EvoStream

Your Media. Delivered Anywhere.

EvoStream Media Server
Application Programming Interface

Table of Contents

TABLE OF CONTENTS **2**

DEFINITION OF TERMS **4**

OVERVIEW **5**

Accessing the Runtime API 5

ASCII *5*

HTTP *6*

PHP and JavaScript *6*

JSON *6*

EVOSTREAM MEDIA SERVER API **7**

Streams 7

pullStream *7*

pushStream *10*

createHLSStream *12*

record *14*

listStreamsIds *15*

getStreamInfo *15*

listStreams *16*

getStreamsCount *18*

shutdownStream *18*

listPullPushConfig *21*

removePullPushConfig *22*

addStreamAlias *23*

listStreamAliases *23*

removeStreamAlias *24*

flushStreamAliases *24*

Connections 25

listConnectionsIds *25*

getConnectionInfo *25*

listConnections *26*

getExtendedConnectionCounters *27*

resetMaxFdCounters *27*

resetTotalFdCounters *27*

getConnectionsCount *28*

getConnectionsCountLimit *28*

setConnectionsCountLimit *28*

getBandwidth *29*

SetBandwidthLimit *30*

Services 31

listServices *31*

createService *32*

enableService *33*

shutdownService *33*

Miscellaneous API Functions 35
help *35*
setAuthentication *35*
setLogLevel *36*
version *36*
quit *36*
shutdownServer *37*

Definition of Terms

EMS	EvoStream Media Server
HTTP	Hyper-Text Transfer Protocol. The basic protocol used for web-page loading and web browsing. Also used for tunneling by many protocols. TCP based.
IDR	Instantaneous Decoding Refresh – This is a specific packet in the H.264 video encoding specification. It is a full snapshot of the video at a specific instance (one full frame). Video players require an IDR frame to start playing any video. “Frames” that occur between IDR Frames are simply offsets/differences from the first IDR.
JSON	JavaScript Object Notation
Lua	A lightweight multi-paradigm programming language
RTCP	Real Time Control Protocol – An protocol that is typically used with RTSP to synchronize two RTP streams, often audio and video streams
RTMP	Real Time Messaging Protocol – Used with Adobe Flash players
RTMPT	Real Time Messaging Protocol Tunneled – Essentially RTMP over HTTP
RTP	Real-time Transport Protocol – A simple protocol used to stream data, typically audio or video data.
RTSP	Real Time Streaming Protocol – Used with Android devices and live streaming clients like VLC or Quicktime. RTSP does not actually transport the audio/video data, it is simply a negotiation protocol. It is normally paired with a protocol like RTP, which will handle the actual data transport.
swfURL	Used in the RTMP protocol, this field is used to designate the URL/address of the Adobe Flash Applet being used to generate the stream (if any).
tcURL	Used in the RTMP protocol, this field is used to designate the URL/address of the originating stream server.
TOS	Type of Service. This is a field in IPv4 packets used by routers to determine how traffic should be dispersed, usually for prioritizing packets.
TTL	Time To Live. This is a field in Ipv4 packets used by routers to determine how many gateways/routers the packet should be able to pass through.
URI	Universal Resource Identifier. The generic form of a URL. URI’s are used to specify the location and type of streams.
URL	Uniform Resource Locator. This is a specific form of the URI used for web browsing (http://ip/page).
VOD	Video On Demand

Overview

This document describes the Application Programming Interface (API) presented by the EvoStream Media Server (EMS). The API provides the ability to manipulate the server at runtime. The server can be told to retrieve or create new streams, return information on streams and connections, or even start or stop functional services. The EvoStream Media Server API allows users to tightly integrate with the server without having to write native plugins or modules.

Accessing the Runtime API

The EvoStream Media Server (EMS) API can be accessed in two ways. The first is through an ASCII telnet interface. The second is by using HTTP requests. The API is identical for both methods of access.

The API functions parameters are NOT case sensitive.

ASCII

The ASCII interface is often the first interface used by users. It can be accessed easily through the telnet application (available on all operating systems) or through common scripting languages.

To access the API via the telnet interface, a telnet application will need to be launched on the same computer that the EMS is running on. The command to open telnet from a command prompt should look something like the following:

```
telnet localhost 1112
```

If you are on Windows 7 you may need to enable telnet. To do this, go to the Control Panel -> Programs -> Turn Windows Features on and off. Turn the telnet program on.

Please also note that on Windows, the default telnet behavior will need to be changed. You will need to turn local echo and new line mode on for proper behavior. Once you have entered telnet, exit the telnet session by typing **“ctrl+]”.** Then enter the following commands:

```
set localecho
set crlf
```

Press Enter/Return again to return to the Windows telnet session.

Once the telnet session is established, you can type out commands that will be immediately executed on the server.

An example of a command request/response from a telnet session would be the following:

Request:

```
version
```

Response:

```
{"data":"1.5","description":"Version","status":"SUCCESS"}
```

HTTP

To access the API via the HTTP interface, you simply need to make an HTTP request on the server with the command you wish to execute. By default, the port used for these HTTP requests is **7777**. The HTTP interface port can be changed in the main configuration file used by the EMS (typically config.lua).

All of the API functions are available via HTTP, but the request must be formatted slightly differently. To make an API call over HTTP, you must use the following general format:

```
http://IP:7777/functionName?params=base64(firstParam=XXX secondParam=YYY …)
```

In example, to call pullStream on an EMS running locally you would first need to base64 encode your parameters:

```
Base64(uri=rtmp://IP/live/myStream localstreamname=testStream) results in:
dXJpPXJ0bXA6Ly9JUC9saXZlL215U3RyZWFtIGxvY2Fsc3RyZWFtbmFtZT10ZXN0U3RyZWFt
```

```
http://192.168.5.5:7777/pullstream?params=
dXJpPXJ0bXA6Ly9JUC9saXZlL215U3RyZWFtIGxvY2Fsc3RyZWFtbmFtZT10ZXN0U3RyZWFt
```

PHP and JavaScript

PHP and JavaScript functions are also provided. These functions simply wrap the HTTP interface calls. They can be found in the *runtime_api* directory.

JSON

The EMS API provides return responses from most of the API functions. These responses are formatted in JSON so that they can be easily parsed and used by third party systems and applications. These responses will be identical, regardless of whether you are using the ASCII or HTTP interface.

When using the ASCII interface, it may be necessary to use a JSON interpreter so that responses can be more human-readable. A good JSON interpreter can be found at: http://chris.photobooks.com/json/default.htm or at http://json.parser.online.fr/.

EvoStream Media Server API

The EMS API can be broken down into a few groups of functionality. The first group, and the one most often used, is Stream Manipulation. The other groups are Connection Details and Services which are discussed later in the document.

Each API function is listed along with its mandatory and optional parameters. Examples of each interface can be found after the description of function parameters.

PLEASE NOTE:
All Boolean parameters are set using 1 for true and 0 for false!
Default values of parameters shown in parentheses or italicized are just remarks.

Streams

Streams are considered to be the actual video and/or audio feeds that are coming from, or going to, the EMS. Streams can be sent over a wide variety of protocols. The following functions are provided to manipulate and query Streams:

pullStream

This will try to pull in a stream from an external source. Once a stream has been successfully pulled it is assigned a "local stream name" which can be used to access the stream from the EMS.

This function has the following parameters:

Parameter Name	**Mandatory**	**Default Value**	**Description**
uri	true	*(null)*	The URI of the external stream. Can be RTMP, RTSP or unicast/multicast (d)mpegts.
keepAlive	false	1 *(true)*	If keepAlive is set to 1, the server will attempt to reestablish connection with a stream source after a connection has been lost. The reconnect will be attempted once every second.
localStreamName	false	*(computed)*	If provided, the stream will be given this name. Otherwise, a fallback technique is used to determine the stream name (based on the URI)
width	false	0	If specified (non-zero), this value will replace the detected value for the pixel width of the video.
height	false	0	If specified (non-zero), this value will replace the detected value for the pixel height of the video

Parameter Name	Mandatory	Default Value	Description
forceTcp	false	1 *(true)*	If 1 and if the stream is RTSP, a TCP connection will be forced. Otherwise the transport mechanism will be negotiated (UDP or TCP).
tcUrl	false	*(zero-length string)*	When specified, this value will be used to set the TC URL in the initial RTMP connect invoke
pageUrl	false	*(zero-length string)*	When specified, this value will be used to set the originating web page address in the initial RTMP connect invoke.
swfUrl	false	*(zero-length string)*	When specified, this value will be used to set the originating swf URL in the initial RTMP connect invoke
ttl	false	operating system supplied	Sets the IP_TTL (time to live) option on the socket
tos	false	operating system supplied	Sets the IP_TOS (Type of Service) option on the socket
rtcpDetectionInterval	false	10	How much time (in seconds) should the server wait for RTCP packets before declaring the RTSP stream as a RTCP-less stream
emulateUserAgent	false	*(EvoStream message)*	When specified, this value will be used as the user agent string. It is meaningful only for RTMP.

The EMS provides several shorthand User Agent strings (not case-sensitive) for convenience:

emulateUserAgent=FMLE	Resolves as "FMLE/3.0 (compatible; FMSc/1.0)"
emulateUserAgent=wirecast	Resolves as "Wirecast/FM 1.0 (compatible; FMSc/1.0)"
emulateUserAgent=evo	Resolves as "EvoStream Media Server (www.evostream.com) player"
emulateUserAgent=flash	Resolves as "MAC 11,3,300,265"

An example of the pullStream interface is:

```
pullStream uri=rtsp://AddressOfStream keepAlive=1 localStreamname=livetest
```

Then, to access that stream via a flash player, the following URI can be used:

```
rtmp://AddressOfEMS/live/livetest
```

The JSON response for pullStream contains the following details:

- data – The data to parse.
    - configID – The configuration ID for this command
    - emulateUserAgent – This is the string that the EMS uses to identify itself with the other server. It can be modified so that EMS identifies itself as, say, a Flash Media Server.
    - forceTcp – Whether TCP MUST be used, or if UDP can be used.
    - height – An optional description of the video stream's pixel height.
    - keepAlive – If true, the stream will attempt to reconnect if the connection is severed.
    - localStreamName – The local name for the stream.
    - pageUrl – A link to the page that originated the request (often unused).
    - rtcpDetectionInterval – Used for RTSP. This is the time period the EMS waits to determine if an RTCP connection is available for the RTSP/RTP stream. (RTSP is used for synchronization between audio and video).
    - swfUrl – The location of the Flash Client that is generating the stream (if any).
    - tcUrl – An RTMP parameter that is essentially a copy of the URI.
    - tos – Type of Service network flag.
    - ttl – Time To Live network flag.
    - uri – Contains key/value pairs describing the source stream's URI.
        - document – The document name of the source stream.
        - documentPath – The document path of the source stream.
        - documentWithFullParameters – The document name with parameters of the source stream.
        - fullParameters – The parameters for the source stream's URI.
        - fullUri – The full URI of the source stream.
        - fullUriWithAuth – The full URI with authentication of the source stream.
        - host – Name of the source stream's host.
        - ip – IP address of the source stream's host.
        - originalUri – The source stream's URI where it was generated.
        - parameters – Parameters for the source stream's URI (if any).
        - password – Password for authenticating the source stream (if required).
        - port – Port used by the source stream.
        - portSpecified – True if the port for the source stream is specified.
        - scheme – The protocol used by the source stream.
        - userName – The user name for authenticating the source stream (if required).
    - width – An optional description of the video stream's pixel width.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: pullStream

pushStream

This will try to push a local stream to an external destination. The pushed stream can only use the RTMP, RTSP or MPEG-TS unicast/multicast protocol.

This function has the following parameters:

Parameter Name	**Mandatory**	**Default Value**	**Description**
uri	true	*(null)*	The URI of the destination point (without stream name).
localStreamName	true	*(computed)*	If provided, the stream will be given this name. Otherwise, a fallback technique is used to determine the stream name (based on the URI).
tos	false	operating system supplied	Sets the IP_TOS (Type of Service) option on the socket.
keepAlive	false	1 *(true)*	If keepAlive is set to 1, the server will attempt to reestablish connection with a stream source after a connection has been lost. The reconnect will be attempted once every second.
targetStreamName	false	*(null)*	The name of the stream at destination. If not provided, the target stream name will be the same as the local stream name.
targetStreamType	false	live	It can be one of following: live, record, append. It is meaningful only for RTMP.
emulateUserAgent	false	*(EvoStream message)*	When specified, this value will be used as the user agent string. It is meaningful only for RTMP.
swfUrl	false	*(zero-length string)*	When specified, this value will be used to set the originating swf URL in the initial RTMP connect invoke
pageUrl	false	*(zero-length string)*	When specified, this value will be used to set the originating web page address in the initial RTMP connect invoke.
tcUrl	false	*(zero-length string)*	When specified, this value will be used to set the TC URL in the initial RTMP connect invoke
ttl	false	operating system supplied	Sets the IP_TTL (Time To Live) option on the socket.

For the EmulateUserAgent parameter, the EMS provides several shorthand User Agent strings (not case-sensitive) for convenience:

emulateUserAgent=FMLE	Resolves as “FMLE/3.0 (compatible; FMSc/1.0)”
emulateUserAgent=wirecast	Resolves as “Wirecast/FM 1.0 (compatible; FMSc/1.0)”
emulateUserAgent=evo	Resolves as “EvoStream Media Server (www.evostream.com)”
emulateUserAgent=flash	Resolves as “MAC 11,3,300,265”

An example of the pullStream interface is:

```
pushStream uri=rtmp://DestinationAddress keepAlive=1 localStreamname=pushtest
```

The JSON response contains the following details:

- data – The data to parse.
  - configID – The configuration ID for this command
  - emulateUserAgent – This is the string that the EMS uses to identify itself with the other server. It can be modified so that EMS identifies itself as, say, a Flash Media Server.
  - forceTcp – Whether TCP MUST be used, or if UDP can be used.
  - keepAlive – If true, the stream will attempt to reconnect if the connection is severed.
  - localStreamName – The local name for the stream.
  - pageUrl – A link to the page that originated the request (often unused).
  - swfUrl – The location of the Flash Client that is generating the stream (if any).
  - targetStreamName – The name of the stream at destination.
  - targetStreamType – One of following: \`live\`, \`record\`, \`append\`. Useful only for RTMP.
  - targetUri – Contains key/value pairs describing the destination stream’s URI.
    - document – The document name of the destination stream.
    - documentPath – The document path of the destination stream.
    - documentWithFullParameters – The document name with parameters of the destination stream.
    - fullDocumentPath – The document path of the destination stream.
    - fullDocumentPathWithParameters – The document path with parameters of the destination stream.
    - fullParameters – The parameters for the destination stream’s URI.
    - fullUri – The full URI of the destination stream.
    - fullUriWithAuth – The full URI with authentication of the destination stream.
    - host – The name of the destination stream’s host.
    - ip – The IP address of the destination stream’s host.
    - originalUri – The destination stream’s URI where it was generated.
    - parameters – Parameters for the destination stream’s URI.
    - password – Password for authenticating the destination stream (if required).
    - port – Port used by the destination stream.
    - portSpecified – True if the port for the destination stream is specified.
    - scheme – The protocol used by the destination stream.
    - userName – The user name for authenticating the destination stream (if required).
  - tcUrl – An RTMP parameter that is essentially a copy of the URI.
  - tos – Type of Service network flag.
  - ttl – Time To Live network flag.
- description – Describes the result of parsing/executing the command.

- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: pushStream

createHLSStream

Create an HTTP Live Stream (HLS) out of an existing H.264/AAC stream. HLS is used to stream live feeds to iOS devices such as iPhones and iPads.

This function has the following parameters:

Parameter Name	**Mandatory**	**Default Value**	**Description**
localStreamNames	true	*(null)*	The stream(s) that will be used as the input. This is a comma-delimited list of active stream names (local stream names).
targetFolder	true	*(null)*	The folder where all the *.ts/*.m3u8 files will be stored. This folder must be accessible by the HLS clients. It is usually in the web-root of the server.
keepAlive	false	1 *(true)*	If true, the EMS will attempt to reconnect to the stream source if the connection is severed.
overwriteDestination	false	1 *(true)*	If true, it will force overwrite of destination files.
staleRetentionCount	false	*(if not specified, it will have the value of playlistLength)*	The number of old files kept besides the ones listed in the current version of the playlist. Only applicable for rolling playlists.
createMasterPlaylist	false	1 *(true)*	If true, a master playlist will be created.
cleanupDestination	false	*(null)*	If 1 (true), all *.ts and *.m3u8 files in the target folder will be removed before HLS creation is started.
bandwidths	false	*(null)*	The corresponding bandwidths for each stream listed in localStreamNames. Again, this can be a comma-delimited list.
groupName	false	*(it will be a random name in the form of hls_group_xxxx)*	The name assigned to the HLS stream or group. If the localStreamNames parameter contains only one entry and groupName is not specified,

			groupName will have the value of the input stream name.
playlistType	false	appending	Either `appending` or `rolling`.
playlistLength	false	10	The length (number of elements) of the playlist. Used only when playlistType is `rolling`. Ignored otherwise.
playlistName	false	playlist.m3u8	The file name of the playlist (*.m3u8).
chunkLength	false	10	The length (in seconds) of each playlist element (*.ts file). If 0, chunking is made on IDR boundary.
chunkBaseName	false	segment	The base name used to generate the *.ts chunks.
chunkOnIDR	false	0 *(false)*	If true, chunking is performed ONLY on IDR. Otherwise, chunking is performed whenever chunk length is achieved.

An example of the createHLSStream interface is:

```
createHLSStream localstreamnames=hlstest bandwidths=128 targetfolder=/MyWebRoot/
groupname=hls playlisttype=rolling playlistLength=10 chunkLength=5
```

The corresponding link to use on an iOS device to pull this stream would then be:

```
http://My_IP_or_Domain/hls/playlist.m3u8
```

In other words: http://<my_web_server>/<HLS_group_name>/<playlist_file_name>

The JSON response contains the following details:

- data – The data to parse.
    - bandwidths – An array of integers specifying the bandwidths used for streaming.
    - chunkBaseName – The base name or prefix used for naming the output HLS chunks.
    - chunkLength – The length (in seconds) of each playlist element (*.ts file). If 0, chunking is made on IDR boundary.
    - chunkOnIdr – If true, chunking was made on IDR boundary.
    - cleanupDestination – If true, HLS files at the target folder were deleted before HLS creation began.
    - createMasterPlaylist – If true, a master playlist is created.
    - groupName – The name of the target folder where HLS files will be created.
    - keepAlive – If true, the stream will attempt to reconnect if the connection is severed.
    - localStreamNames – An array of local names for the streams.
    - overwriteDestination – If true, forced overwrite was enabled during HLS creation.

- playlistLength – The number of elements in the playlist. Useful only for `rolling` playlistType.
- playlistName – The file name of the playlist (*.m3u8).
- playlistType – Either `appending` or `rolling`.
- staleRetentionCount – The number of old files kept besides the ones listed in the current version of the playlist. Only applicable for rolling playlists.
- targetFolder – The folder where all the *.ts/*.m3u8 files are stored.

- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: createHLSStream

record

Records an RTMP, RTSP, or MPEG-TS stream. The record command allows users to record a stream that may not yet exist. When a new stream is brought into the server, it is checked against a list of streams to be recorded. Streams can be recorded in two ways, as FLV files and as MPEG-TS files. ONLY RTMP STREAMS CAN BE RECORDED AS FLV FILES!

Parameter Name	**Mandatory**	**Default Value**	**Description**
localStreamName	true	*(null)*	The name of the stream to be used as input for recording.
pathToFile	true	*(null)*	Specify path and file name to write to.
type	false	flv *(or)* ts	The type of file that the recorded stream will have: `flv` if RTMP or RTSP, or `ts` if MPEG-TS. **Only RTMP streams can be recorded to FLV files**
overwrite	false	0 *(false)*	If false, when a file already exists for the stream name, a new file will be created with the next appropriate number appended. If 1 (true), files with the same name will be overwritten.
keepAlive	false	1 *(true)*	If 1 (true), the server will restart recording every time the stream becomes available again.

An example of the record interface is:

```
record localStreamName=Video1 pathtofile=/recording/path type=flv overwrite=1
```

This records the local stream named Video1 to directory /recording/path in FLV format with overwrite enabled. The JSON response contains the following details about recording a stream:

- data – The data to parse.
  - keepAlive – If true, the stream will attempt to reconnect if the connection is severed.
  - localStreamName – The local name for the stream.
  - overwrite – If true, files with the same name will be overwritten.
  - pathToFile – Path to the folder where recorded files will be written.
  - type – Type of file for recording. Either `flv` (for RTMP/RTSP) or `ts` (for MPEG-TS).

- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: record

listStreamsIds

Get a list of IDs for every active stream.

This function has no parameters.

A JSON message will be returned containing the IDs of the active streams:

- data – Contains an array of IDs (integers) for the active streams.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: listStreamsIds

getStreamInfo

Returns a detailed set of information about a stream

Parameter Name	**Mandatory**	**Default Value**	**Description**
id	true	*(null)*	The uniqueId of the stream. Usually a value returned by listStreamsIDs

The JSON response contains the following details about a given stream:

- data – The data to parse.
    - audio – stats about the audio portion of the stream.
        - bytesCount - Total amount of audio data received.
        - droppedPacketsCount – The number of lost audio packets.
        - packetsCount – Total number of audio packets received.
    - bandwidth – The current bandwidth utilization of the stream.
    - creationTimestamp – The UNIX timestamp for when the stream was created. UNIX time is expressed as the number of seconds since the UNIX Epoch (Jan 1, 1970).
    - name – the "localStreamName" for this stream.
    - outStreamsUniqueIDs – *For pulled streams*. An array of the "out" stream IDs associated with this "in" stream.
    - pullSettings/pushSettings – Not present for streams requested by a 3rd party (IE player/client). A copy of the parameters used in the **pullStream** or **pushStream** command.
        - configId – The identifier for the pullPushConfig.xml entry.
        - emulateUserAgent – The string that the EMS uses to identify itself with the other server. It can be modified so that EMS identifies itself as, say, a Flash Media Server.
        - forceTcp – Whether TCP MUST be used, or if UDP can be used.
        - height – An optional description of the video stream's pixel height.

        - isHds – True if this is an HDS stream.
        - isHls – True if this is an HLS stream.
        - isRecord – True if this stream is actively recording.
        - keepAlive – If true, the stream will try to reconnect if the connection is severed.
        - localStreamName – Same as the above “name” field.
        - pageUrl – A link to the page that originated the request (often unused).
        - rtcpDetectionInterval – Used for RTSP. This is the time period the EMS waits to determine if an RTCP connection is available for the RTSP/RTP stream. (RTSP is used for synchronization between audio and video).
        - swfUrl – The location of the Flash Client that is generating the stream (if any).
        - tcUrl – An RTMP parameter that is essentially a copy of the URI.
        - tos – Type of Service network flag.
        - ttl – Time To Live network flag.
        - uri – The parsed values of the source streams URI.
        - width – An optional description of the video stream’s pixel width.
    - queryTimestamp – The time (in UNIX seconds) when the information in this request was populated.
    - type – The type of stream this is. The first two characters are of most interest:
        - char 1 = I for inbound, O for outbound.
        - char 2 = N for network, F for file.
        - char 3+ = further details about stream.
        - example: INR = Inbound Network Stream (a stream coming from the network into the EMS).
    - uniqueId – The unique ID of the stream (integer).
    - uptime – The time in seconds that the stream has been alive/running for.
    - video – Stats about the video portion of the stream.
        - bytesCount - Total amount of video data received.
        - droppedBytesCount – The number of video bytes lost.
        - droppedPacketsCount – The number of lost video packets.
        - packetsCount – Total number of video packets received.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: getStreamInfo

listStreams

Provides a detailed description of all active streams.

This interface does not have any parameters.

The JSON response contains the following details about each stream:

- data – The data to parse.
    - audio – stats about the audio portion of the stream.
        - bytesCount – Total amount of audio data received.
        - droppedBytesCount – The number of audio bytes lost.
        - droppedPacketsCount – The number of lost audio packets.
        - packetsCount – Total number of audio packets received.
    - bandwidth – The current bandwidth utilization of the stream.

    - canDropFrames – *Outstreams only*. Flag set by client allowing for dropped frames/packets.
    - creationTimestamp – The UNIX timestamp for when the stream was created. UNIX time is expressed as the number of seconds since the UNIX Epoch (Jan 1, 1970).
    - edgePid – Internal flag used for clustering.
    - inStreamUniqueID – *For pushed streams.* The id of the source stream.
    - name – the "localstreamname" for this stream.
    - outStreamsUniqueIDs – *For pulled streams.* An array of the "out" stream IDs associated with this "in" stream.
    - pullSettings/pushSettings – *Not present for streams requested by a 3rd party (IE player/client).* A copy of the parameters used in the **pullStream** or **pushStream** command.
        - configId – The identifier for the pullPushConfig.xml entry.
        - emulateUserAgent – The string that the EMS uses to identify itself with the other server. It can be modified so that EMS identifies itself as, say, a Flash Media Server.
        - forceTcp – Whether TCP MUST be used, or if UDP can be used.
        - height – An optional description of the video stream's pixel height.
        - isHds – True if this is an HDS stream.
        - isHls – True if this is an HLS stream.
        - isRecord – True if this stream is actively recording.
        - keepAlive – If true, the stream will try to reconnect if the connection is severed.
        - localStreamName – Same as the above "name" field.
        - pageUrl – A link to the page that originated the request (often unused).
        - rtcpDetectionInterval – Used for RTSP. This is the time period the EMS waits to determine if an RTCP connection is available for the RTSP/RTP stream. (RTSP is used for synchronization between audio and video).
        - swfUrl – The location of the Flash Client that is generating the stream (if any).
        - tcUrl – An RTMP parameter that is essentially a copy of the URI.
        - tos – Type of Service network flag.
        - ttl – Time To Live network flag.
        - uri – The parsed values of the source streams URI.
    - width – An optional description of the video stream's pixel width.
    - queryTimestamp – The time (in UNIX seconds) when the information in this request was populated.
    - type – The type of stream this is. The first two characters are of most interest:
        - char 1 = I for inbound, O for outbound.
        - char 2 = N for network, F for file.
        - char 3+ = further details about stream.
        - example: INR = Inbound Network Stream (a stream coming from the network into the EMS).
    - uniqueId – The unique ID of the stream (integer).
    - uptime – The time in seconds that the stream has been alive/running for.
    - video – Stats about the video portion of the stream.
        - bytesCount – Total amount of video data received.
        - droppedBytesCount – The number of video bytes lost.
        - droppedPacketsCount – The number of lost video packets.
        - packetsCount – Total number of video packets received.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: listStreams

getStreamsCount

Returns the number of active streams.

This function has no parameters.

A JSON message will be returned giving the number of active streams:

- data – The data to parse.
    - `Count` – The number of active streams.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: getStreamsCount

shutdownStream

Terminates a specific stream. When permanently=1 is used, this command is analogous to `removePullPushConfig`

This function has the following parameters:

Parameter Name	**Mandatory**	**Default Value**	**Description**
id	false	*0*	The uniqueId of the stream that needs to be terminated. The stream ID's can be obtained using the listStreams command
localStreamName	false	*"" (zero length String)*	The name of the inbound stream which you wish to terminate.
permanently	false	1 *(true)*	If true, the corresponding push/pull configuration will also be terminated. Therefore, the stream will NOT be reconnected when the server restarts.

An example of the shutdownStream interface is:

```
shutdownstream id=55 permanently=1
```

This will shut down the stream with id of 55 and remove its push/pull configuration.

The JSON response contains the following details about the stream being shut down:

- data – The data to parse.
    - protocolStackInfo – Contains key/value pairs describing the protocol stack used by the stream.
        - carrier – Details about the connection itself.
            - farIP – The IP address of the distant party.
            - farPort – The port used by the distant party.
            - nearIP – The IP address used by the local computer.

    - nearPort – The port used by the local computer.
    - rx – Total bytes received on this connection.
    - tx – Total bytes transferred on this connection.
    - type – The connection type (TCP, UDP) .
  - stack[1] – Describes the farthest protocol primitive.
    - applicationID – the ID of the internal application using the connection.
    - creationTimestamp – The time (in UNIX seconds) when the application started using the connection.
    - id – The unique ID for this stack relation.
    - isEnqueueForDelete – Internal flag used for cleanup.
    - queryTimestamp – The time (in UNIX seconds) when this data was populated.
    - type – A descriptor for how the application is using the connection.
  - stack[2] – Describes the next protocol primitive.
    - applicationId – the ID of the internal application using the connection.
    - creationTimestamp – The time (in UNIX seconds) when the application started using the connection.
    - id – The unique ID for this stack relation.
    - isEnqueueForDelete – Scheduled for deletion.
    - queryTimestamp – The time (in UNIX seconds) when this data was populated.
    - rxInvokes – Number of received RTMP function invokes.
    - streams[1]
      - audio – Stats about the audio portion of the stream.
        - bytesCount – Total amount of audio data received.
        - droppedBytesCount – The number of audio bytes lost.
        - droppedPacketsCount – The number of lost audio packets.
        - packetsCount – Total number of audio packets received.
      - bandwidth – The current bandwidth utilization of the stream.
      - canDropFrames – *Outstreams only*. Flag set by client allowing for dropped frames/packets.
      - creationTimestamp – The time (in UNIX secs) when the stream was created.
      - inStreamUniqueId – *For pushed streams.* The id of the source stream.
      - name – the “localstreamname” for this stream.
      - queryTimestamp – The time (in UNIX secs) when this data was populated.
      - type – The type of stream this is. See **getStreamInfo** for details.
      - uniqueId – The unique ID of the stream (integer).
      - upTime – The time in seconds that the stream has been alive/running for.
      - video
        - bytesCount – Total amount of video data received.
        - droppedBytesCount – The number of video bytes lost.
        - droppedPacketsCount – The number of lost video packets.
        - packetsCount – Total number of video packets received.
    - streams[2]
      - bandwidth – The current bandwidth utilization of the stream.
      - creationTimestamp – The time (in UNIX secs) when the stream was created.
      - name – the “localstreamname” for this stream.
      - outStreamsUniqueIDs – *For pulled streams.* An array of the “out” stream IDs associated with this “in” stream.
      - queryTimestamp – The time (in UNIX secs) when this data was populated.

            - type – The type of stream this is. See **getStreamInfo** for details.
            - uniqueId – The unique ID of the stream (integer).
            - uptime – The time in seconds that the stream has been alive/running for.
        - txInvokes – Number of sent RTMP function invokes.
        - type – A descriptor for how the application is using the connection.
    - streamInfo
        - bandwidth – The current bandwidth utilization of the stream.
        - creationTimestamp – The time (in UNIX seconds) when the stream was created.
        - name – the "localstreamname" for this stream.
        - outStreamsUniqueIds – *For pulled streams.* An array of the "out" stream IDs associated with this "in" stream.
        - queryTimestamp – The time (in UNIX seconds) when this data was populated.
        - type – The type of stream this is. See **getStreamInfo** for details.
        - uniqueId – The unique ID of the stream (integer).
        - upTime – The time in seconds that the stream has been alive/running for.
- description – Describes the result of parsing/executing the command.
- status – \`SUCCESS\` if the command was parsed and executed successfully, \`FAIL\` if not.

A typical response in parsed JSON format is shown here: shutdownStream

listPullPushConfig

Returns a list with all push/pull configurations.

Whenever the pullStream or pushStream interfaces are called, a record containing the details of the pull or push is created in the pullpushconfig.xml file. Then, the next time the EMS is started, the pullpushconfig.xml file is read, and the EMS attempts to reconnect all of the previous pulled or pushed streams.

This interface has no parameters.

The JSON response contains the following details about the pull/push configuration:

- data – The data to parse.
  - hds (not supported in v1.5)
  - hls (see fields of **createHLSStream** command)
  - pull (see fields of **pullStream** command)
  - push (see fields of **pushStream** command)
  - record (see fields of **record** command)
    - status (within the stream types shown above) – array of current and previous states
      - current/previous
        - code – An integer representing the state of the stream.
        - description – Describes the state of the stream.
        - timestamp – The time (in Unix secs) the state was updated.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: listPullPushConfig

removePullPushConfig

This command will both stop the stream and remove the corresponding configuration entry. This command is the same as performing: `shutdownStream permanently=1`

This function has the following parameters:

Parameter Name	Mandatory	Default Value	Description
id	true*	*(null)*	The configId of the configuration that needs to be removed. ConfigId's can be obtained from the listPullPushConfig interface. *Mandatory only if the HlsHdsGroup parameter is not specified.
HlsHdsGroup	true*	*(null)*	The name of HLS group that needs to be removed. *Mandatory only if the id parameter is not specified.
removeHlsHdsFiles	false	0 *(false)*	If 1 (true) and the stream is HLS, the folder associated with it will be removed.

An example of the removePullPushConfig interface is:

```
removePullPushConfig id=555
```

The JSON response contains the following details about the pull/push configuration:

- data – The data to parse.
  - configId – The identifier for the pullPushConfig.xml entry.
  - isHds – True if this is an HDS stream.
  - isHls – True if this is an HLS stream.
  - isRecord – True if this is a stream that is being recorded.
  - Other fields present are dependent on stream type.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: removePullPushConfig

addStreamAlias

Allows you to create secondary name(s) for internal streams. Once an alias is created the localstreamname cannot be used to request playback of that stream. Once an alias is used (requested by a client) the alias is removed. Aliases are designed to be used to protect/hide your source streams.

This function has the following parameters:

Parameter Name	**Mandatory**	**Default Value**	**Description**
localStreamName	true	*(null)*	The original stream name.
aliasName	true	*(null)*	The alias alternative to the localStreamName.

An example of the addStreamAlias interface is:

```
addStreamAlias localStreamName=bunny aliasName=video1
```

The JSON response contains the following details:

- data – The data to parse.
    - aliasName – The alias alternative to the localStreamName.
    - localStreamName – The original stream name.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: addStreamAlias

listStreamAliases

Returns a complete list of aliases.

This function has no parameters.

The JSON response contains the following details.

- data – Contains an array of pairs of aliasName and localStreamName.
    - aliasName – The alias alternative to the localStreamName.
    - localStreamName – The original stream name.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: listStreamAliases

removeStreamAlias

Removes an alias of a stream.

This function has the following parameters:

Parameter Name	Mandatory	Default Value	Description
aliasName	true	*(null)*	The alias to delete

An example of the removeStreamAlias interface is:

```
removeStreamAlias aliasName=video1
```

The JSON response contains the following details.

- data – The data to parse.
    - aliasName – The alias of the stream that was removed.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: removeStreamAlias

flushStreamAliases

Invalidates all streams aliases.

This function has no parameters.

The JSON response contains the following details.

- data – Nothing to parse for this command.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: flushStreamAliases

Connections

The Connections API functions allow the user to manipulate and query the actual network connections between the EMS and other systems or applications. The most common connections will occur between the EMS and a media player. However, there are a variety of other situations where connections can occur, such as (but not limited to) connections between two EMS instances, or an EMS and another server.

listConnectionsIds

Returns a list containing the IDs of every active connection

This interface has no parameters.

The JSON response contains the following details:

- data – An array of connection IDs.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: listConnectionsIds

getConnectionInfo

Returns a detailed set of information about a connection

Parameter Name	**Mandatory**	**Default Value**	**Description**
id	true	*(null)*	The uniqueId of the connection. Usually a value returned by listConnectionsIds

An example of the getConnectionInfo interface is:

```
getConnectionInfo id=5
```

This gets connection info about a connection with id of 5.

The JSON response contains the following details about one connection:

- data – The data to parse. See the **listConnections** command for a description of the fields.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: getConnectionInfo

listConnections

Returns details about every active connection

Parameter Name	Mandatory	Default Value	Description
excludeNonNetworkProtocols	false	1 *(true)*	If 1 (true), all non-networking protocols will be excluded. If 0 (false), non-networking protocols will be included.

An example of the listConnections interface is:

```
listConnections excludeNonNetworkProtocols=0
```

This lists connections including non-networking protocols.

The JSON response contains the following details about each connection:

- data – The data to parse.
  - carrier – Details about the connection itself.
    - farIP – The IP address of the distant party.
    - farPort – The port used by the distant party.
    - nearIP – The IP address used by the local computer.
    - nearPort – The port used by the local computer.
    - rx – Total bytes received on this connection.
    - tx – Total bytes transferred on this connection.
    - type – The connection type (TCP, UDP) .
  - pushSettings/pullSettings/hlsSettings/recordSettings – A copy of the parameters used in the stream command that caused this connection to be made.
    - configId – The identifier for the pullPushConfig.xml entry.
    - isHds – True if this is an HDS stream.
    - isHls – True if this is an HLS stream.
    - isRecord – True if this is a stream that is being recorded.
    - Other fields present depend on the stream type (see **pushStream**, **pullStream**, **createHLSStream**, **record** commands).
  - stack – details about what internal resources are using the connection..
    - applicationID – the ID of the internal application using the connection.
    - creationTimestamp – The time (in UNIX seconds) when the application started using the connection.
    - id – The unique ID for this stack relation.
    - isEnqueueForDelete – Internal flag used for cleanup.
    - queryTimestamp – The time (in UNIX seconds) when this data was populated.
    - rxInvokes – Number of received RTMP function invokes.
    - streams – Details about the streams that are using the connection (see fields in ListStreams).
    - txInvokes – Number of sent RTMP function invokes.
    - type – A descriptor for how the application is using the connection.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: listConnections

getExtendedConnectionCounters

Returns a detailed description of the network descriptors counters. This includes historical high-water-marks for different connection types and cumulative totals.

This interface has no parameters.

The JSON response contains the following details:

- data – The data to parse.
    - origin
        - grandTotal – Stats for all connections.
        - managedNonTcpUdp – Stats for non-TCP/UDP connections.
        - managedTcp – Stats for TCP connections.
        - managedTcpAcceptors – Stats for TCP acceptors.
        - managedTcpConnectors – Stats for TCP connectors.
        - managedUdp – Stats for UDP connections.
        - rawUdp – Stats for raw UDP.
    - Total – Summary.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: getExtendedConnectionCounters

resetMaxFdCounters

Reset the maximum, or high-water-mark, from the Connection Counters

This interface has no parameters

The JSON response contains the following details:

- data – Nothing to parse for this command.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: resetMaxFdCounters

resetTotalFdCounters

Reset the cumulative totals from the Connection Counters

This interface has no parameters

The JSON response contains the following details:

- data – Nothing to parse for this command.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: resetTotalFdCounters

getConnectionsCount

Returns the number of active connections

This interface has no parameters

The JSON response contains the following details:

- data – The data to parse.
    - count – The number of active connections.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: getConnectionsCount

getConnectionsCountLimit

Returns the limit of concurrent connections. This is the maximum number of connections an EMS instance will allow at one time.

This interface has no parameters.

The JSON response contains the following details:

- data – The data to parse.
    - current – The current number of concurrent connections.
    - limit – The maximum number of concurrent connections.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: getConnectionsCountLimit

setConnectionsCountLimit

This interface sets a limit on the number of concurrent connections the EMS will allow.

This function has the following parameters:

Parameter Name	**Mandatory**	**Default Value**	**Description**
count	true	*(null)*	The maximum number of connections allowed on this instance at one time. CLI connections are not affected.

An example of the setConnectionsCountLimit interface is:

```
setConnectionsCountLimit count=500
```

This sets the connection limit to 500.

The JSON response contains the following details:

- data – The data to be parsed.
    - current – The current bandwidths.
        - in – The inbound bandwidth.
        - out – The outbound bandwidth.
    - max – The maximum bandwidths.
        - in – The inbound limit.
        - out – The outbound limit.
- description – Describes the result of parsing/executing the command.
- status – \`SUCCESS\` if the command was parsed and executed successfully, \`FAIL\` if not.

A typical response in parsed JSON format is shown here: setConnectionsCountLimit

getBandwidth

Returns bandwidth information: current values and limits.

This function has no parameters.

The JSON response contains the following details:

- data – The data to be parsed.
    - current – The current bandwidths.
        - in – The inbound bandwidth.
        - out – The outbound bandwidth.
    - max – The maximum bandwidths.
        - in – The inbound limit.
        - out – The outbound limit.
- description – Describes the result of parsing/executing the command.
- status – \`SUCCESS\` if the command was parsed and executed successfully, \`FAIL\` if not.

A typical response in parsed JSON format is shown here: getBandwidth

SetBandwidthLimit

Enforces a limit on input and output bandwidth.

This function has the following parameters:

Parameter Name	Mandatory	Default Value	Description
in	true	*(null)*	Maximum input bandwidth. 0 means disabled. CLI connections are not affected.
out	true	*(null)*	Maximum output bandwidth. 0 means disabled. CLI connections are not affected.

An example of the setBandwidthLimit interface is:

```
setBandwidthLimit in=400000 out=300000
```

This sets the inbound bandwidth limit to 400,000, and the outbound bandwidth limit to 300,000 bytes/sec.

The JSON response contains the following details:

- data – Provides the following information for current values and maximum values:
  - in = The inbound bandwidth current value / maximum value.
  - out = The outbound bandwidth current value / maximum value.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: setBandwidthLimit

Services

The services API functions allow the user to manipulate the Networking Services that are added to an EMS application. These services are also called acceptors.

listServices

Returns the list of available services.

This interface has no parameters.

The JSON response contains the following details:

- data – Provides the following information for each protocol:
    - acceptedConnectionsCount – The number of active connections using the service.
    - appId – The ID of the application linked to the service.
    - appName – The name of the application linked to the service.
    - droppedConnectionsCount – The number of dropped connections.
    - enabled - `true` if the service is enabled, `false` if not.
    - id = ID of the service.
    - ip = The IP address bound to the service.
    - port – The port bound to the service.
    - protocol – The protocol bound to the service.
    - sslCert – The SSL certificate.
    - sslKey – The SSL certificate key.
    - useLengthPadding – `true` if padding is enabled, `false` if not (for some protocols only).
    - waitForMetadata – `true` if metadata is required, `false` if not (for some protocols only).
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: listServices

createService

Creates a new service.

This function has the following parameters:

Parameter Name	Mandatory	Default Value	Description
ip	true	*(null)*	The IP address to bind on.
port	true	*(null)*	The port to bind on.
protocol	true	*(null)*	The protocol stack name to bind on.
sslCert	false	*(null)*	The SSL certificate to be used.
sslKey	false	*(null)*	The SSL certificate key to be used.

An example of the setConnectionsLimit interface is:

```
createService ip=0.0.0.0 port=9556 protocol=inboundRtmp
```

This creates an acceptor for every hosted IP to accept inbound RTMP requests on port 9556.

The JSON response contains the following details:

- data – The data to parse.
    - acceptedConnectionsCount – The number of active connections using the service.
    - appId – The ID of the application using the service.
    - appName – The name of the application using the service.
    - droppedConnectionsCount – The number of dropped connections.
    - enabled - `true` if the service is enabled, `false` if not.
    - id = ID of the service.
    - ip = The IP address bound to the service.
    - port – The port bound to the service.
    - protocol – The protocol bound to the service.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: createService

enableService

Enable or disable a service.

This function has the following parameters:

Parameter Name	Mandatory	Default Value	Description
id	true	*(null)*	The id of the service.
enable	true	*(null)*	1 to enable, 0 to disable service.

An example of the enableService interface is:

```
enableService id=5 enable=0
```

This *disables* the service with an id of 5.

The JSON response contains the following details:

- data – The data to parse.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: enableService

shutdownService

Terminates a service

This function has the following parameters:

Parameter Name	Mandatory	Default Value	Description
id	true	*(null)*	The id of the service

An example of the shutdownService interface is:

```
shutdownService id=5
```

This shuts down the service with an id of 5.

The JSON response contains the following details:

- data – The data to parse.
    - acceptedConnectionsCount – Number of active connections.
    - appId – ID of application using the service.
    - appName – Application using the service.
    - droppedConnectionsCount – Number of dropped connections.
    - enabled - `true` if the service is enabled, `false` if not.
    - id – ID of the service.
    - ip – IP address used by the service.
    - port – Port used by the service.
    - protocol – Protocol used by the service.
    - sslCert – SSL certificate.
    - sslKey – SSL certificate key.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: shutdownService

Miscellaneous API Functions

help

This function prints out descriptions of the API in JSON format.

This function has no parameters.

The JSON response contains the following details:

- data – The data to parse.
    - command – The name of a valid command.
    - deprecated – Is `true` if the command is deprecated, `false` if not.
    - description – Describes the use of the command.
    - parameters – Parameter settings for the command.
        - defaultValue – The default value if the parameter is omitted.
        - description – Describes the use of the parameter.
        - mandatory – Is `true` if the parameter is mandatory, `false` if not.
        - name – The name of a parameter for the command.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: help

setAuthentication

Will enable/disable RTMP authentication

This function has the following parameters:

Parameter Name	Mandatory	Default Value	Description
enabled	true	*(null)*	1 to enable, 0 to disable authentication

An example of the setAuthentication interface is:

```
setAuthentication enabled=1
```

This enables authentication.

The JSON response contains the following details:

- data – The data to parse.
    - enabled – `true` if authentication is enabled, `false` if not.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: setAuthentication

setLogLevel

Change the log level for all log appenders

Parameter Name	Mandatory	Default Value	Description
level	true	*(null)*	A value between 0 and 7. 0 - no logging; 7 - detailed logs

An example of the setLogLevel interface is:

```
setLogLevel level=5
```

This sets the log level to 5.

The JSON response contains the following details:
- data – Nothing to parse for this command.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: setLogLevel

version

Returns the versions for framework and this application

This function has no parameters.

The JSON response contains the following details:
- data – Contains an integer representing the version.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: version

quit

This function quits the ASCII Command Line Interface (CLI)

This function has no parameters.

The JSON response contains the following details:
- data – Nothing to parse for this command.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: quit

shutdownServer

This function ends the server process, completely shutting down the EMS. This function must be called twice, once with a blank parameter, allowing you to obtain the shutdown key, and then a second time with the key, which actually causes the EMS to terminate.

Parameter Name	**Mandatory**	**Default Value**	**Description**
Key	false	*(null)*	The key to shutdown the server. shutdownServer must be called without the key to obtain the key and once again with the returned key to shutdown the server

An example of the shtudownServer interface is:

```
shutdownServer
```

The JSON response contains the following details:

- data – The data to parse
    - key – The key that needs to be used in a subsequent call to shutdownServer.
- description – Describes the result of parsing/executing the command.
- status – `SUCCESS` if the command was parsed and executed successfully, `FAIL` if not.

A typical response in parsed JSON format is shown here: shutdownServer