StreamEngine File Transcoder Documentation

Release 1.0.110717

Zipreel Inc.

Dec 07, 2017

CONTENTS

Contents:

CHAPTER

ONE

REST API DOCUMENTATION

This section documents StreamEngine's REST API. The key components of this API include

1. Authentication
2. Locations
3. Configurations
4. JobSetups
5. Jobs
6. Cluster Status

1.1 Authentication

The first step is to typically query the system for an authentication token. This token should be used to authenticate every API call.

1.1.1 Details

Endpoint: /api/v0/api-token-auth/

Supported methods: POST

Request parameters/keys:

1. username (string): username allowed to operate transcoder e.g. test
2. password (string): password associated with username, e.g. test

Response parameters/keys:

1. token (string): API token corresponding to the username/password

1.1.2 Code sample

The following sample code snippet in python requests for a token (with username as "test" and password as "test").:

```
import requests # For HTTP request-response handling

# URL (this value could be different in your installation)
baseurl = "http://li1249-5.members.linode.com:8080"
```

```
api_auth_token_url = "/api/v0/api-token-auth/"

userdict = {"username": "test", "password": "test"}

response = requests.post(baseurl + api_auth_token_url, data=userdict, headers={
↪"Content-Type": "application/json"})
```

A more detailed code sample can be found at http://li1249-5.members.linode.com:8080/documentation_new/api-usage-tutorial/sample-code-for-api-access-to-auth-token/.

1.2 Locations

Once the authentication token is obtained, the next step is to to create Locations. Before StreamEngine can begin transcoding files, it needs to know where the source files reside and where the outputs should go once transcoding finishes. In other words, the input and output file Locations need to be specified. The following steps show you how to specify input and output locations via REST APIs.

Input or outputs locations can be HTTP URLs (prefixed by http://) or NFS mount points (prefixed by nfs://) or local folders.

1.2.1 Details

Endpoint: /api/v0/locations/

Supported methods: GET, POST, PUT, DELETE

Request parameters/keys:

1. id (integer): ID of Location (for GET, PUT, DELETE)
2. name (string): label to use for this location
3. location (string): HTTP URL or NFS mountpoint or local file/folder path
4. watch_flag (boolean): if this is a NFS mountpoint or local folder, and this parameter is set to TRUE, then this folder will be "watched" for new files
5. input_flag (boolean): TRUE if this is an input location and FALSE otherwise

Response parameters/keys:

1. id (integer): ID of Location
2. name (string): label used for this location
3. location (string): HTTP URL or NFS mountpoint or local file/folder path
4. mounted_loc (string): if location is mounted locally, the local mountpoint used
5. input_flag (boolean): TRUE if this is an input location and FALSE otherwise
6. watch_flag (boolean): if this is a NFS mountpoint or local folder, and this parameter is set to TRUE, then this folder will be "watched" for new files
7. media_set (array of dictionaries with one dictionary per file): this field captures the file characteristics for all files associated with this (input) location. Each dictionary has key/value pairs corresponding to the *Medias* response parameters/keys
8. error_desc (string): error string associated with the location (typically if it is inaccessible/unreachable or cannot be mounted)

9. creat_output_folder_flag (boolean): in the specified output location, if the input file is inside a sub-directory, create a sub-directory inside the output location with the same name and place the final output transcoded files inside this sub-directory
10. match_input_filename_flag (boolean): give the final output transcoded file the same name as the input file (only valid when there is one ProfileConfig used)
11. copy_companion_files_flag (boolean): copy any non-media companion files in the specified input location over to the final output location

1.2.2 Code sample

The following sample code snippet in python creates an input location:

```
import requests

# URL (this value could be different in your installation)
baseurl = "http://li1249-5.members.linode.com:8080"

relative_url = "/api/v0/locations/"

# Location info
input_location_dict = {'name': 'My input location 1',
                       'location': 'http://128.6.192.166/www/1-short.ts',
                       'watch_flag': False,
                       'input_flag': True,}

# POST /api/v0/locations/ to create a new location
response = requests.post(baseurl + relative_url, data=input_location_dict)
```

The following sample code snippet in python creates an output location.:

```
import requests

# URL (this value could be different in your installation)
baseurl = "http://li1249-5.members.linode.com:8080"

relative_url = "/api/v0/locations/"

# Location info
output_location_dict = {'name': 'My output location 1',
                        'location': '/home/native/output/',
                        'watch_flag': False,
                        'input_flag': False,}

# POST /api/v0/locations/ to create a new location
response = requests.post(baseurl + relative_url, data=output_location_dict)
```

A more detailed code sample can be found at http://li1249-5.members.linode.com:8080/documentation_new/api-usage-tutorial/sample-code-for-api-access-to-locations/

1.3 Medias

Endpoint: /api/v0/medias/

Supported methods: GET

Response parameters/keys:

1. id (integer): ID of Media file
2. name (string): Pathname of media file
3. url (string): Local or global URL to media file
4. num_audio_streams (integer): Number of audio streams in file
5. num_video_streams (integer): Number of video streams in file
6. input_flag (boolean): If this is an input file and part of an input Location
7. size_in_bytes (integer): Size of the file in bytes
8. video_set (array of dictionaries, one dictionary per video stream discovered): after the file characteristics are discovered, info. about the video streams is maintained and reported via this field. The fields of the per-video-stream dictionary are:
   (a) id (integer): ID of video stream
   (b) codec (string): Video codec used to encode video stream
   (c) width (integer): width of video frames (part of resolution)
   (d) height (integer): height of video frames (part of resolution)
   (e) bitrate_in_kbps (float): bit-rate in Kbps at which video stream is encoded
   (f) frames_per_second (float): framerate of video stream
   (g) duration_in_sec (float): Duration of video stream
   (h) num_frames (integer): Number of frames in video stream
   (i) progressive_or_interlaced (integer): whether the video stream is progressive or interlaced
9. audio_set (array of dictionaries, one dictionary per audio stream discovered): after the file characteristics are discovered, info. about the audio streams is maintained and reported via this field. The fields of the per-audio-stream dictionary are:
   (a) id (integer): ID of audio stream
   (b) codec (string): Audio codec used to encode audio stream
   (c) bitrate_in_kbps (float): bit-rate in Kbps at which audio stream is encoded
   (d) num_frames (integer): Number of frames in audio stream
   (e) num_channels (integer): Number of channels in audio stream
10. is_transcode_attempted (boolean): has transcoding been attempted for this file?
11. is_transcode_in_progress (boolean): is transcoding in progress for this file?
12. is_transcode_error_or_abort (boolean): did transcoding the file result in an error or was the transcode job aborted?
13. is_transcode_completed (boolean): has the transcoding for this file completed?
14. is_discovery_in_progress (boolean): are the characteristics of this file being discovered?
15. is_discovery_completed (boolean): is the discovery of file characteristics complete?
16. is_partial_discovery (boolean): is the discovery of file characteristics on a part of the file or all of it?
17. container (string): what is the container used to encapsulate the video and audio streams
18. content_hash (string): unique hash used internally to identify file contents

19. error_flag (boolean): is there an error associated with this file?
20. start_timecode (string): Start timecode in file

1.4 Configurations

Now that StreamEngine knows about input and output Locations, the next step is to create Configurations. A configuration is the set of parameters used to transcode input files. To support adaptive-stream outputs, each configuration consists of one or more profile-specific configurations and one common configuration. Each profile-specific configuration or ProfileConfig corresponds to the transcoding parameters used to generate one output profile in single/multi-stream output. Each common output configuration or CommonConfig corresponds to the common transcoding parameters used across profiles.

NOTE: Users not wanting to set configurations, and just re-use existing ones, can skip the subsection on creating configurations

1.4.1 Details

Endpoint 1: /api/v0/profileconfigs/

Supported methods: GET, POST, PUT, DELETE

Request/response parameters/keys (for ProfileConfig):

1. ID (integer): ID of ProfileConfig (for GET, PUT, DELETE)
2. name (string): A label for this stream configuration
3. width (integer): Target output video width (resolution)
4. video_maxrate (integer): Target max-rate (kbps) in VBR mode
5. height (integer): Target output video height (resolution)
6. video_muxrate (integer): Target mux-rate (kbps)
7. video_format (string): Target output video format with valid inputs choices including “progressive” or “interlaced”
8. video_framerate (integer): Target output video framerate. Valid inputs are [0, 1, 2]. 0 = full-rate, 1 = 1/2-rate, 2 = 1/4-rate
9. video_bitrate_mode (integer): Target output video bitrate mode. Valid inputs are [0, 1]. 0 = cbr, 1 = vbr

Endpoint 2: /api/v0/commonconfigs/

Supported methods: GET, POST, PUT, DELETE

Request/response parameters/keys (for CommonConfig):

1. id (integer): ID of CommonConfig (for GET, PUT, DELETE)
2. name (string): A label for this configuration
3. operating_mode (string): Operating mode. Valid inputs are: [‘transcoding’]
4. output_container (string): Output container to be used. Valid inputs are: [‘ts’, ‘mp4’, ‘hls’, ‘silverlight’]
5. multirate_segment_size_sec (integer): Segment size if multirate output is requested. Valid inputs are >= 2 and <= 4/20 (depending on whether ‘silverlight’ or ‘hls’ is the container type). This value will be ignored if output_container is not ‘hls’ or ‘silverlight’

6. gop_length (float): Target GOP length in seconds
7. ip_distance (integer): Target I/P frame distance
8. video_codec (string): Video codec to be used during transcode. Valid inputs are: ['h.264', 'mpeg2']
9. h264_profile (string): H.264 profile to be used (baseline, main, or high). This value will be ignored if video_codec is equal to 'mpeg2'
10. h264_quality (integer): Target output video quality setting between 1 (lowest) and 5 (highest). This value will be ignored if video_codec is equal to 'mpeg2'
11. multirate_type (string): Multirate output type. Valid inputs are: ['none']
12. audio_codec (string): Audio codec to be used during transcode. Valid inputs are: ['passthrough', 'aaclc', 'dolby5.1', 'ac3']
13. audio_volume (integer): Target audio volume (0-200). This value will be ignored when audio_codec is equal to 'passthrough'
14. audio_bitrate (float): Target audio bitrate (kbps). Valid inputs are codec-dependent (look at documentation). This value will be ignored when audio_codec is equal to 'passthrough'
15. secondary_audio_bitrate (float): Target audio bitrate (kbps) for secondary audio stream (if present). Valid inputs are codec-dependent (look at documentation). This value will be ignored when audio_codec is equal to 'passthrough'
16. audio_bitrate_backup (float): Backup audio bitrate (kbps) used for the case when we want to handle both Dolby 5.1 input and AC3 stereo input. If the input is stereo, then this value will be used. Otherwise, the primary bit-rate specified ('audio_bitrate') will be used. Valid inputs are codec-dependent (look at documentation). This value will be ignored when audio_codec is equal to 'passthrough'
17. primary_audio_downmix_to_stereo_flag (boolean): If the primary audio stream is 5.1, downmix to stereo if this flag is set to True
18. profileconfig_set (field): For POST and PUT, this is the ProfileConfig IDs to be used as part of this base configuration (multiple IDs can be specified in comma-separated manner (e.g. 11,12,13)); for GET, this is an array of dictionaries with one dictionary per associated ProfileConfig. Key,value pairs in each dictionary follow the model for ProfileConfig
19. detelecine_flag (boolean): Manually set this flag to True to deal with inputs that are known to have a 2:3 pulldown from 24fps to 29.97fps. When set to False, such assets will automatically get detected. In both cases, a combination of detelecine + soft telecine will be used by the Transcoder
20. video_aspect_ratio (string): Target aspect ratio. Valid inputs are ["0" (passthrough) or "1" (4:3) or "2" (16:9)]
21. adaptive_spatial_prefilter (integer): Target adaptive spatial prefilter [strength 0 - 32]
22. smil_output_flag (integer, optional): Generate SMIL file? Valid inputs are 0 (No) or 1 (Yes). This value will be ignored if output_container is not 'mp4' or 'ts'
23. closed_captions_flag (integer, optional): Write CC to output? Valid inputs are 0 (do not write) or 1 (write). EIA 608 and 708 captions present on the video PID will automatically get passed through to the output (independent of the value specified here)
24. single_package_flag (integer, optional): Multiplex output files into one package? Valid inputs are 0 (disable) or 1 (enable). This value will be ignored if output_container is not 'silverlight'
25. encryption_type (string): DRM vendor type. Valid inputs are: ['none']. Not supported yet
26. verimatrix_asset_type (string, optional): Asset type (Verimatrix)
27. buydrm_keyid (string, optional): Key ID (BuyDRM)
28. verimatrix_client_url (string, optional): Client URL (Verimatrix)

29. buydrm_userkey (string, optional): User key (BuyDRM)
30. verimatrix_iv_mode (string, optional): IV mode (Verimatrix)
31. verimatrix_encryptor_url (string, optional): Encryptor URL (Verimatrix)
32. vix_output_flag (integer, optional): Generate VIX output file? Valid inputs are 0 (No) or 1 (Yes). This value will be ignored if output_container is not 'ps'
33. verimatrix_resource_id (string, optional): Resource ID (Verimatrix)
34. verimatrix_max_segments (string, optional): Max. segments (Verimatrix)
35. buydrm_contentid (string, optional): Content ID (BuyDRM)
36. buydrm_mediaid (string, optional): Media ID (BuyDRM)

1.4.2 Code sample to use existing Configurations

To use an existing configuration, you need to make a GET request to /api/v0/commonconfigs/ and select the ID of the CommonConfig you want to use. E.g.:

```
import json
import requests

# URL (this value could be different in your installation)
baseurl = "http://li1249-5.members.linode.com:8080"

relative_url = "/api/v0/commonconfigs/"

# GET /api/v0/commonconfigs/
response = requests.post(baseurl + relative_url)
json_response = json.loads(response.content)

chosen_config = None
for config in json_response:
  if config['name'] == "RCN":
    chosen_config = config
    break

# Use chosen_config['id'] later on
```

1.4.3 Code sample to create Configurations

The steps below show you how to create ProfileConfigs and CommonConfigs using StreamEngine's REST APIs. We begin with a python code snippet to create SD and HD ProfileConfigs.:

```
import requests

# URL (this value could be different in your installation)
baseurl = "http://li1249-5.members.linode.com:8080"

relative_url = "/api/v0/profileconfigs/"

# SD profileconfig
sd_config_dict= {
      "name": "SD",
      "width": 640,
```

```
      "height": 480,
      "video_muxrate": 1500,
      "video_framerate": 0,
      "video_maxrate": -1,
      "video_bitrate_mode": 0,
      "video_format": "progressive"
      }


# POST /api/v0/profileconfigs/ to create SD configuration
response = requests.post(baseurl + relative_url, data=sd_config_dict)

# HD config
hd_config_dict= {
      "name": "HD",
      "width": 1280,
      "height": 720,
      "video_muxrate": 3000,
      "video_framerate": 0,
      "video_maxrate": -1,
      "video_bitrate_mode": 0,
      "video_format": "progressive"
      }

# POST /api/v0/configurations/ to create HD configuration
response = requests.post(baseurl + relative_url, data=hd_config_dict)
```

A more detailed code sample can be found at http://li1249-5.members.linode.com:8080/documentation_new/api-usage-tutorial/sample-code-for-api-access-to-profileconfigs/

We now look at a code snippet to create CommonConfigs using StreamEngine's REST APIs.:

```
import requests

# URL (this value could be different in your installation)
baseurl = "http://li1249-5.members.linode.com:8080"

relative_url = "/api/v0/commonconfigs/"

''' Create single-rate commonconfig '''
ts_config_dict= {
      "name": "SD-TS",
      "output_container": "ts",
      "video_codec": "h.264",
      "ip_distance": 3,
      "gop_length": 3,
      "h264_profile": "high",
      "h264_quality": 5,
      "adaptive_spatial_prefilter": 0,
      "video_aspect_ratio": "2",
      "audio_codec": "aaclc",
      "audio_bitrate": 256.0,
      "audio_volume": 100,
      "audio_bitrate_backup": 64.0,
      "primary_audio_downmix_to_stereo_flag": False,
      "secondary_audio_bitrate": 64.0,
      "detelecine_flag": False,
      "multirate_type": "none",
```

```
      "multirate_segment_size_sec": -1,
      "profileconfig_set": 12, # assuming this is the ID of a ProfileConfig created␣
↪earlier
      "encryption_type": "none"
      }

response = requests.post(baseurl + relative_url, data=ts_config_dict)
```

A more detailed code sample can be found http://li1249-5.members.linode.com:8080/documentation_new/api-usage-tutorial/sample-code-for-api-access-to-commonconfigs/

1.5 JobSetups

The next step is to create Jobsetups. A Jobsetup puts together input locations, the configuration to be applied during transcoding, and output locations. A Jobsetup is the last step before job creation. A user can manually add jobs to the queue using the /api/v0/jobs/ end-point or via the GUI. Alternatively, if the input locations include watch folders, then creating a Jobsetup will automatically create jobs for the files in these locations (or files that are added afterwards).

1.5.1 Details

Endpoint: /api/v0/job_setups/

Supported methods: GET, POST, PUT, DELETE

Request parameters/keys:

1. id (integer, optional): ID of JobSetup (for GET, PUT, DELETE)
2. name (string): A label for this job_setup
3. input_location (field): One or more (comma-separated) input location ids
4. jobs_priority (integer): A single integer value >= 1 (GOLD) or <= 3 (BRONZE)
5. config (integer): A single configuration ID
6. output_locations (field): One or more (comma-separated) output location IDs

Response parameters/keys:

1. id (integer): ID of JobSetup
2. name (string): A label for this job_setup
3. input_location (array of dictionaries with one dictionary for each input location): the serialized information for each input location as per the response model used above for Locations (and Medias)
4. jobs_priority (string): One value from “platinum”, “gold”, “silver”, or “bronze”
5. config (dictionary): Serialized information corresponding to the CommonConfig selected to be part of this JobSetup. Follows the same key,value pair schema used by CommonConfig (and ProfileConfig)
6. output_locations (array of integer IDs with one integer ID for each Location): Array of output location IDs

1.5.2 Code sample

The following sample code snippet in python creates a Jobsetup from scratch by creating input and output locations, a profileconfig, a commonconfig and then combining these components.:

```
import json
import requests

# URL (this value could be different in your installation)
baseurl = "http://li1249-5.members.linode.com:8080"

'''
Step 1: Create input location
'''
relative_url = "/api/v0/locations/"
input_location_dict = {'name': 'My input location 1',
                       'location': 'http://console.orbit-lab.org/input/ido125mbps.ts',
                       'watch_flag': False,
                       'input_flag': True,}
response = requests.post(baseurl + relative_url, data=input_location_dict)
input_location_id = json.loads(response.content['id'])

'''
Step 2: Create output location
'''
output_location_dict = {'name': 'My output location 1',
                        'location': 'nfs://node1.orbit-lab.org:/home/native/output/',
                        'watch_flag': False,
                        'input_flag': False,}
response = requests.post(baseurl + relative_url, data=output_location_dict)
output_location_id = json.loads(response.content['id'])

'''
Step 3a: create profileconfig
'''
relative_url = "/api/v0/profileconfigs/"

# SD profileconfig
sd_config_dict= {
      "name": "SD",
      "width": 640,
      "height": 480,
      "video_muxrate": 1500,
      "video_framerate": 0,
      "video_maxrate": -1,
      "video_bitrate_mode": 0,
      "video_format": "progressive"
              }

# POST /api/v0/profileconfigs/ to create SD profileconfig
response = requests.post(baseurl + relative_url, data=sd_config_dict)
sd_profileconfig_id = json.loads(response.content['id'])

'''
Step 3b: create commonconfig
'''
relative_url = "/api/v0/commonconfigs/"

''' Create single-rate commonconfig '''
ts_config_dict= {
      "name": "SD-TS",
      "output_container": "ts",
      "video_codec": "h.264",
```

```
      "ip_distance": 3,
      "gop_length": 60,
      "h264_profile": "high",
      "h264_quality": 5,
      "adaptive_spatial_prefilter": 1,
      "video_aspect_ratio": 2,
      "audio_codec": "aaclc",
      "audio_bitrate": 64.0,
      "audio_volume": 100,
      "audio_bitrate_backup": 64.0,
      "primary_audio_downmix_to_stereo_flag": False,
      "secondary_audio_bitrate": 64.0,
      "detelecine_flag": False,
      "multirate_type": "none",
      "multirate_segment_size_sec": -1,
      "streamconfig_set": sd_profileconfig_id,
      "encryption_type": "none"
      }

# POST /api/v0/profileconfigs/ to create single-rate commonconfig
response = requests.post(baseurl + relative_url, data=ts_config_dict)
ts_commonconfig_id = json.loads(response.content['id'])

'''
Step 4: create jobsetup
'''
relative_url = "/api/v0/job_setups/"
valid_job_setup_dict = {'name': "My Jobsetup 1",
                        'config': unicode(ts_commonconfig_id),
                        'input_location': [unicode(input_location_id),],
                        'output_locations': unicode(output_location_id),
                        'jobs_priority': 1,}

response = requests.post(baseurl + relative_url, valid_job_setup_dict)
```

A more detailed code sample can be found at http://li1249-5.members.linode.com:8080/documentation_new/api-usage-tutorial/sample-code-for-api-access-to-job_setups/

1.6 Jobs

The final step is to create Jobs. A user can either manually add jobs to the queue or have them auto-added by creating Jobsetups with input Locations that are watch folders (i.e. have the watch_flag variable set to TRUE).

1.6.1 Details

Endpoint: /api/v0/jobs/

Supported methods: GET, POST, PUT

Request parameters/keys:

1. id (integer, optional): ID of Job (for GET, PUT)
2. job_setup_id (integer): A single job_setup ID
3. input_location_id (integer, optional): A single input location ID

4. input_media (integer, optional): A single media ID
5. cur_state (string): State of job. Valid inputs are 'enqueued' for a new job or 'aborted' for an existing, incomplete job
6. job_priority (integer): A single integer value >= 1 (GOLD) or <= 3 (BRONZE)

Response parameters/keys:

1. id (integer): ID of Job
2. job_setup_id (integer): A single job_setup ID
3. is_complete (boolean): TRUE if job is complete, and FALSE otherwise
4. has_error (boolean): TRUE if transcoding the file resulted in an error and FALSE otherwise
5. input_location_id (integer): The input location ID
6. input_location_nfs_path (string): Full input NFS/CIFS path
7. input_location_mount_path (string): Input NFS/CIFS path to be mounted
8. input_location_dfs_uname (string): NFS/CIFS username used to mount the input location
9. input_location_dfs_passwd (string): NFS/CIFS password used to mount the input location
10. input_location_mount_path_includes_subdir (boolean): True if the input mounted path includes sub-directories and False otherwise
11. input_media (dictionary): the key/value pairs are the same as those for the *Medias* response parameters/keys
12. output_folder (string): Location where final output files will be placed
13. output_location_nfs_path (string): Full output NFS/CIFS path
14. output_location_mount_path (string): Output NFS/CIFS path to be mounted
15. output_location_dfs_uname (string): NFS/CIFS username used to mount the output location
16. output_location_dfs_passwd (string): NFS/CIFS password used to mount the output location
17. output_location_folder_create (boolean): in the specified output directory, if the input file is inside a sub-directory, create a sub-directory inside the output directory with the same name and place the final output transcoded files inside the sub-directory
18. output_location_match_input_filename (boolean): give the final output transcoded file the same name as the input file (only valid when one output profile is used)
19. output_location_copy_companion_files (boolean): copy any non-media companion files in the specified input location over to the final output location
20. job_type (string): used internally
21. job_params (dictionary): the key/value pairs are the same as those for the *CommonConfig* response parameters/keys
22. enhanced_mode (boolean): FALSE for now (used internally)
23. webserver_ipaddr (string): used internally
24. cur_state (string): Value indicating Job state
25. job_priority (string): Value indicating Job priority
26. input_storage_location: Value indicating where input media is stored (used internally)
27. input_frames_processed_per_sec (float): Processing speed in frames/sec

28. num_output_files (integer): Number of output files moved to the output folder
29. relative_url_to_output_files (string): relative URL path to view/access output files produced by this transcoding job
30. preset_width (integer): used internally
31. preset_height (integer): used internally
32. preset_video_muxrate (integer): used internally
33. preset_detelecine_flag (boolean): used internally
34. compound_job (boolean): used internally
35. master_m3u8_created (boolean): used internally
36. simple_job_list (string): used internally
37. smil_file_created (boolean): used internally
38. error_desc (string): error string associated with this job

1.6.2 Code Sample

The following sample code snippet in python creates a Job using an existing Jobsetup:

```
import json
import requests

# URL (this value could be different in your installation)
baseurl = "http://li1249-5.members.linode.com:8080"

'''
Step 1: Get available job_setups
'''
relative_url = "/api/v0/job_setups/"
response = requests.get(baseurl + relative_url)

# jobsetup id for first jobsetup
job_setup_id = json.loads(response.content[0]['id'])

'''
Step 2: Create a job with the first available Jobsetup
'''
relative_url = "/api/v0/jobs/"
valid_jobs_dict = {'job_setup_id': job_setup_id, 'job_priority': 2,}
response = requests.post(baseurl + relative_url, valid_jobs_dict)

job_info = json.loads(response.content)
job_ids = job_info["id"]

for job_id in job_ids:
    cur_job_url = relative_url + str(job_id) + "/"
    while True:
        job_response = requests.get(baseurl + cur_job_url)
        json_response = json.loads(job_response)
        if json_response and "is_complete" in json_response and json_response["is_
↪complete"]:
            if "has_error" in json_response and json_response["has_error"]:
                break
```

```
            else:
                if "cur_state" in json_response and (json_response["cur_state"] ==
↪"alldone" or json_response["cur_state"] == "aborted"):
                    break
                else:
                    job_response = requests.get(baseurl + cur_job_url)
                    json_response = json.loads(job_response)
                    time.sleep(2)
        else:
            job_response = requests.get(baseurl + cur_job_url)
            json_response = json.loads(job_response)
            time.sleep(2)
```

A more detailed code sample can be found at http://li1249-5.members.linode.com:8080/documentation_new/api-usage-tutorial/sample-code-for-api-access-to-jobs/

1.6.3 Values for the 'cur_state' field

The cur_state field can be used to track the state of a Job. It can take one of the following values:

- 'enqueued' - Job is in the queue and not started processing yet
- 'p0scheduled' - Job is being processed
- 'pdiscovered' - Job is being processed
- 'p1scheduled' - Job is being processed
- 'transferred' - Job is being processed
- 'discovered' - Job is being processed
- 'p2scheduled' - Job is being processed
- 'processed' - Job is being processed
- 'aborted' - Job is aborted
- 'p3scheduled' - Job is being processed
- 'ftransferred' - Job is being processed
- 'ftransferpending' - Job is being processed
- 'p4scheduled' - Job is being processed
- 'p5scheduled' - Job is being processed
- 'alldone' - Job is DONE

1.6.4 How to determine when a job is done?

A Job is done when, in the response, the 'is_complete' flag is set to True AND either the 'has_error' flag is set to True OR 'cur_state' field is set to 'alldone' OR 'aborted'. See sample code above.

1.7 Cluster Status

You can also get information on the status of the transcoding cluster in terms of the number of active servers, the number of active transcoder processes, and the number of busy transcoder processes. This information can be used

to determine the number of simultaneous jobs that can be executed (=the number of active transcoder processes), the current load (=the number of busy transcoder processes). It can also be used to debug failure scenarios.

1.7.1 Details

Endpoint: /api/v0/status/

Supported methods: GET

Request/response parameters/keys:

1. num_config_tcoders (integer): Expected number of transcode worker processes in cluster
2. num_active_tcoders (integer): Active transcode worker processes in cluster
3. num_active_servers (integer): Number of servers in cluster
4. num_busy_tcoders (integer): Busy transcode worker processes in cluster
5. streamengine_release_version (string): Installed release version of the StreamEngine file transcoder system
6. component_release_versions (array of strings): Release version of the components that make up the StreamEngine file transcoder system

1.7.2 Code sample

The following code snippet in python shows how to estimate the current system load in terms of the number of transcode jobs in progress as well as the number of "free slots" available.:

```
import json
import requests

# URL (this value could be different in your installation)
baseurl = "http://li1249-5.members.linode.com:8080"

'''
Step 1: Get cluster status
'''
relativeurl = "/api/v0/status/"
response = requests.get(baseurl + relativeurl)
json_response = json.loads(response.content)

print "Number of transcode jobs in progress:", json_response["num_busy_tcoders"]
print "Number of free slots:", json_response["num_active_tcoders"] - json_response[
↪"num_busy_tcoders"]
print "Number of failed transcode worker processes:", json_response["num_config_
↪tcoders"] - json_response["num_active_tcoders"]
print "StreamEngine release version:", json_response["streamengine_release_version"]
for idx,item in enumerate(json_response["component_release_versions"]):
    print "StreamEngine component %d release version: %s" % (idx, item,)
```

CHAPTER

TWO

FREQUENTLY ASKED QUESTIONS (FAQ)

- How do I check if connectivity to the StreamEngine transcoder system?

Answer: Issue a HTTP GET request to http://{domain-name-of-transcoder}/api/v0/status/ . If a HTTP 200 return code is received, connectivity to the transcoder is fine. If a timeout or a HTTP 404 is received, the transcoder is unreachable (problem with the network?) or down (problem with StreamEngine?) or there is a typo in the URL used (input/user error).

- How do I check status of ONLY active or incomplete jobs?

Answer: Issue a HTTP GET request to http://{domain-name-of-transcoder}/api/v0/jobs/?incomplete . A HTTP 200 return code is expected and the JSON response can be parsed to further infer status as shown below.:

```
import json
import requests

# URL (this value could be different in your installation)
baseurl = "http://li1249-5.members.linode.com:8080"

relative_url = "/api/v0/jobs/?incomplete"

response = requests.get(baseurl + relative_url)
json_response = json.loads(response.content)

for per_job_json in json_response:
      print per_job_json["id"], per_job_json["cur_state"]
```

- How do I check status in terms of the current system load, how many transcode jobs can you do, and how many are active right now?

Answer: Issue a HTTP GET request to http://{domain-name-of-transcoder}/api/v0/status/ and process the JSON response as shown in the code snippet below.:

```
import json
import requests

# URL (this value could be different in your installation)
baseurl = "http://li1249-5.members.linode.com:8080"

relative_url = "/api/v0/status/"

response = requests.get(baseurl + relative_url)
json_response = json.loads(response.content)

print "Number of transcode jobs in progress or current system load:", json_response[
↪"num_busy_tcoders"]
```

```
print "Number of free slots available:", json_response["num_active_tcoders"]
print "Number of free slots available right now:", json_response["num_active_tcoders
↪"] - json_response["num_busy_tcoders"]
print "Number of failed transcode worker processes:", json_response["num_config_
↪tcoders"] - json_response["num_active_tcoders"]
```

- How do I get a list of predefined transcoding "profiles"?

Answer: Issue a HTTP GET request to http://{domain-name-of-transcoder}/api/v0/commonconfigs/ and process the JSON response as shown in the code snippet below.:

```
import json
import requests

# URL (this value could be different in your installation)
baseurl = "http://li1249-5.members.linode.com:8080"

relative_url = "/api/v0/commonconfigs/"

# GET /api/v0/commonconfigs/
response = requests.post(baseurl + relative_url)
json_response = json.loads(response.content)

for config in json_response:
   print config
```

- How do I create a job?

Answer: Short answer, issue a HTTP POST request to http://{domain-name-of-transcoder}/api/v0/jobs/ with the job_setup ID included in the POST parameters sent. For more details, read the Sub-section on job_setups and jobs in the Section on REST API documentation.

- How do I set transcode job priority?

Answer: There are two ways to set the transcode job priority: (a) set the priority of the corresponding job_setup and (b) set the priority of the job when you are executing the HTTP POST to /api/v0/jobs/. The first approach is useful when the input location is a "watch folder" and you want jobs to get started when a new input file shows up in the input location/folder. The second approach is useful when the input location is a URL or a file with an external, user-defined script/program controlling the creation of the job on a per-file basis.

For more details, read the sub-section on job_setups and jobs in section 1 on REST API documentation.

- How do I abort/cancel a running job?

Answer: Issuing a HTTP PUT command to /api/v0/jobs/JOBID with the "cur_state" field set to "aborted" will cancel/abort the in-progress job.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search