Pipeline Documentation

Release 0.0.4

Andreas Aderhold

February 11, 2015

Contents

Contents:

CHAPTER 1

API

The pipeline can be driven through the HTTP protocol using a restful approach. It is therefore possible to use the advanced distributed queing system as well as the application assembling functionality through 3rd party applications or for batch processing.

The API follows an Restful archtictural approach and makes advanced use of HTTP verbs and JSON data structures.

In the following documentation the resource URIs are realtive to the service endpoint:

```
http://localhost:5000/api/v1
```

Localhost is used throughout this exmaple and refers to a development environment. Just replace the host/port part of the domain with your network installation.

1.1 Basics

Pipeline REST requests are HTTP requests as defined per RFC 2616. A typical REST action consists of sending an HTTP request to the Pipeline and waiting for the response. Like any HTTP request, a REST request to the Pipeline system contains a request method, a URI, request headers, and maybe a query string or request body. The response contains an HTTP status code, response headers, and maybe a response body.

To acceess the Pipeline through the API you need to send a properly encoded HTTP request to the service endpoint. The encoding of the request neesd to be UTF-8 and the mimetype needs to be set to `application/json` or sometimes `application/octet-stream`. You can use the cURL tool to test your requests, e.g:

```
curl -H "Content-type: application/json" -X GET \
    http://localhost:5000/api/v1/bundles
```

1.1.1 Resources

In a RESTful pardigm each URL represents a unique resource. Requests are safe, idempotent, and cacheable. Through the use of HTTP verbs certain actions can be expressed using HTTP protocol means, instead of defining a new and arbitrary vocabulary for operations on resources. While most browsers today only implement GET and POST verbs, this is not a limiting factor for APIs. For example, in order to delete a resource, the HTTP DELETE verb is used on a uinque resource URI.

1.1.2 Payloads and Buckets

In order to process some uploaded files by the pipeline the files need to be stored somewhere in the first place. We achieve this by providing upload buckets. Think of a bucket as of a remote upload floder you can drop files to. Much similar to a S3 bucket, but much more temporary and entirely managed by the pipeline service.

A payload forms the input set of files to be fed to the pipeline. Currently the API recognizes two different payloads types: buckets and http URLs. In order to distinguish the input payload type, we use the URI scheme. For example to feed the pipeline with the contents of a payload bucket, the corrcet way to tell this the API is:

```
bucket://<bucket_id>
```

For exmaple:

```
bucket://afc34398da0f890dsa890fd8s9afc
```

In order to use a URI as input source, just specify the http protocol, domain and other parts e.g.:

```
http://domain.tld/some/archive.zip
http://domain.tld/some/model.ply
```

The storage backend can be completely transparent. In the future it is planned to add more storage backends to the system like Amazon S3, Glacier, SVN, WebDAV, Hadoop, etc. For the time being only HTTP and pipeline managed buckets are working.

1.1.3 Jobs

Jobs kick off the pipleline processing and bundle all relevant information as well as provide a interface for interacting with a current processing job. For exmaple get status information, pause, delete or restart a job. Jobs are excuted asynchronously and distributed by the Celery queue system.

1.2 Basic Workflow

The basic workflow to use the pipeline is as follows:

Post a file to the server using a bucket:

```
POST /api/v1/buckets HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate, compress
Content-Length: 12888639
Content-Type: application/octet-stream
Host: pipelineserver.tld
X-Filename: test.ply

<binary data not shown>
```

You get back a JSON response containing a cleartext message, a bucket ID and the filename of your uploaded date within the bucket. Save the ID and the filename in your application, you will need it later. You can upload a zip file as well.

Note: For the moment, use a zip file to upload multiple models and/or assets to the server. The POST more files to a existing bucket is not implemented yet.

Get a list of templates to select from:

```
GET /api/v1/bundles
Accept: application/json
Accept-Encoding: gzip, deflate, compress
Host: pipelineserver.tld
```

You can alos perform this step before uploading. For exmaple you could call the URL one time your application starts and cache the results. You will want to use the "name" field of the template you want to use for processing when starting a job.

Add a job to the processing queue and start it:

```
POST /api/v1/jobs
Accept: application/json
Accept-Encoding: gzip, deflate, compress
Host: pipelineserver.tld

{
    "payload": "bucket://<bucket_id>",
    "payload_filename": "herkules.ply",
    "template": "basic"
}
```

You will get back more information about this job:

```
{
    "task_id": "123",
    "job_url":   "full.host/v1/jobs/123",
    "progress_url": "full.host/v1/stream/123"
}
```

The most imporant information is the `job_url` which is used to freuqently query the backend about the status of a job:

```
GET /api/v1/jobs/123
Accept: application/json
Accept-Encoding: gzip, deflate, compress
Host: pipelineserver.tld
```

Do not query the pipeline to often this way (about every 5-10 seconds is proabably enough). To save transport overhead, you can resport to making a HEAD request only in order to check status. If status is 200 then you can issue an addtional request to get the rest of the information.

Additionally you can make use of the `progress_url` to get realtime-push messages about the status. This URL implements the EventSource push protocol.

> **Warning:** Do not use the push messages to determine if the conversion is complete as they are not relieable enougth to do so.

In case the job is still running a simple HTTP code of 102 is returend. If the job completed you will get a 200 return code as well as the following information:

```
{
    "download_url":"http:/domain.tld/somplace/download.zip",
    "preview_url": "htt://domain.tld/someplace/index.html"
}
```

With the download and preview URLs you then can get the completed job output and download it or redirect a browser to the preview HTML page.

All URLs returned by the API are absolute, you do not need to keep track of base URLs, just use the returend values.

For a detailed and up-to date description of JSON fields and responses, see below.

1.3 Overview

HTTP verb	URI	Return codes	Description
GET	/	200	API version, help URL and other relevant information
GET	/bundles	200	Returns a JSON formatted list of application bundles
GET	/buckets	200,404	NOOP: A list of payload buckets so far
POST	/buckets	201,500,420,415	Upload a file to the server which creates a bucket
GET	/buckets/<bucket-id>	200,404	NOOP: A list of files in the bucket with the resp. ID
POST	/buckets/<bucket-id>	201, 404	NOOP: Upload another file to a specific resource bucket (noop as of yet)
GET	/jobs	200,404	Returns a list of jobs, 404 if no jobs are found
POST	/jobs	201,400,415,403	Add a new job to the queuing system. By default processing starts immediately.
GET	/jobs/<task-id>	200, 102	Returns status of a job with given GUID. Will return HTTP status code 102 if the job can't be found or is still processing. Returns a 200 if job did complete. You can make a HEAD request without sending data to make the lookup quicker. If the job did complete, JSON response includes details. It is probably best to not query the backend too often (maybe every 5 seconds).
GET	/stream/<task-id>/		Live updates of processing events. This endpoint provides a push service with EventSource type messages. It can be used to display realtime status messages.

1.4 POST /buckets

modelconvert.api.views.**add_bucket**()

This methods allows for uploading arbitrary files to the server which later are used for processing:

```
POST /buckets
```

The HTTP request requires a filename header to identify the file:

```
X-Filename: herkules.ply
```

Additionally the Content-Type header needs to be set to:

```
Content-Type: application/octet-stream
```

You can then stream binary content in the body.

Note that you have to specify a filename even if you stream the file directly from an application. This is required in case you want to upload other assets to the bucket which are name dependent (for exmaple when using the metadata template).

After successful upload, a bucket ID among with a status message and the filename under which the data is stored within he bucket.

1.5 POST /buckets/<bucket_id>

modelconvert.api.views.**add_to_bucket**()
Not implemented yet.

Additionally with the use of this id it is possible to upload multiple files to the same bucket sequentially. e.g.:

```
POST /buckets/id123
```

This feature is not yet implemented. For the moment, if you like to upload multiple files to the same bucket, please use a ZIP file.

1.6 POST /jobs

modelconvert.api.views.**add_job**()
Adding a job to the processing queue either by fetching data from a known source or by using a previously created bucket full off resources (see ..):

```
{
    "payload": "bucket://1234afdsafdsfdsa232",

    // for the moment you also need to specify the filename you want
    "payload_filename: "herkules.ply"

    // alternatively, to load from a URI use this instead:
    "payload": "http://someplace.zip",

    // all the following are optional:
    "email_to": "some@address.com",

    // array of strings representing meshlab filters. leave out
    // "meshlab" entry at all if no meshalb pre-processing is desired
    "meshlab": [
        "Remove Duplicate Faces",
        "Remove Duplicated Vertex",
        "Remove Zero Area Faces",
        "Remove Isolated pieces (wrt Face Num.)",
        "Remove Unreferenced Vertex",
        "Extract Information"
    ]

    // one out of the list of names you get with /bundles
    // always use the "name" field you get from GET /bundles as the name
    // might change. You can cache the names locally but be sure to expire
    // once in a while.
    "template": "basic",

    // the bundle name to be used for this job
    // in the future its also possible to override templte specific settings and options (shown
    //"bundle": {
    //    "name": "modelconvert.bundles.pop",   // this can also contain a bundle spec and relat
    //    "settings": {
    //        "aopt.pop": true,
    //        "aopt.command": "{command} {input} {output} -what -ever={0} -is -required",
    //        "meshlab.enabled": false,
    //   }
```

```
    // }
}
```

For exmaple a simple payload to convert a single model without meshalb sourced from a URL could look like this:

```
{
    "payload": "http://domain.tld/model.obj",
    "template": "basic"
}
```

In return you will get a json response with various data about your request:

```
{
    // clear text informational message, HTTP status code serves as numeric indicator
    "message": "Job accepted with ID 123",

    // the task ID the job is running on
    "task_id": "123",

    // poll URI for checking less frequently for results
    "job_url":   "full.host/v1/jobs/123",

    // URI for status updates through push protocl. This implements
    // the W3C EventSource specification. So your client needs to
    // support this in order to reciece push updates.
    "progress_url": "full.host/v1/stream/123",
}
```

1.7 GET /jobs/<task-id>

modelconvert.api.views.**job_status**()

Check status of a specific job. Note that currently this maps to a Celery Task ID. Later it will be a seperate entity which can have many associated celery tasks.

Note that Celery returns PENDING if the Task ID is non existant. This is an optimization and could be recitified like so: http://stackoverflow.com/questions/9824172/find-out-whether-celery-task-exists

When results are ready we provide json data response with:

```
{
    "message": "Conversion ready.",
    "download_url":"http:/domain.tld/somplace/download.zip",
    "preview_url": "htt://domain.tld/someplace/index.html",
}
```

1.8 GET /stream/<task-id>/

modelconvert.api.views.**stream**()

Push EventSource formatted messages to the client. This can be used to display real-time status information.

CHAPTER 2

Deployment

– Work in progress –

2.1 App Configuration

In production environments, you need to configure the application through environment variables as well. There are many ways to do this: Webserver config, startup script, wsgi file, virtualenv loaders, etc.

Note: The env variables also must be set when running the celery worker daemon. Make sure that debugging is turned off in your production configuration.

2.2 Provisioning a production system

In order to deploy the application in a prodcution environment, you need to provision your deployment machine accordingly. There are severals ways to do this automatically with tools like **'Puppet'_** or Chef. You can of course do this manually as well.

2.2.1 Celery

In order to run the Celery deamon on your production site, please use the generic init/upstart script provided with celery. For more information see the **'daemonizing'_** chapter of the Celery documentation or refer to your devops people ;)

2.2.2 Xvfb

In order to use meshlab, you also need a running X11 instance or xvfb as DISPLAY number 99 if you are running a headless setup (the display number can be overridden in you config file). Plese refer to your Linux distribution of how to setup xvfb.

2.2.3 Webserver

Depending on your system, you can deploy using Apache **'mod_wsgi'_** for convenience. The more sensible option however is **'nginx'_**/**'uwsgi'_**. More detailed info on how to deploy can be found here:

http://flask.pocoo.org/docs/deploying/

2.2.4 Flower

There's an nice tool called Flower to graphically manage and monitor the celery task queue. We highly recommend it for debugging purposes on the production system. It has been installed with the requirement.txt loading business above. So you should be ready to go. Please refer to the Flower manual for more information.

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

A

J

S