Introduction

The Plex Media API provides an easy and powerful way for clients to browse and manage media libraries, as well as rich transcode functionality for video, images, and audio. The API is roughly RESTful in nature, and the server can return XML or JSON responses.

Core Concepts

One of the most important concepts of the API is that it allows “browsability”. This means that you can start at the “root” endpoint of a Plex Media Server, and walk the full tree of media by following some simple rules.

Most responses from the Media Server come in the form of a Media Container element. Let’s look at (a slightly simplified version of) the root XML:

```
<MediaContainer size="7" friendlyName="Retina Media Server"
                machineIdentifier="d4ed404613461edf6412ec38391c962bef783a4f"
                mediaAnalysisVersion="1" myPlex="1" platform="MacOSX"
                platformVersion="10.8.2" requestParametersInCookie="1"
                transcoderActiveVideoSessions="0" transcoderAudio="1"
                transcoderVideoBitrates="64,96,208,320,720,1500,2000,3000,4000,8000,
                10000,12000,20000"
                transcoderVideoQualities="0,1,2,3,4,5,6,7,8,9,10,11,12"
                transcoderVideoResolutions="128,128,160,240,320,480,768,720,720,1080,
                1080,1080,1080" updatedAt="1349080834" version="0.9.7.0-27d5f5f"
                webkit="1">
  <Directory count="1" key="channels" title="channels" />
  <Directory count="1" key="library" title="library" />
  <Directory count="2" key="music" title="music" />
  <Directory count="2" key="photos" title="photos" />
  <Directory count="1" key="video" title="video" />
</MediaContainer>
```

Media Containers have top-level information such as number of elements (*size*), and other pertinent general information. In this case (the root level), it contains information about the server, its capabilities (e.g. has an audio transcoder, supports WebKit), platform, and version. The Directory elements indicate that they can be browsed into, via the *key* attribute. That attribute is very similar to the *src* attribute in an HTML hyperlink element; it can be relative (e.g. “music”), absolute (e.g. “/library/sections/1”) or full (e.g. “http://.......”. If you follow these rules for key construction, you can browse your way through the entire media tree with very few exceptions.

If you'd like to request responses in JSON, you can do so by passing an "*Accept: application/json*" HTTP header. It should be noted that channels at this point do not yet support JSON responses.

Finally, there are times when responses can be quite large. The server allows paging for any response, by passing two HTTP headers: *X-Plex-Container-Size* and *X-Plex-Container-Start*. These parameters, as with any other starting with X-Plex, maybe also be passed as special querystring parameters at the end of a regular query. For example, to return only the first library section in JSON:

```
$ curl -H "Accept: application/json" "http://localhost:32400/library/sections?X-Plex-Container-
                        Start=0&X-Plex-Container-Size=1"
```

Media Structure

Media (from a music track to a movie to a photo) is returned in a consistent and flexible form, so it's important to understand the structure of media elements. Generally speaking, each element is composed of the following hierarchy:

- Metadata element (e.g. Video, Photo, Track) with attributes describing the high level metadata for the element (e.g. A movie's release date, plot summary, and title).
- Each Metadata element contains child Media elements. These elements describe alternate media for the metadata item. For example, a movie might have a 1080p version and a 720p version. A channel may advertise an MP4 stream and an RTMP stream. Media elements contain a summary of the media (e.g. resolution and bitrate).
- Each Media element contains child Part elements. This allows for media which has been split into multiple parts, such as some videos which are split to fit onto multiple CDs. The part contains details on how to retrieve the part (stored in the key, of course), size of the part, the format (container) of the part, and so forth.
- Each Part element can contain child Stream elements. Generally speaking, streams are present when details is requested for a single item, for performance reasons. Each Stream element contains detailed information about the video, audio, or subtitle codec. Photos don't have streams.

Let's look at an example, which should serve to clarify the explanation above:

```
<Video ratingKey="178" key="/library/metadata/178"
                        guid="com.plexapp.agents.imdb://tt0945571?lang=en" studio="Pixar"
                        type="movie" title="Lifted" contentRating="G" summary="When an
                        overconfident teen alien..." rating="9.6000003814697301" year="2006"
                        tagline="Failure is an option." thumb="/library/metadata/178/thumb?
                        t=1348566474" art="/library/metadata/178/art?t=1348566474"
                        duration="240000" originallyAvailableAt="2006-10-14"
                        addedAt="1348566458" updatedAt="1348566474">
  <Media id="178" duration="302304" bitrate="4648" width="1280" height="688"
                        aspectRatio="1.85" audioChannels="6" audioCodec="ac3"
                        videoCodec="h264" videoResolution="720" container="mkv"
                        videoFrameRate="24p">
    <Part id="178" key="/library/parts/178/file.mkv" duration="302304"
                        file="/Users/elan/Desktop/Pixar Short Film
                        Collection/lifted.2006.720p.bluray.x264-sinners.mkv" size="175655686"
                        container="mkv">
      <Stream id="1" streamType="1" codec="h264" index="0" bitrate="4000"
                        language="English" languageCode="eng" cabac="1" duration="302304"
                        frameRate="23.976" height="688" level="51" profile="high"
                        refFrames="8" width="1280" />
      <Stream id="2" streamType="2" selected="1" codec="ac3" index="1" channels="6"
                        bitrate="640" bitrateMode="cbr" dialogNorm="-27" duration="302304"
                        samplingRate="48000" />
    </Part>
  </Media>
  <Genre id="41" tag="Comedy" />
  <Genre id="42" tag="Computer Animation" />
  <Director id="48" tag="Gary Rydstrom" />
  <Country id="47" tag="USA" />
</Video>
```

In this example, we see a single video item (further specified to be of type *movie*) with a single media item, and a single part (the simplest example). The part has two streams (one audio, one video).

Browsing the Library

Without further ado, let’s learn how to browse the library. The Plex media library is split into section, each of which can store movies (or video, more generally), TV shows, photos, or music. Each of these types has a slightly different structure; TV shows and music have a grandparent/parent/child structure (show ⇢ season ⇢ episode for TV shows and artist ⇢ album ⇢ track) for music.

The root of the library is, not surprisingly, “/library”, which lists the sections. We’re starting to see some common attributes here, including *title* and *type* (further specifying the subtype of an element).

```
<MediaContainer size="9" allowSync="0" identifier="com.plexapp.plugins.library"
                        mediaTagPrefix="/system/bundle/media/flags/"
                        mediaTagVersion="1342927275" title1="Plex Library">
  <Directory art="/:/resources/movie-fanart.jpg" refreshing="0" key="660" type="movie"
                        title="Documentaries" agent="com.plexapp.agents.imdb" scanner="Plex
                        Movie Scanner" language="en"
                        uuid="c75ed6d4663b478d22815a28be7f1cbd2009adbc"
                        updatedAt="1348907427" createdAt="1300660202">
    <Location path="/Volumes/Drobo/Documentaries/Movies" />
  </Directory>
  <Directory art="/:/resources/movie-fanart.jpg" refreshing="0" key="1" type="movie"
                       title="Movies" agent="com.plexapp.agents.imdb" scanner="Plex Movie
                       Scanner" language="en"
                       uuid="6feb25d9ccd5eb0cb126b3c50725c8533e85561d"
                       updatedAt="1348907452" createdAt="1272532302">
    <Location path="/Volumes/Drobo/Movies" />
  </Directory>
</MediaContainer>
```

From here, we just use the regular key-construction rules to build the path into "/library/sections/660", and we can keep going all the way through the library.

One thing to note: When you get to a "leaf" element (e.g. a movie, or an episode), the key for that element leads you to what seems to be a media container with only the element in it. This is the "details" view, which includes full information (streams, full cast, genres, etc.) which is not present in lists of elements.

The Plex XML is full of rich graphical "adornments" for media. For example, *thumb* points to a poster or thumbnail for an item, *art* points to background art for an item, *banner* points to banner art. The *theme* attribute specifies the URL for theme music to be played with an item (e.g. a TV show).

In addition, there are media "flags" available for some items. The media container has two attributes: *mediaTagPrefix* and *mediaTagVersion*. These are combined, along with attributes from the Media element (*aspectRatio*, *audioChannels*, *audioCodec*, *videoCodec*, *videoResolution*, *container*, *studio,* and *videoFrameRate*) to form a URL like this:
http://127.0.0.1:32400/system/bundle/media/flags/studio/Pixar%20Animation%20Studios?t=1342927275

The uniform structure of the data across different types of media, and the richness (tags! banners! genres! background art!) are two of the more powerful and far-reaching design choices we've made in the Plex API and underlying library.

Global Library Operations

There are two endpoints which aggregate media across all library sections. These are the global equivalents to the per-section endpoints.

```
GET /library/onDeck
GET /library/recentlyAdded?[unwatched=1]
```

Library Management

Browsing a library is all well and fine, but we need to be able to manage it too. Let's take a tour of the API available around management.

We can create a section with a single request. Sections are associated with a primary metadata agent, which is the entity responsible for imbuing the media with rich metadata, usually from an online source. Want to know which agents are available? Request "/system/agents". Scanners are responsible for looking on disk for media and pre-processing them before injecting into the library. Want to know which scanners are available for movie sections (movie type = 1)? Request "/system/scanners/1".

```
POST /library/sections?
                    type=movie&agent=com.plexapp.agents.imdb&scanner=Plex+Movie+Sca
                    nner&language=en&location=%2Ftmp%2Felan&name=Ze+Test
```

Deleting a section is easy as well, where X is the key of the section. Want to just modify a section? You can use the RESTful PUT verb on the same path.

```
PUT /library/sections/X
DELETE /library/sections/X
```

You can refresh all sections, or just a single section. Refreshing a section runs the scanners to see what has changed in a section. You can pass a few arguments to these endpoints: *turbo* (0/1) specifies whether to run a quick "turbo" scan, not venturing into directories that it appears haven't changed. This generally works well, but you can run into trouble with some NAS drives and networked filesystems that don't update modification times appropriately. The *deep* (0/1) parameter forces a full scan, and the *force* (0/1) forces a refresh and reload of all metadata (which involves going out to the web and can take a long time.

Finally, you can force the server to perform media analysis (usually performed at import time) on an entire section.

```
GET /library/sections/all/refresh
GET /library/sections/X/refresh
PUT /library/sections/X/analyze
```

There are also some maintenance tasks (hey, every library needs a custodian!) as well. The optimize endpoint ensures the database is as fast as possible, and the clean endpoint deletes unused metadata and media bundles from the disk (those bundles are written to by the agents, and then sucked into the database by the media server). Last, but not least, when a scan finds that an item has gone missing, it doesn't delete the associated metadata yet; rather, it marks it as deleted (setting the *deletedAt* attribute). This is done in case the user has painstakingly customized metadata for an item, and it's only temporarily missing. Emptying trash for a section removes all these items.

```
PUT /library/optimize
PUT /library/clean/bundles
PUT /library/sections/X/emptyTrash
```

Next up, there are quite a few endpoints dealing with a single item. These generally hang off the “details” endpoint (“/library/metadata/X”. To kick things off, there are single-item versions of a few endpoints we’ve seen above:

```
PUT /library/metadata/X/refresh
PUT /library/metadata/X/analyze
```

The *force* parameter is not supported when refreshing a single item.

The agents almost always download multiple graphical assets (posters, art, etc.) for an item, and it’s easy to get the full list of assets available. These endpoints return a media container list of assets (note that *thumbs* returns just the thumbnails for a file generated via media analysis). The last endpoint allows setting a newly selected one.

```
GET /library/metadata/X/thumbs
GET /library/metadata/X/posters
GET /library/metadata/X/arts
GET /library/metadata/X/themes
PUT /library/metadata/X/poster (or art, or theme) ?url=X
```

As intelligent (sentient, some would say) as the metadata agents are, sometimes they get a match wrong. Not to worry, there are a few endpoints around matching. The *matches* endpoint returns all the potential matches, and has a number of optional arguments you can use to customize the results (e.g. specifying another *title*, a *year*, or a totally different *agent*. The *manual* parameter specifies that this is a user-initiated action, and ensures that the agent works extra hard to find matches.

When the library encounters two pieces of media that look like the same actual thing (e.g. “300.avi” and “300.mkv”), it merges them together (as seen above, we’ll now have two Media elements under the same Video). If we want to split them, use the *split* endpoint. Alternatively, if you’d like to merge items into a target item, use the *merge* endpoint.

```
GET /library/metadata/X/matches?
                    [title=X]&[agent=Y]&[language=Z]&[year=A]&[manual=0/1]
PUT /library/metadata/X/match?guid=X&title=Y
PUT /library/metadata/X/unmatch
PUT /library/metadata/X/split
PUT /library/metadata/X/merge?ids=Y,Z
```

Finally, we might want to edit some fields in the actual item itself, or set a new poster or art, for example. Each of the fields that can be modified can also be locked. This implies that the data is set in stone, and will not be overwritten by new agent-provided data. Let’s look at a fairly complicated example which shows off the power of this endpoint. Here we're setting and locking the title, setting a genre (and locking it) and setting the cast (with a single actor).

```
PUT /library/metadata/X?
        title=Foo&title.locked=1&genre[0].tag.tag=Action&genre[0].tagging.index
        =1&genre.locked=1&actor[0].tag.tag=Anna+Kim&actor[0].tagging.text=T
        he+Wife
```

Media Streams

Video media parts can have multiple streams of type video, audio or subtitle. There may be multiple audio and subtitle streams, and exactly one stream must be selected of each type. By default, the media server automatically picks streams based on the language and subtitle preferences.

There is an endpoint to set the selected streams for each part:

```
PUT /library/parts/81551?subtitleStreamID=X&audioStreamID=Y
```

Passing a subtitle stream of 0 means

Advanced Filtering

You can get a filtered view of any section by using the following endpoint:

```
GET /library/sections/X/all?<filters>
```

The filters are expressed via querystring parameters. Each parameter contains a field, operator and value(s). Valid operators are =, >=, <=, and !=. For tag-based fields (e.g. genre), only = and != operators are allowed, and correspond to "are tagged with" and "are not tagged with".

- year=2010 ⇢ returns all media from the year 2010
- year>=1990 ⇢ returns all media from years 1990 and later.
- year!=2012 ⇢ returns all media from years other than 2012.
- year=2012,2013,2015 ⇢ returns all media from years 2012, 2013, and 2015.

When multiple parameters are passed, they are logically ANDed together.

- year=2010&rating>=5 ⇢ all media from the year 2012 with a 5 or greater.

Alphanumeric Pagination

You can get a list of alphanumeric characters to display by using the following endpoint:

```
GET /library/sections/X/firstCharacter
```

You can then use the provided keys to return media items for that character.

```
GET /library/sections/X/firstCharacter/Y
```

Miscellaneous Endpoints

When connecting to a remote server, you might need to know what token to send it; if you don’t know the machine identifier of the remote server, that’s hard, so there is an endpoint you can hit which returns basic information about the server, and isn’t protected.

```
GET /identity
```

```
<?xml version="1.0" encoding="UTF-8"?>
<MediaContainer size="0" machineIdentifier="567-456-457" version="0.9.7.3-36679df" />
```

Browsing Channels

blah

Advanced Channel Concepts

indirect and postURL

The Transcoders

It’s possible to obtain a list of the current transcode sessions:

```
GET /transcode/sessions
```

Given a session key, you can kill a transcode session with the following endpoint:

```
DELETE /transcode/sessions/<key>
```

Progress and Scrobbles

blah

Universal Search

…

At the end of the `/search` result XML, you'll see this:

```
<Provider key="/system/search" title="Local Network" type="mixed"/>
<Provider key="/search?type=10" title="Tracks" type="tracks"/>
<Provider key="/search/actor" title="Actors" type="actors"/>
```

These providers are "vectors" to additional search results. In the case of the tracks provider, it's split out in order to make the initial search as quick as possible.

The `/system/search` provider is much more interesting, as it returns providers for other servers on the network, as well as any channel search services.

Usually, clients issues the requests to the provider vectors in parallel, and then merge results as they come in.

Remote Control

Using the remote control API, a client can control players and command them to play media remotely. The first step is obtaining a list of players that are available:

```
GET /clients
```

This returns a list of players currently available:

```
<MediaContainer size="2">
  <Server name="Amazon Galaxy Nexus" host="10.0.0.129" address="10.0.0.129"
                          port="3000" machineIdentifier="10c90376-9a49-4b5e-a188-
                          5f034dacfa2d" version="2.5.0.1"/>
  <Server name="Mac-mini" host="10.0.0.9" address="10.0.0.9" port="3000"
                          machineIdentifier="2d421881-576f-478e-a355-4a4903f6d385"
                          version="0.9.5.4-f067f15"/>
</MediaContainer>
```

Once selected, commands are sent to the remote player by vectoring them through the media server. This ensures that any differences in player API (for AirPlay players, for example, or DLNA renderers) is abstracted by the interface between the client and the media server.

In order to play a piece of media remotely, use the following endpoint:

```
GET /system/players/<host>/application/playMedia
```

The parameters are as follows:

- **key**: The key attribute for the item (e.g. “/library/metadata/509283” for a library item, or “/system/services/url/lookup?url=XXXX”).
- **path**: The full URL of the container where the media item was (e.g. “http://x.x.x.x:32400/library/sections/1046/all” or “http://x.x.x.x:32400/video/vimeo/....”). The player will retrieve this container, and then match the key passed in to an item in the container. (Picture, for example, an album being played, starting at the specified track.)
- **viewOffset**: (optional) Milliseconds to offset the playback (used when resuming).
- **userAgent**: (optional) The User Agent to use when requesting the media.
- **httpCookies**: (optional) The cookies to use when requesting the media.

The following endpoints are available for player control:

```
GET /system/players/<host>/playback/stop
GET /system/players/<host>/playback/pause
GET /system/players/<host>/playback/stop
GET /system/players/<host>/playback/play
GET /system/players/<host>/playback/skipPrevious
GET /system/players/<host>/playback/skipNext
GET /system/players/<host>/playback/bigStepBack
GET /system/players/<host>/playback/bigStepForward
GET /system/players/<host>/playback/stepForward
GET /system/players/<host>/playback/stepBack
GET /system/players/<host>/application/setVolume?level=<0-100>
```

There is no facility currently for obtaining current status of the player, but this is expected to change in the near future.

Sync

Since syncing may involve transcoding or downloading (for channel content), sync provides an endpoint which provides visibility into the queue of media it’s processing.

```
GET /sync/transcodeQueue
```

This returns a list of transcode jobs, with a rich set of attributes for clients to use:

```
<TranscodeJob type="transcode" key="ddffc99252acaa993dea354d7da86c9f3aab5af6"
            progress="11" speed="25" remaining="15" size="19310334" title="Mater
            and the Ghostlight (2006)" thumb="/library/metadata/95237/thumb?
            t=1348518563" syncItemId="504" syncItemTitle="Mater and the
            Ghostlight"/>
```

The following attributes are returned:

- **type**: either “download” or “transcode”.
- **key**: the transcode session ID.
- **progress**: from 0-100, represents the percentage completion of the job.
- **speed**: represents the speed of the operation, as a factor of real time (only for transcodes).
- **remaining**: a rough estimate, in seconds, of the time remaining.