)
Represents a container of objects generated from a query inputted by the user. The client will display an input dialog with the given prompt.

key
A string specifying the path to a container representing the directory's content. This is usually a function callback generated using `Callback()` (page 48). The function must accept a *query* argument, which contains the query entered by the user.

title
A string specifying the directory's title.

tagline
A string specifying the directory's tagline.

summary
A string specifying the directory's summary.

thumb
A string specifying an image resource to use as the directory's thumbnail.

art
A string specifying an image resource to use as the directory's background art.

prompt
A string specifying the prompt that should be displayed to the user in the input dialog displayed after selecting the object.

class `PrefsObject` (***kwargs*)
Represents an item that invokes the user preferences dialog when selected.

title
A string specifying the object's title.

tagline
A string specifying the object's tagline.

summary
A string specifying the object's summary.

thumb
A string specifying an image resource to use as the object's thumbnail.

art
A string specifying an image resource to use as the object's background art.

Media objects

class MediaObject(***kwargs*)
Represents a single piece of media. Each metadata object may have multiple media objects associated with it. All media objects for the metadata object should refer to the same media, but in different formats, resolutions and/or containers (where available), allowing the client to choose the most appropriate item for playback.

protocols
A list of strings specifying the protocols that the server and client must support in order to play the media directly. The list should contain values defined in `Protocol` (page 88).

platforms
A list of strings specifying the client platforms the media item should be made available to. If the list is empty, the item will be available on all platforms. The list should contain values defined in `Platform` (page 50).

bitrate
An integer specifying the media's bitrate.

aspect_ratio
An integer specifying the media's aspect ratio (`width / height`).

audio_channels
An integer specifying the number of audio channels the media has (e.g. `2` for stereo, `6` for 5.1).

audio_codec
A string specifying the media's audio codec. This attribute should be set to one of the constants defined in `AudioCodec` (page 89).

video_codec
A string specifying the media's video codec. This attribute should be set to one of the constants defined in `VideoCodec` (page 89).

video_resolution
An integer specifying the vertical resolution of the video.

container
A string specifying the media's container. This attribute should be set to one of the constants defined in `Container` (page 89).

video_frame_rate
An integer representing the frame rate of the media.

duration
An integer specifying the duration of the media, in milliseconds.

add(*obj*)
Adds the `PartObject` (page 65) instance *obj* to the media object's list of parts. The media object can also be populated by passing a list of objects to the constructor as the *parts* argument:

```
media = MediaObject(
  parts = [
    PartObject(...)
  ]
)
```

class PartObject(***kwargs*)
Represents a part of a piece of media. Most *MediaObject* instances will usually only have one part, but can contain more if the media is split across several files or URLs. The client will play each part contiguously.

key
A string specifying the location of the part. This can be an absolute URL, or a callback to another function generated using `Callback()` (page 48).

duration
An integer specifying the duration of the part, in milliseconds.

URL generating functions

The framework also provides three functions for handling URLs that require Plex's WebKit playback capabilities. To instruct the client to play the given URL using WebKit, the developer can simply use one of the functions below to set the value of a `PartObject` (page 65)'s key:

```
PartObject(
   key = WebVideoURL(url)
)
```

WebVideoURL(*url*)
Returns a URL that starts WebKit playback of the given web page URL. There must be a matching *site configuration* (page 13).

RTMPVideoURL(*url*, *clip=None*, *clips=None*, *width=None*, *height=None*, *live=False*)
Returns a URL that starts WebKit playback of the given RTMP URL using Plex's hosted RTMP player.

WindowsMediaVideoURL(*url*, *width=None*, *height=None*)
Returns a URL that starts WebKit playback of the given Windows Media URL using Plex's hosted Silverlight player.

6.1.3 Accessing bundled resources

Resource

The Resource API provides methods that can be used to load files contained in the bundle's Resources directory.

`Resource.`**`ExternalPath`**(*itemname*)
Generates an externally accessible path for the named resource. This method is usually used to assign bundled images to object attributes.

Note: The alias `R()` is equivalent to this method.

`Resource.`**`SharedExternalPath`**(*itemname*)
Generates an externally accessible path for the named shared resource. A list of valid shared resource names is provided below.

Note: The alias `S()` is equivalent to this method.

`Resource.`**`Load`**(*itemname*, *binary=True*)
Loads the named resource file and returns the data as a string.

`Resource.`**`LoadShared`**(*itemname*, *binary=True*)
Loads the named shared resource file and returns the data as a string.

`Resource.`**`GuessMimeType`**(*path*)
Attempts to guess the MIME type of a given path based on its extension.

`Resource.`**`AddMimeType`**(*mimetype*, *extension*)
Assigns the specified MIME type to the given extension.

6.1.4 Threading

The framework uses threading extensively under the hood to handle multiple concurrent requests and perform certain sets of tasks concurrently. This is sufficient for most plug-ins; however, in certain cases the developer may need to exert more control over the execution of their code. To enable this, a specific API for threading is provided.

Thread

The Thread API allows the developer to create multiple threads of execution, and provides various methods of synchronization.

`Thread.`**`Create`**(*f*, *globalize=True*, **args*, ***kwargs*)
Creates a new thread which executes the given function, using the arguments and keyword arguments provided.

Parameters **f** (*function*) – The function to execute.

Note: By default, threads are globalized (removed from the current request context and executed in the global context - see *contexts*). If the developer chooses not to globalize a thread, it is their responsibility to ensure the main request thread stays alive while the new thread is running to avoid the request context being released too early.

`Thread.`**`CreateTimer`**(*interval*, *f*, *globalize=True*, **args*, ***kwargs*)

Similar to the *Create* method above, but executes the function in a background thread after the given time interval rather than executing it immediately.

Parameters

- **interval** (*float* (http://docs.python.org/library/functions.html#float)) – The number of seconds to wait before executing the function
- **f** (*function*) – The function to execute.

`Thread.`**`Sleep`**(*interval*)

Pauses the current thread for a given time interval.

Parameters **interval** (*float* (http://docs.python.org/library/functions.html#float)) – The number of seconds to sleep for.

`Thread.`**`Lock`**(*key=None*)

Returns the lock object with the given key. If no named lock object exists, a new one is created and returned. If no key is provided, a new, anonymous lock object is returned.

Parameters **key** (*str or None*) – The key of the named lock to return.

Return type Lock (http://docs.python.org/library/threading.html#lock-objects)

`Thread.`**`AcquireLock`**(*key*)

Acquires the named lock for the given key.

Parameters **key** (*str* (http://docs.python.org/library/functions.html#str)) – The key of the named lock to acquire.

Note: It is the developer's responsibility to ensure the lock is released correctly.

`Thread.`**`ReleaseLock`**(*key*)

Releases the named lock for the given key.

Parameters **key** (*str* (http://docs.python.org/library/functions.html#str)) – The key of the named lock to acquire.

`Thread.`**`Event`**(*key=None*)

Returns the event object with the given key. If no named event object exists, a new one is created and returned. If no key is provided, a new, anonymous event object is returned.

Parameters **key** (*str or None*) – The key of the named event to return.

Return type Event (http://docs.python.org/library/threading.html#event-objects)

Thread.**Block**(*key*)
Clears a named event, causing any threads that subsequently wait for the event will block.

Parameters **key** (*str* (http://docs.python.org/library/functions.html#str)) – The key of the named event.

Thread.**Unblock**(*key*)
Sets a named event, unblocking any threads currently waiting on it.

Parameters **key** (*str* (http://docs.python.org/library/functions.html#str)) – The key of the named event.

Thread.**Wait**(*key*, *timeout=None*)
If the named event is unset, this method causes the current thread to block until the event is set, or the timeout interval elapses (if provided).

Parameters

- **key** (*str* (http://docs.python.org/library/functions.html#str)) – The key of the named event.
- **timeout** (*float* (http://docs.python.org/library/functions.html#float)) – The number of seconds to wait before aborting.

Returns True if the named event was set while waiting, or False if the timeout occurred first.

Return type bool

Thread.**Semaphore**(*key=None*, *limit=1*)
Returns the semaphore object with the given key. If no named semaphore object exists, a new one is created and returned. If no key is provided, a new, anonymous semaphore object is returned.

Parameters **key** (*str or None*) – The key of the named semaphore to return.

Return type Semaphore (http://docs.python.org/library/threading.html#semaphore-objects)

6.1.5 Networking

Communicating with the outside world is one of the most common things plug-ins need to do. The framework includes a number of features that make networking as simple as possible. The HTTP methods in particular simplify the underlying request mechanism, providing features such as automatic caching and cookie support.

Network

Network.**Address**
Returns the server's IP address. Unless the server is connected directly to the internet, this will usually be a local IP address.

Return type str

`Network.`**`PublicAddress`**
Returns the public IP address of the network the server is currently connected to.

Return type str

`Network.`**`Hostname`**
Returns the server's hostname.

Return type str

`Network.`**`Socket`**`()`
Creates a new *socket* object.

Return type socket (http://docs.python.org/library/socket.html#socket-objects)

`Network.`**`SSLSocket`**`()`
Creates a new *SSLSocket* object. These objects operate almost identically to the *socket* objects returned by the above method, but support SSL connections. The only additional requirement is that the *do_handshake()* method is called after establishing a connection.

HTTP

The HTTP API provides methods for making HTTP requests and interacting with the framework's HTTP subsystem.

`HTTP.`**`Request`**(*url*, *values=None*, *headers={}*, *cacheTime=None*, *encoding=None*, *errors=None*, *timeout=<object object at 0x10042bc40>*, *immediate=False*, *sleep=0*, *data=None*)
Creates and returns a new `HTTPRequest` (page 70) object.

Parameters

- **url** (*str* (http://docs.python.org/library/functions.html#str)) – The URL to use for the request.
- **values** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Keys and values to be URL encoded and provided as the request's POST body.
- **headers** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Any custom HTTP headers that should be added to this request.
- **cacheTime** (*int* (http://docs.python.org/library/functions.html#int)) – The maximum age (in second) of cached data before it should be considered invalid.
- **timeout** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum amount of time (in seconds) to wait for the request to return a response before timing out.
- **immediate** (*bool* (http://docs.python.org/library/functions.html#bool)) – If set to `True`, the HTTP request will be made immediately when the object is created (by default, requests are delayed until the data they return is requested).

- **sleep** (*float* (http://docs.python.org/library/functions.html#float)) – The amount of time (in seconds) that the current thread should be paused for after issuing a HTTP request. This is to ensure undue burden is not placed on the server. If the data is retrieved from the cache, this value is ignored.
- **data** (*str* (http://docs.python.org/library/functions.html#str)) – The raw POST data that should be sent with the request. This attribute cannot be used in conjunction with *values*.

HTTP.**GetCookiesForURL**(*url*)

HTTP.**SetPassword**(*url*, *username*, *password*, *realm=None*)

HTTP.**PreCache**(*url*, *values=None*, *headers={}*, *cacheTime=None*, *encoding=None*, *errors=None*)

Instructs the framework to pre-cache the result of a given HTTP request in a background thread. This method returns nothing - it is designed to ensure that cached data is available for future calls to *HTTP.Request*.

HTTP.**ClearCookies**()

Clears the plug-in's HTTP cookies.

HTTP.**ClearCache**()

Clears the plug-in's HTTP cache.

HTTP.**RandomizeUserAgent**(*browser=None*)

Sets the plug-in's user agent string to a random combination of browser & operating system versions.

Parameters **browser** (*str* (http://docs.python.org/library/functions.html#str)) – The browser to generate a user agent string for. Either `Firefox` or `Safari`, or `None` to randomly select the browser used.

HTTP.**CacheTime**

The default cache time (in seconds) used for all HTTP requests without a specific cache time set. By default, this value is 0.

Type float

HTTP.**Headers**

A dictionary containing the default HTTP headers that should be issued with requests:

```
HTTP.Headers['User-Agent'] = 'My Plug-in'
```

class HTTP.**HTTPRequest**

This class cannot be instantiated directly. These objects are returned from `HTTP.Request()` (page 69). They encapsulate information about a pending HTTP request, and issue it when necessary.

load()

Instructs the object to issue the HTTP request, if it hasn't done so already.

headers

A dictionary of HTTP headers returned by the server in response to the request.

content
A string containing the response's body.

XMLRPC

The XMLRPC API provides access to Python's xmlrpclib (http://docs.python.org/library/xmlrpclib.html#module-xmlrpclib) module, allowing developers to easily communicate with XMLRPC services.

XMLRPC.**Proxy**(*url*, *encoding=None*)
Returns an xmlrpclib.ServerProxy (http://docs.python.org/library/xmlrpclib.html#xmlrpclib.Ser object for the given *url*. The framework's default HTTP headers will be used.

6.1.6 Parsing and manipulating data

The most common requirement for plug-ins is the ability to retrieve data, manipulate that data to extract meaningful content from it, and return that content to either a client or the media server itself. Because data available online comes in a variety of formats, the framework includes a number of data parsing libraries capable of handling the most common types of data format and provides simple APIs for converting between these formats and regular Python objects.

XML

The XML API provides methods for converting between XML-formatted strings and trees of XML Element objects. New XML element trees can also be constructed. The underlying functionality is provided by the lxml etree (http://lxml.de/tutorial.html#the-element-class) and objectify (http://lxml.de/objectify.html) libraries.

Note: It is strongly recommended that developers read lxml's XPath Tutorial (http://lxml.de/xpathxslt.html#xpath). Manipulating elements returned by the etree library using XPath is a very powerful way of finding and accessing data within a XML document. Learning to use XPath efficiently will greatly simplify the plug-in's code.

XML.**Element**(*name*, *text=None*, ***kwargs*)
Returns a new XML element with the given name and text content. Any keyword arguments provided will be set as attributes.

Parameters

- **name** (*str* (http://docs.python.org/library/functions.html#str)) – The name of the new element.
- **text** (*str* (http://docs.python.org/library/functions.html#str)) – The text content of the new element.

Return type _Element (http://lxml.de/api/lxml.etree._Element-class.html)

XML.**StringFromElement**(*el*, *encoding='utf8'*)
Converts the XML element object *el* to a string representation using the given encoding.

XML.**ElementFromString**(*string*, *encoding=None*)
Converts *string* to a HTML element object.

Return type

_Element (http://lxml.de/api/lxml.etree._Element-class.html)

XML.**ElementFromURL**(*url*, *values=None*, *headers={}*, *cacheTime=None*, *encoding=None*, *errors=None*, *timeout=<object object at 0x10042bc40>*, *sleep=0*)
Retrieves the content for a given HTTP request and parses it as XML using the above method.

Parameters

- **url** (*str* (http://docs.python.org/library/functions.html#str)) – The URL to retrieve content from.
- **values** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Values to pass as URL encoded content for a POST request.
- **headers** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Custom HTTP headers to add to the request.
- **cacheTime** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum age (in seconds) that cached data should still be considered valid.
- **timeout** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum amount of time (in seconds) that the framework should wait for a response before aborting.
- **sleep** (*float* (http://docs.python.org/library/functions.html#float)) – The number of seconds the current thread should pause for if a network request was made, ensuring undue burden isn't placed on web servers. If cached data was used, this value is ignored.

XML.**StringFromObject**(*obj*, *encoding='utf-8'*)
Attempts to create objectified XML from the given object.

XML.**ObjectFromString**(*string*)
Parses *string* as XML-formatted content and attempts to build a Python object using the objectify library.

Return type ObjectifiedElement (http://lxml.de/api/lxml.objectify.ObjectifiedElement-class.html)

XML.**ObjectFromURL**(*url*, *values=None*, *headers={}*, *cacheTime=None*, *autoUpdate=False*, *encoding=None*, *errors=None*, *timeout=<object object at 0x10042bc40>*, *sleep=0*)
Retrieves the content for a given HTTP request and parses it as objectified XML using the above method.

Parameters

- **url** (*str* (http://docs.python.org/library/functions.html#str)) – The URL to retrieve content from.
- **values** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Values to pass as URL encoded content for a POST request.
- **headers** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Custom HTTP headers to add to the request.
- **cacheTime** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum age (in seconds) that cached data should still be considered valid.
- **timeout** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum amount of time (in seconds) that the framework should wait for a response before aborting.
- **sleep** (*float* (http://docs.python.org/library/functions.html#float)) – The number of seconds the current thread should pause for if a network request was made, ensuring undue burden isn't placed on web servers. If cached data was used, this value is ignored.

HTML

The HTML API is similar to the XML API, but is better suited to parsing HTML content. It is powered by the lxml html (http://lxml.de/lxmlhtml.html) library.

`HTML.`**`Element`**(*name*, *text=None*, ***kwargs*)

Returns a new HTML element with the given name and text content. Any keyword arguments provided will be set as attributes.

Parameters

- **name** (*str* (http://docs.python.org/library/functions.html#str)) – The name of the new element.
- **text** (*str* (http://docs.python.org/library/functions.html#str)) – The text content of the new element.

Return type HtmlElement (http://lxml.de/api/lxml.html.HtmlElement-class.html)

`HTML.`**`StringFromElement`**(*el*, *encoding='utf8'*)

Converts the HTML element object *el* to a string representation using the given encoding.

`HTML.`**`ElementFromString`**(*string*)

Converts *string* to a HTML element object.

Return type

HtmlElement (http://lxml.de/api/lxml.html.HtmlElement-class.html)

HTML.**ElementFromURL**(*url*, *values=None*, *headers={}*, *cacheTime=None*, *encoding=None*, *errors=None*, *timeout=<object object at 0x10042bc40>*, *sleep=0*)

Retrieves the content for a given HTTP request and parses it as HTML using the above method.

Parameters

- **url** (*str* (http://docs.python.org/library/functions.html#str)) – The URL to retrieve content from.
- **values** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Values to pass as URL encoded content for a POST request.
- **headers** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Custom HTTP headers to add to the request.
- **cacheTime** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum age (in seconds) that cached data should still be considered valid.
- **timeout** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum amount of time (in seconds) that the framework should wait for a response before aborting.
- **sleep** (*float* (http://docs.python.org/library/functions.html#float)) – The number of seconds the current thread should pause for if a network request was made, ensuring undue burden isn't placed on web servers. If cached data was used, this value is ignored.

JSON

The JSON API provides methods for easily converting JSON-formatted strings into Python objects, and vice versa.

More information about the JSON format can be found here (http://www.json.org/).

Note: The framework includes two JSON parsers - one is fast, but very strict, while the second is slower but more tolerant of errors. If a string is unable to be parsed by the fast parser, an error will be logged indicating the position in the string where parsing failed. If possible, the developer should check for and resolve these errors, as slow JSON parsing can have a severely detrimental effect on performance, especially on embedded systems.

JSON.**StringFromObject**(*obj*)

Converts the given object to a JSON-formatted string representation.

JSON.**ObjectFromString**(*string*, *encoding=None*)

Converts a JSON-formatted string into a Python object, usually a dictionary.

JSON.**ObjectFromURL**(*url*, *values=None*, *headers={}*, *cacheTime=None*, *encoding=None*, *errors=None*, *timeout=<object object at 0x10042bc40>*, *sleep=0*)

Retrieves the content for a given HTTP request and parses it as JSON-formatted content using the above method.

Parameters

- **url** (*str* (http://docs.python.org/library/functions.html#str)) – The URL to retrieve content from.
- **values** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Values to pass as URL encoded content for a POST request.
- **headers** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Custom HTTP headers to add to the request.
- **cacheTime** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum age (in seconds) that cached data should still be considered valid.
- **timeout** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum amount of time (in seconds) that the framework should wait for a response before aborting.
- **sleep** (*float* (http://docs.python.org/library/functions.html#float)) – The number of seconds the current thread should pause for if a network request was made, ensuring undue burden isn't placed on web servers. If cached data was used, this value is ignored.

YAML

YAML.**ObjectFromString**(*string*)

Parses the given YAML-formatted string and returns the object it represents.

YAML.**ObjectFromURL**(*url*, *values=None*, *headers={}*, *cacheTime=None*, *encoding=None*, *errors=None*, *timeout=<object object at 0x10042bc40>*, *sleep=0*)

Retrieves the content for a given HTTP request and parses it as YAML-formatted content using the above method.

Parameters

- **url** (*str* (http://docs.python.org/library/functions.html#str)) – The URL to retrieve content from.
- **values** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Values to pass as URL encoded content for a POST request.
- **headers** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Custom HTTP headers to add to the request.
- **cacheTime** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum age (in seconds) that cached data should still be considered valid.

- **timeout** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum amount of time (in seconds) that the framework should wait for a response before aborting.
- **sleep** (*float* (http://docs.python.org/library/functions.html#float)) – The number of seconds the current thread should pause for if a network request was made, ensuring undue burden isn't placed on web servers. If cached data was used, this value is ignored.

RSS

The RSS API provides methods for parsing content from RSS, RDF and ATOM feeds. The framework includes the excellent Universal Feed Parser (http://www.feedparser.org) library to achieve this functionality.

For more information about the objects returned by the feed parser, please consult the documentation (http://www.feedparser.org/docs/) here.

RSS.**FeedFromString**(*string*)
Parses the given string as an RSS, RDF or ATOM feed (automatically detected).

RSS.**FeedFromURL**(*url*, *values=None*, *headers={}*, *cacheTime=None*, *autoUpdate=False*, *encoding=None*, *errors=None*, *timeout=<object object at 0x10042bc40>*, *sleep=0*)
Retrieves the content for a given HTTP request and parses it as an RSS, RDF or ATOM feed using the above method.

Parameters

- **url** (*str* (http://docs.python.org/library/functions.html#str)) – The URL to retrieve content from.
- **values** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Values to pass as URL encoded content for a POST request.
- **headers** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Custom HTTP headers to add to the request.
- **cacheTime** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum age (in seconds) that cached data should still be considered valid.
- **timeout** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum amount of time (in seconds) that the framework should wait for a response before aborting.
- **sleep** (*float* (http://docs.python.org/library/functions.html#float)) – The number of seconds the current thread should pause for if a network request was made, ensuring undue burden isn't placed on web servers. If cached data was used, this value is ignored.

Plist

The Plist API greatly simplifies handling content in Apple's XML-based property list format. Using these methods, data can easily be converted between property lists and regular Python objects. The top-level object of a property list is usually a dictionary.

More information about the property list format can be found here (http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man5/plist.5.html).

`Plist.`**`StringFromObject`**`(`*obj*`)`
Converts a given object to a Plist-formatted string representation.

`Plist.`**`ObjectFromString`**`(`*string*`)`
Returns an object representing the given Plist-formatted string.

`Plist.`**`ObjectFromURL`**`(`*url*, *values=None*, *headers={}*, *cacheTime=None*, *encoding=None*, *errors=None*, *timeout=<object object at 0x10042bc40>*, *sleep=0*`)`
Retrieves the content for a given HTTP request and parses it as a Plist using the above method.

Parameters

- **url** (*str* (http://docs.python.org/library/functions.html#str)) – The URL to retrieve content from.
- **values** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Values to pass as URL encoded content for a POST request.
- **headers** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Custom HTTP headers to add to the request.
- **cacheTime** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum age (in seconds) that cached data should still be considered valid.
- **timeout** (*float* (http://docs.python.org/library/functions.html#float)) – The maximum amount of time (in seconds) that the framework should wait for a response before aborting.
- **sleep** (*float* (http://docs.python.org/library/functions.html#float)) – The number of seconds the current thread should pause for if a network request was made, ensuring undue burden isn't placed on web servers. If cached data was used, this value is ignored.

6.1.7 Data storage and caching

It is often useful for developers to be able to store data for use by the plug-in later on. To assist with this, the framework provides three methods of data storage, enabling the developer to choose the method most suitable for their requirements.

Data

The Data API provides methods for storing and retrieving arbitrary data. Plug-ins are generally not allowed to access the user's file system directly, but can use these methods to load and save files in a single directory assigned by the framework.

The data storage locations are unique for each plug-in, preventing one plug-in from modifying data stored by another.

Note: The actual location of data files stored by these methods may vary between platforms.

`Data.`**`Load`**`(`*item*`)`
Loads a previously stored binary data item.

Parameters **item** (*str* (http://docs.python.org/library/functions.html#str)) – The name of the data item to load.

Returns The contents of the data item stored on disk.

Return type str

`Data.`**`Save`**`(`*item*, *data*`)`
Stores binary data as a data item with the given name.

Parameters

- **item** (*str* (http://docs.python.org/library/functions.html#str)) – The name of the data item to store.
- **data** (*str* (http://docs.python.org/library/functions.html#str)) – The binary data to store.

`Data.`**`LoadObject`**`(`*item*`)`
Loads a Python object previously stored as a data item.

Parameters **item** (*str* (http://docs.python.org/library/functions.html#str)) – The name of the data item to load.

Returns The contents of the data item stored on disk.

Return type object

`Data.`**`SaveObject`**`(`*item*, *obj*`)`
Stores a Python object as a data item with the given name.

Parameters

- **item** (*str* (http://docs.python.org/library/functions.html#str)) – The name of the data item to store.
- **data** (*object* (http://docs.python.org/library/functions.html#object)) – The Python object to store.

`Data.`**`Exists`**`(`*item*`)`
Checks for the presence of a data item with the given name

Parameters **item** (*str* (http://docs.python.org/library/functions.html#str)) – The name of the data item to check for.

Returns The existence of the item.

Return type bool

`Data.`**`Remove`**`(`*item*`)`
Removes a previously stored data item with the given name.

Parameters **item** (*str* (http://docs.python.org/library/functions.html#str)) – The name of the data item to remove.

Dict

The Dict API provides access to a global singleton dictionary object, which can be used as a key-value store by the developer. The dictionary is automatically persisted across plug-in relaunches, and can safely be accessed from any thread.

Note: Only basic Python data types are officially supported by the dictionary. Many other types of object will be accepted correctly, but certain objects will cause errors (e.g. XML and HTML element objects). The developer should convert these objects to a more basic type before attempting to store them.

`x = Dict[key]`
Retrieves the value for the given key from the dictionary.

`Dict[key] = x`
Stores the contents of the variable in the dictionary using the given key.

`key in Dict`
Evaluates as `True` if the dictionary contains the given key.

`Dict.`**`Save`**`()`
Causes the dictionary to be saved to disk immediately. The framework will automatically save the file periodically, but the devloper can call this method to ensure recent changes are persisted.

`Dict.`**`Reset`**`()`
Removes all data from the dictionary.

6.1.8 Location Services

Locale

The Locale API provides methods for determining locale information about the current user (for example, their spoken language and geographical location).

`Locale.`**`Geolocation`**
Returns the user's country, obtained via IP-based geolocation, e.g. `US`.

`Locale.`**`CurrentLocale`**
Returns the locale of the user currently making a request to the plug-in, or *None* if no locale was provided.

`Locale.`**`DefaultLocale`**
Returns the default locale currently in use by the plug-in, e.g. `en-us`.

`Locale.`**`LocalString`**(*key*)
Retrieves the localized version of a string with the given key. Strings from the user's current locale are given highest priority, followed by strings from the default locale.

See *String files* for more information on proividing localized versions of strings.

`Locale.`**`LocalStringWithFormat`**(*key*, **args*)
Retrieves the localized version of a string with the given key, and formats it using the given arguments.

Locale.Language

Locale.Language provides constants representing every language known by the framework. These constants represent two-character ISO 639-1 language codes (http://en.wikipedia.org/wiki/List_of_ISO_639-1_codes).

classmethod `Language.`**`Match`**(*name*)
Attempt to match a given string to a language. Returns the unknown code (xx) if no match could be found.

The following languages are currently defined:

Language	Code
Unknown	xx
Afar	aa
Abkhazian	ab
Afrikaans	af
Akan	ak
Albanian	sq
Amharic	am
Arabic	ar
Aragonese	an
Armenian	hy
Assamese	as
Avaric	av
Avestan	ae
Aymara	ay
Azerbaijani	az
Bashkir	ba
Bambara	bm

Continued on next page

Table 6.1 – continued from previous page

Basque	eu
Belarusian	be
Bengali	bn
Bihari	bh
Bislama	bi
Bosnian	bs
Breton	br
Bulgarian	bg
Burmese	my
Catalan	ca
Chamorro	ch
Chechen	ce
Chinese	zh
ChurchSlavic	cu
Chuvash	cv
Cornish	kw
Corsican	co
Cree	cr
Czech	cs
Danish	da
Divehi	dv
Dutch	nl
Dzongkha	dz
English	en
Esperanto	eo
Estonian	et
Ewe	ee
Faroese	fo
Fijian	fj
Finnish	fi
French	fr
Frisian	fy
Fulah	ff
Georgian	ka
German	de
Gaelic	gd
Irish	ga
Galician	gl
Manx	gv
Greek	el
Guarani	gn
Gujarati	gu
Haitian	ht
Hausa	ha
Hebrew	he
Herero	hz

Continued on next page

Table 6.1 – continued from previous page

Hindi	hi
HiriMotu	ho
Croatian	hr
Hungarian	hu
Igbo	ig
Icelandic	is
Ido	io
SichuanYi	ii
Inuktitut	iu
Interlingue	ie
Interlingua	ia
Indonesian	id
Inupiaq	ik
Italian	it
Javanese	jv
Japanese	ja
Kalaallisut	kl
Kannada	kn
Kashmiri	ks
Kanuri	kr
Kazakh	kk
Khmer	km
Kikuyu	ki
Kinyarwanda	rw
Kirghiz	ky
Komi	kv
Kongo	kg
Korean	ko
Kuanyama	kj
Kurdish	ku
Lao	lo
Latin	la
Latvian	lv
Limburgan	li
Lingala	ln
Lithuanian	lt
Luxembourgish	lb
LubaKatanga	lu
Ganda	lg
Macedonian	mk
Marshallese	mh
Malayalam	ml
Maori	mi
Marathi	mr
Malay	ms
Malagasy	mg

Continued on next page

Table 6.1 – continued from previous page

Maltese	mt
Moldavian	mo
Mongolian	mn
Nauru	na
Navajo	nv
SouthNdebele	nr
NorthNdebele	nd
Ndonga	ng
Nepali	ne
NorwegianNynorsk	nn
NorwegianBokmal	nb
Norwegian	no
Chichewa	ny
Occitan	oc
Ojibwa	oj
Oriya	or
Oromo	om
Ossetian	os
Panjabi	pa
Persian	fa
Pali	pi
Polish	pl
Portuguese	pt
Pushto	ps
Quechua	qu
RaetoRomance	rm
Romanian	ro
Rundi	rn
Russian	ru
Sango	sg
Sanskrit	sa
Serbian	sr
Sinhalese	si
Slovak	sk
Slovenian	sl
Sami	se
Samoan	sm
Shona	sn
Sindhi	sd
Somali	so
Sotho	st
Spanish	es
Sardinian	sc
Swati	ss
Sundanese	su
Swahili	sw

Continued on next page

Table 6.1 – continued from previous page

Swedish	sv
Tahitian	ty
Tamil	ta
Tatar	tt
Telugu	te
Tajik	tg
Tagalog	tl
Thai	th
Tibetan	bo
Tigrinya	ti
Tonga	to
Tswana	tn
Tsonga	ts
Turkmen	tk
Turkish	tr
Twi	tw
Uighur	ug
Ukrainian	uk
Urdu	ur
Uzbek	uz
Venda	ve
Vietnamese	vi
Volapuk	vo
Welsh	cy
Walloon	wa
Wolof	wo
Xhosa	xh
Yiddish	yi
Yoruba	yo
Zhuang	za
Zulu	zu
Brazilian	pb
NoLanguage	xn

6.1.9 Reading user preferences

The framework provides a basic system for defining and retrieving user preferences. Allowing the user to modify and save preferences is handled automatically by the clients and the framework - the developer only needs to read them at the required points in their code.

Prefs

The Prefs API provides a simple method for retrieving a given preference.

`x = Prefs[id]`
Retrieves the value of the user preference with the given *id*.

Preference files

Preference files are used to define a set of preferences available for a plug-in. The files should contain a JSON-formatted array, composed of a dictionary for each preference. The following code demonstrates the available preference types:

```
[

        {
                "id":                   "pref1",
                "label":                "Text preference",
                "type":         "text",
                "default":      "Default Value",
        },

        {
                "id":                   "pref2",
                "label":                "Hidden text preference (passwords, etc.
                "type":         "text",
                "option":       "hidden",
                "default":      "",
                "secure":   "true",  # Indicates that the value should not be tr
        },

        {
                "id":                   "pref3",
                "label":                "Boolean preference",
                "type":         "bool",
                "default":      "true",
        },

        {
                "id":                   "pref4",
                "label":                "Enum preference",
                "type":         "enum",
                "values":       ["value1", "value2", "value3"],
                "default":      "value3",
        },
]
```

6.1.10 Utilities

Hash

The Hash API provides methods for calculating common hashes of binary data. All methods return a lowercase string containing a hexadecimal representation of the data's hash.

`Hash.`**`MD5`**`(`*data*`)`

`Hash.`**`SHA1`**`(`*data*`)`

`Hash.`**`SHA224`**`(`*data*`)`

`Hash.`**`SHA256`**`(`*data*`)`

`Hash.`**`SHA384`**`(`*data*`)`

`Hash.`**`SHA512`**`(`*data*`)`

`Hash.`**`CRC32`**`(`*data*`)`

String

The String API provides several methods for performing common utility functions on strings.

`String.`**`Encode`**`(`*s*`)`
: Encodes the given string using the framework's standard encoding (a slight variant on Base64 which ensures that the string can be safely used as part of a URL).

`String.`**`Decode`**`(`*s*`)`
: Decodes a string previously encoded using the above function.

`String.`**`Quote`**`(`*s*, *usePlus=False*`)`
: Replaces special characters in *s* using the `%xx` escape. Letters, digits, and the characters `'_.-'` are never quoted. If *usePlus* is `True`, spaces are escaped as a plus character instead of `%20`.

`String.`**`Unquote`**`(`*s*, *usePlus=False*`)`
: Replace `%xx` escapes by their single-character equivalent. If *usePlus* is `True`, plus characters are replaced with spaces.

`String.`**`StripTags`**`(`*s*`)`
: Removes HTML tags from a given string.

`String.`**`UUID`**`()`
: Generates a universally unique identifier (UUID) string. This string is guaranteed to be unique.

`String.`**`StripDiacritics`**`(`*s*`)`
: Removes diacritics from a given string.

`String.`**`Pluralize`**`(`*s*`)`
: Attempts to return a pluralized version of the given string (e.g. converts `boot` to `boots`).

`String.`**`LevenshteinDistance`**`(`*first*, *second*`)`
: Computes the Levenshtein distance (http://en.wikipedia.org/wiki/Levenshtein_distance) between two given strings.

`String.`**`LongestCommonSubstring`**`(`*first*, *second*`)`
: Returns the longest substring contained within both strings.

Datetime

The Datetime API provides a number of methods for manipulating Python *datetime* objects and converting them to and from other formats like timestamps or string representations.

`Datetime.`**`Now`**`()`
Returns the current date and time.

Return type

datetime (http://docs.python.org/library/datetime.html#datetime-objects)

`Datetime.`**`ParseDate`**`(`*date*`)`
Attempts to convert the given string into a datetime object.

Return type

datetime (http://docs.python.org/library/datetime.html#datetime-objects)

`Datetime.`**`Delta`**`(`***kwargs*`)`
Creates a *timedelta* object representing the duration given as keyword arguments. Valid argument names are *days*, *seconds*, *microseconds*, *milliseconds*, *minutes*, *hours* and *weeks*. This object can then be added to or subtracted from an existing *datetime* object.

Return type timedelta (http://docs.python.org/library/datetime.html#timedelta-objects)

`Datetime.`**`TimestampFromDatetime`**`(`*dt*`)`
Converts the given *datetime* object to a UNIX timestamp.

Return type int

`Datetime.`**`FromTimestamp`**`(`*ts*`)`
Converts the given UNIX timestamp to a *datetime* object.

Return type

datetime (http://docs.python.org/library/datetime.html#datetime-objects)

Util

The Util API provides other useful utility functions.

`Util.`**`Random`**`()`
Returns a random number between 0 and 1.

Return type float

`Util.`**`RandomInt`**`(`*a*, *b*`)`
Returns a random integer *N* such that `a <= N <= b`.

`Util.`**`RandomItemFromList`**`(`*l*`)`
Returns a random item selected from the given list.

6.1.11 Debugging

Log

The Log API allows the developer to add their own messages to the plug-in's log file. The different methods represent the different log levels available, in increasing levels of severity. The first argument to each method should be a format string, which is formatted using any additional arguments and keyword arguments provided.

Log.**Debug**(*fmt*, **args*, ***kwargs*)

Log.**Info**(*fmt*, **args*, ***kwargs*)

Log.**Warn**(*fmt*, **args*, ***kwargs*)

Log.**Error**(*fmt*, **args*, ***kwargs*)

Log.**Critical**(*fmt*, **args*, ***kwargs*)

Log.**Exception**(*fmt*, **args*, ***kwargs*)
The same as the *Critical* method above, but appends the current stack trace to the log message.

Note: This method should only be called when handling an exception.

6.1.12 Constants

Client Platforms

ClientPlatform.**MacOSX**

ClientPlatform.**Linux**

ClientPlatform.**Windows**

ClientPlatform.**iOS**

ClientPlatform.**Android**

ClientPlatform.**LGTV**

ClientPlatform.**Roku**

Protocols

Protocol.**Shoutcast**

Protocol.**WebKit**

Protocol.**HTTPMP4Video**

Protocol.**HTTPVideo**

Protocol.**RTMP**

Protocol.**HTTPLiveStreaming**

Protocol.**HTTPMP4Streaming**

View Types

ViewType.**Grid**

ViewType.**List**

Summary Text Types

SummaryType.**NoSummary**

SummaryType.**Short**

SummaryType.**Long**

Audio Codecs

AudioCodec.**AAC**

AudioCodec.**MP3**

Video Codecs

VideoCodec.**H264**

Container File Types

Container.**MP4**

Container.**MKV**

Container.**MOV**

Container.**AVI**

Object Container Content Types

ContainerContent.**Secondary**

ContainerContent.**Mixed**

ContainerContent.**Genres**

ContainerContent.**Playlists**

ContainerContent.**Albums**

ContainerContent.**Tracks**

ContainerContent.**GenericVideos**

ContainerContent.**Episodes**

ContainerContent.**Movies**

ContainerContent.**Seasons**

ContainerContent.**Shows**

ContainerContent.**Artists**

CHAPTER

SEVEN

SITE CONFIGURATION REFERENCE

7.1 Tags (Alphabetical)

7.1.1 <action>

```
<!ELEMENT action (click|goto|lockPlugin|move|pause|run|type|visit)*>
```

The <action> elemement is used inside a state and event element and tells the player "what do to" when it's corresponding condition is true.

eg.

```
<state name="playing">
  <event>
      <condition>
          <command name="pause"/>
      </condition>
      <action>
          <goto state="paused"/>
      </action>
  </event>
</state>
```

Here is a list of action tags:

- <*click*> (page 93)
- <*goto*> (page 96)
- <*lockPlugin*> (page 96)
- <*move*> (page 96)
- <*pause*> (page 98)
- <*run*> (page 99)
- <*type*> (page 101)
- <*visit*> (page 102)

7.1.2 <and>

```
<!ELEMENT and (and|color|command|condition|frameLoaded|javascript|not|or|pref|ti
```

This element is used inside of *<condition>* (page 93) tags to build out boolean logic.

Unlike the `<condition>` and `<not>` tags, the `<and>` tag may have more than one of any operator/conditional.

See also:

- *Event Conditions* (page 20)
- *<condition>* (page 93)

ex.

```
<condition name="foo">
  <and>
    <or>
        <color x="1" y="1" rgb="ffffff"/>
        <color x="2" y="2" rgb="ffffff"/>
    </or>
    <not>
      <color x="3" y="3" rgb="ffffff"/>
    </not>
    <color x="4" y="4" rgb="ffffff"/>
  </and>
</condition>
```

7.1.3 <bigStep>

```
<!ELEMENT bigStep EMPTY>
<!ATTLIST bigStep minus CDATA #REQUIRED>
<!ATTLIST bigStep plus CDATA #REQUIRED>
```

This is only used in *Javascript* (page 18) seekbars. It defines two javascript expressions to be executed.

One when big stepping forward and one backwards.

ex.

```
<bigStep plus="myPlayerObj.skipForward(10);" minus="myPlayerObj.skipBackward(10)
```

Both the `minus` and `plus` attributes may also contain a preference based replacement variable. For example, the following code would replace `${mystepsize}` with the value of the preference named `mystepsize`:

```
<bigStep plus="myPlayerObj.skipForward(${mystepsize});" minus="myPlayerObj.skipB
```

7.1.4 <condition>

```
<!ELEMENT condition (and|color|command|condition|frameLoaded|javascript|not|or|p
<!ATTLIST condition name CDATA #IMPLIED>
```

The condition tag can be a child of either the *<site>* (page 100) tag or the *<event>* (page 95) tag.

When directly under the *<site>* (page 100) tag it is acting as a named conditional block. See: *Named Condition Definitions* (page 21)

When under the *<event>* (page 95) tag it is acting as an inline definition. See: *Event Conditions* (page 20)

The `<condition>` tag may only have one (or none) of either an operator (and, or, not) or a condition tag (color, command, condition, frameLoaded, javascript, pref, title, url)

7.1.5 <color>

```
<!ELEMENT color EMPTY>
<!ATTLIST color x CDATA "0">
<!ATTLIST color y CDATA "0">
<!ATTLIST color rgb CDATA #IMPLIED>
<!ATTLIST color op (dimmer-than|brighter-than) #IMPLIED>
```

This condition allows you to test pixel colors underneath a specific x and y coordinate. X/Y coordinates are relative to the size of the player and not the entire web page. Negative (relative) coordinates are allowed.

The `rgb` attribute is in the hex RGB value. It should not include the '#' symbol nor should it be abbreviated as it can be in CSS.

eg:

```
rgb="FFFFFF"  - ok
rgb="#FFFFFF" - no
rgb="#FFF"    - no
rgb="FFF"     - no
```

If the `op` attribute is omitted, then the operation is equality. Otherwise the `rgb` value is used as reference for `brighter-than` or `dimmer-than`

7.1.6 <click>

```
<!ELEMENT click EMPTY>
<!ATTLIST click x CDATA #REQUIRED>
<!ATTLIST click y CDATA #REQUIRED>
<!ATTLIST click skipMouseUp (true|false) "false">
```

This condition allows you to click at a specific x and y coordinate. X/Y coordinates are relative to the size of the player and not the entire web page. Negative (relative) coordinates are allowed.

If you do not want a mouse up event to be fired, then set the `skipMouseUp` attribute to `true`.

7.1.7 <command>

```
<!ELEMENT command EMPTY>
<!ATTLIST command name CDATA #REQUIRED> <!-- pause|play|bigstep+|smallstep+|bigs
```

This condition can be used to test for Plex video player events. These occur in response to the keyboard, mouse or remote control.

Valid values for the `name` attribute are:

- pause
- play
- bigstep+
- bigstep-
- smallstep+
- smallstep-
- chapter+
- chapter-

7.1.8 <crop>

```
<!ELEMENT crop EMPTY>
<!ATTLIST crop x CDATA #REQUIRED>
<!ATTLIST crop y CDATA #REQUIRED>
<!ATTLIST crop width CDATA #REQUIRED>
<!ATTLIST crop height CDATA #REQUIRED>
```

This tag is used to crop the video player. Usually it's used to chop out blank space and or some player control so that the screen it zoomed in to focus soley on the video content.

If the player has extra data that you would like to crop out you can note the pixel width and height that you wish to crop from the top (y) and left (x). To crop from the right or bottom just reduce the `width` and `height` attributes.

If width and height are set to 0, then they are considered "auto" and will not crop the right or bottom.

7.1.9 <dead>

```
<!ELEMENT dead EMPTY>
<!ATTLIST dead x CDATA "0">
<!ATTLIST dead y CDATA "0">
```

Used in seekbars. This is the x/y coordinate where you would like the mouse to rest after clicking on a position in the seekbar. Sometimes a players controls need to fade back out and you need to put the mouse in a "safe" place.

7.1.10 <end>

```
<!ELEMENT end EMPTY>
<!ATTLIST end x CDATA "0">
<!ATTLIST end y CDATA "0">
<!ATTLIST end condition CDATA #IMPLIED>
```

For *Simple* (page 17) and *Thumb* (page 17) seekbars you should specify the x/y coordinates of the end of the playback timeline indicator.

For *Javascript* (page 18) seekbars, you should specify the `condition` attribute with a javascript expression evaluating to true if the player is at the "end"

In the case of *Javascript* (page 18) seekbars, the `condition` attribute may also contain a preference based replacement variable. For example, the following code would replace `${username}` with the value of the preference named `username`:

```
<end condition="'${username}' == 'david'">
```

See: *Seekbars* (page 16)

7.1.11 <event>

```
<!ELEMENT event (condition,action)>
```

A container who's parent is the *<state>* (page 100) and that contains a *<condition>* (page 93) and a *<action>* (page 91) tag.

See: *States and Events* (page 18)

7.1.12 <frameLoaded>

```
<!ELEMENT frameLoaded EMPTY>
```

A condition which evaluates as true when the webpage has been loaded.

7.1.13 <goto>

```
<!ELEMENT goto EMPTY>
<!ATTLIST goto state CDATA #REQUIRED>
<!ATTLIST goto param CDATA #IMPLIED>
```

An action which will transition to the state specified in the `state` attribute

If the `state` attribute is "end" it is handled specially and will exit the video and return to the previous navigation menu in Plex. Addiionally, if you provide some text to the `param` attribute when transitioning to the "end" state, then this message will be displayed in a message box to the user. Useful for error messages when exiting.

7.1.14 <javascript>

Changed in version Plex: 0.8.3 *<javascript>* (page 96) `script` attribute can accept javascript replacement

```
<!ELEMENT javascript EMPTY>
<!ATTLIST javascript script CDATA #REQUIRED>
<!ATTLIST javascript matches CDATA #REQUIRED>
```

This condition takes a javascript expression in the `script` attribute and an value to compare the expression to in the `matches` attribute.

Examples assuming `myVariable` is 10 :

```
<javascript script="myVariable" matches="10"/> <!-- true condition -->
<javascript script="myVariable == 10 ? 1 : 0" matches="1"/> <!-- true condition
<javascript script="myVariable == 99 ? 1 : 0" matches="1"/> <!-- false condition
```

7.1.15 <lockPlugin>

```
<!ELEMENT lockPlugin EMPTY>
```

This is only relevant if you have also set the `manualLock` attribue to `true` in the *<site>* (page 100) tag.

You should only use the combination of manualLock / lockPlugin if you have to perform actions/conditions prior to getting to the video itself. For example, logging in, changing the size of the player with javascript, or setting a cookie and redirecting to another url etc.

7.1.16 <move>

```
<!ELEMENT move EMPTY>
<!ATTLIST move x CDATA #REQUIRED>
<!ATTLIST move y CDATA #REQUIRED>
```

This condition allows you to move the mouse to a specific x and y coordinate. X/Y coordinates are relative to the size of the player and not the entire web page. Negative (relative) coordinates are allowed.

7.1.17 <not>

```
<!ELEMENT not (and|color|command|condition|frameLoaded|javascript|not|or|pref|ti
```

This element is used inside of *<condition>* (page 93) tags to build out boolean logic.

The <not> tag must have one and only one of either an operator (and, or, not) or a condition tag (color, command, condition, frameLoaded, javascript, pref, title, url)

See also:

- *Event Conditions* (page 20)
- *<condition>* (page 93)

ex.

```
<condition name="foo">
  <and>
    <or>
    <color x="1" y="1" rgb="ffffff"/>
    <color x="2" y="2" rgb="ffffff"/>
    </or>
    <not>
      <color x="3" y="3" rgb="ffffff"/>
    </not>
    <color x="4" y="4" rgb="ffffff"/>
  </and>
</condition>
```

7.1.18 <or>

```
<!ELEMENT or (and|color|command|condition|frameLoaded|javascript|not|or|pref|tit
```

This element is used inside of *<condition>* (page 93) tags to build out boolean logic.

Unlike the <condition> and <not> tags, the <or> tag may have more than one of any operator/conditional.

See also:

- *Event Conditions* (page 20)
- *<condition>* (page 93)

ex.

```
<condition name="foo">
<and>
 <or>
     <color x="1" y="1" rgb="ffffff"/>
     <color x="2" y="2" rgb="ffffff"/>
 </or>
 <not>
   <color x="3" y="3" rgb="ffffff"/>
 </not>
 <color x="4" y="4" rgb="ffffff"/>
</and>
</condition>
```

7.1.19 <pause>

```
<!ELEMENT pause EMPTY>
<!ATTLIST pause time CDATA "0"> <!-- number of milliseconds to wait -->
```

An action which will wait for the specified number of milliseconds.

ex.

```
<pause time="1000"/> <!-- wait one second -->
```

7.1.20 <percentComplete>

```
<!ELEMENT percentComplete EMPTY>
<!ATTLIST percentComplete equals CDATA #REQUIRED>
```

This is only used in *Javascript* (page 18) seekbars. It defines a single javascript expression which should evaluate to floating point number between 0.0 and 100.0.

ex.

```
<percentComplete equals="(myPlayerObj.position() / myPlayerObj.vidLength()) * 10
```

The `equals` attribute may also be a preference based replacement variable. For example, `${vidlength}` would be replaced by the value of the preference named `vidlength`

```
<percentComplete equals="(myPlayerObj.position() / ${vidlength}) * 100.0"/>
```

7.1.21 <played>

```
<!ELEMENT played (color|condition)*>
```

The color(s) representing the played portion of the Seekbar (for simple seekbars) or the color(s) that should match the Seekbar thumb (for thumb seekbars)

Only used in *Simple* (page 17) and *Thumb* (page 17) seekbars

7.1.22 <pre>

```
<!ELEMENT pre EMPTY>
<!ATTLIST pre m CDATA #REQUIRED>
```

TODO: what exactly is this and how should it be used? Prerequisites too like does it require manualLock=True etc.

7.1.23 <pref>

```
<!ELEMENT pref EMPTY>
<!ATTLIST pref name CDATA #REQUIRED>
<!ATTLIST pref exists (true|false) "false">
```

A condition to test for the presence or absense of a preference.

ex.

```
<pref name="username" exists="true"/>
```

7.1.24 <run>

```
<!ELEMENT run EMPTY>
<!ATTLIST run script CDATA #REQUIRED>
```

An action to run the specified javascript.

ex.

```
<run script="myVariable=10;"/>
```

The `script` attribute may also contain a preference based replacement variable. For example:

```
<run script="myUsername='${username}';"/>
```

7.1.25 <seekbar>

```
<!ELEMENT seekbar (start?|end?|played?|dead?|bigStep?|smallStep?|percentComplete
<!ATTLIST seekbar type (javascript|simple|thumb) #REQUIRED>
```

The container for seekbars. See *Seekbars* (page 16) for more information.

7.1.26 <setCookie>

Changed in version Plex: 0.8.3 *<setCookie>* (page 99) `value` attribute can accept javascript replacement

```
<!ELEMENT setCookie EMPTY>
<!ATTLIST setCookie domain CDATA #IMPLIED>
<!ATTLIST setCookie path CDATA #IMPLIED>
<!ATTLIST setCookie name CDATA #IMPLIED>
<!ATTLIST setCookie secure CDATA #IMPLIED>
<!ATTLIST setCookie value CDATA #IMPLIED>
```

Allows you to set a cookie.

ex.

```
<site>
  <setCookie domain=".videosite.com" path="/" name="tosAccepted" value="true" se
  <!-- ... -->
</site>
```

7.1.27 <site>

```
<!ELEMENT site (condition|crop|pre|seekbar|setCookie|state)*>
<!ATTLIST site agent CDATA #IMPLIED>
<!ATTLIST site allowPopups (true|false) "false">
<!ATTLIST site identifier CDATA #IMPLIED>
<!ATTLIST site initialState CDATA #REQUIRED>
<!ATTLIST site manualLock (true|false) "false">
<!ATTLIST site plugin CDATA #IMPLIED>
<!ATTLIST site randomAgent (true|false) "false">
<!ATTLIST site site CDATA #REQUIRED>
<!ATTLIST site version CDATA "1.0">
<!ATTLIST site windowHeight CDATA #IMPLIED>
<!ATTLIST site windowWidth CDATA #IMPLIED>
```

7.1.28 <start>

```
<!ELEMENT start EMPTY>
<!ATTLIST start x CDATA "0">
<!ATTLIST start y CDATA "0">
```

The x/y coordinates of the end of the playback timeline indicator.

Only used in *Simple* (page 17) and *Thumb* (page 17) seekbars

See: *Seekbars* (page 16)

7.1.29 <state>

```
<!ELEMENT state (event)*>
<!ATTLIST state name CDATA #REQUIRED>
```

A state contains one or more *<event>* (page 95) tags each of which contain a *<condition>* (page 93) and an *<action>* (page 91).

Typically, in an action a *<goto>* (page 96) tag is used with a name to transition to another state with the corresponding `state` attribute set.

See *States and Events* (page 18) for more information

7.1.30 <smallStep>

```
<!ELEMENT smallStep EMPTY>
<!ATTLIST smallStep minus CDATA #REQUIRED>
<!ATTLIST smallStep plus CDATA #REQUIRED>
```

This is only used in *Javascript* (page 18) seekbars. It defines two javascript expressions to be executed.

One when big stepping forward and one backwards.

ex.

```
<smallStep plus="myPlayerObj.skipForward(5);" minus="myPlayerObj.skipBackward(5)
```

Both the `minus` and `plus` attributes may also contain a preference based replacement variable. For example, the following code would replace `${mystepsize}` with the value of the preference named `mystepsize`:

```
<smallStep plus="myPlayerObj.skipForward(${mystepsize});" minus="myPlayerObj.ski
```

7.1.31 <title>

```
<!ELEMENT title EMPTY>
<!ATTLIST title matches CDATA #REQUIRED>
```

A condition which evaluates to true when the title of the webpage matches the regular expression in `matches`.

7.1.32 <type>

```
<!ELEMENT type EMPTY>
<!ATTLIST type text CDATA #IMPLIED>
<!ATTLIST type key CDATA #IMPLIED>
```

An action to either type a string or a single key code.

You may use `text` or `key` but not both together.

The `text` attribute may also be a preference based replacement variable. For example, the following code block would type out the characters that are contained in the value for the `username` preference :

```
<type text="${username}"/>
```

7.1.33 <url>

```
<!ELEMENT url EMPTY>
<!ATTLIST url matches CDATA #REQUIRED>
```

A condition which evaluates to true when the url of the webpage matches the regular expression in `matches`.

7.1.34 <visit>

```
<!ELEMENT visit EMPTY>
<!ATTLIST visit url CDATA #REQUIRED>
```

An action which causes the url specified in `url` to be visited.

7.2 XML DTD

Download the DTD

```
<!-- ROOT -->
<!ELEMENT site (condition|crop|pre|seekbar|setCookie|state)*>
<!ATTLIST site allowPopups (true|false) "false">
<!ATTLIST site identifier CDATA #IMPLIED>
<!ATTLIST site initialState CDATA #REQUIRED>
<!ATTLIST site manualLock (true|false) "false">
<!ATTLIST site plugin CDATA #IMPLIED>
<!ATTLIST site randomAgent (true|false) "false">
<!ATTLIST site site CDATA #REQUIRED>
<!ATTLIST site version CDATA "1.0">
<!ATTLIST site windowHeight CDATA #IMPLIED>
<!ATTLIST site windowWidth CDATA #IMPLIED>
<!ATTLIST site agent CDATA #IMPLIED>

<!-- CORE STRUCTURE -->

<!ELEMENT action (click|goto|lockPlugin|move|pause|run|type|visit)*>

<!ELEMENT condition (and|color|command|condition|frameLoaded|javascript|not|or|p
<!ATTLIST condition name CDATA #IMPLIED>

<!ELEMENT event (condition,action)>

<!ELEMENT state (event)*>
<!ATTLIST state name CDATA #REQUIRED>
```

```
<!-- SEEKBAR / PLAYER SIZE -->

<!ELEMENT seekbar (start?|end?|played?|dead?|bigStep?|smallStep?|percentComplete
<!ATTLIST seekbar type (javascript|simple|thumb) #REQUIRED>

<!ELEMENT start EMPTY>
<!ATTLIST start x CDATA "0">
<!ATTLIST start y CDATA "0">

<!ELEMENT end EMPTY>
<!ATTLIST end x CDATA "0">
<!ATTLIST end y CDATA "0">
<!ATTLIST end condition CDATA #IMPLIED>

<!ELEMENT played (color|condition)*>

<!ELEMENT dead EMPTY>
<!ATTLIST dead x CDATA "0">
<!ATTLIST dead y CDATA "0">

<!-- javascript seekbar stuff -->
<!ELEMENT bigStep EMPTY>
<!ATTLIST bigStep minus CDATA #REQUIRED>
<!ATTLIST bigStep plus CDATA #REQUIRED>

<!ELEMENT smallStep EMPTY>
<!ATTLIST smallStep minus CDATA #REQUIRED>
<!ATTLIST smallStep plus CDATA #REQUIRED>

<!ELEMENT percentComplete EMPTY>
<!ATTLIST percentComplete equals CDATA #REQUIRED>

<!-- crop the player (flash or silverlight) to obtain the "playable area" -->
<!ELEMENT crop EMPTY>
<!ATTLIST crop x CDATA #REQUIRED>
<!ATTLIST crop y CDATA #REQUIRED>
<!ATTLIST crop width CDATA #REQUIRED>
<!ATTLIST crop height CDATA #REQUIRED>

<!-- CONDITIONS -->

<!ELEMENT and (and|color|command|condition|frameLoaded|javascript|not|or|pref|ti
<!ELEMENT or (and|color|command|condition|frameLoaded|javascript|not|or|pref|tit
<!ELEMENT not (and|color|command|condition|frameLoaded|javascript|not|or|pref|ti

<!ELEMENT color EMPTY>
<!ATTLIST color x CDATA "0">
<!ATTLIST color y CDATA "0">
<!ATTLIST color rgb CDATA #IMPLIED>
<!ATTLIST color op (dimmer-than|brighter-than) #IMPLIED>
```

```
<!ELEMENT command EMPTY>
<!ATTLIST command name CDATA #REQUIRED> <!-- pause|play|bigstep+|smallstep+|bigs

<!ELEMENT frameLoaded EMPTY>

<!ELEMENT javascript EMPTY>
<!ATTLIST javascript script CDATA #REQUIRED>
<!ATTLIST javascript matches CDATA #REQUIRED>

<!ELEMENT pref EMPTY>
<!ATTLIST pref name CDATA #REQUIRED>
<!ATTLIST pref exists (true|false) "false">

<!ELEMENT title EMPTY>
<!ATTLIST title matches CDATA #REQUIRED>

<!ELEMENT url EMPTY>
<!ATTLIST url matches CDATA #REQUIRED>

<!-- ACTIONS -->

<!ELEMENT click EMPTY>
<!ATTLIST click x CDATA #REQUIRED>
<!ATTLIST click y CDATA #REQUIRED>
<!ATTLIST click skipMouseUp (true|false) "false">

<!ELEMENT goto EMPTY>
<!ATTLIST goto state CDATA #REQUIRED>
<!ATTLIST goto param CDATA #IMPLIED>

<!ELEMENT lockPlugin EMPTY>

<!ELEMENT move EMPTY>
<!ATTLIST move x CDATA #REQUIRED>
<!ATTLIST move y CDATA #REQUIRED>

<!ELEMENT pause EMPTY>
<!ATTLIST pause time CDATA "0">

<!ELEMENT run EMPTY>
<!ATTLIST run script CDATA #REQUIRED>

<!ELEMENT type EMPTY>
<!ATTLIST type text CDATA #IMPLIED>
<!ATTLIST type key CDATA #IMPLIED>


<!ELEMENT visit EMPTY>
<!ATTLIST visit url CDATA #REQUIRED>
```

```
<!-- MISC -->

<!ELEMENT pre EMPTY>
<!ATTLIST pre m CDATA #REQUIRED>

<!ELEMENT setCookie EMPTY>
<!ATTLIST setCookie domain CDATA #IMPLIED>
<!ATTLIST setCookie path CDATA #IMPLIED>
<!ATTLIST setCookie name CDATA #IMPLIED>
<!ATTLIST setCookie secure CDATA #IMPLIED>
<!ATTLIST setCookie value CDATA #IMPLIED>
```

7.3 Keyboard Keycode Lookup

You can find the key code for use with the `<type key="??"/>` tag.

Use the *Dec* column from http://asciitable.com

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y