Pipewelder Documentation

Release 0.1.4

Jeff Klukas

March 09, 2015

Contents

CHAPTER 1

Pipewelder

Pipewelder is a framework that provides a command-line tool and Python API to manage AWS Data Pipeline jobs from flat files. Simple uses it as a cron-like job scheduler.

Source https://github.com/SimpleFinance/pipewelder

Documentation http://pipewelder.readthedocs.org

PyPI https://pypi.python.org/pypi/pipewelder

1.1 Overview

Pipewelder aims to ease the task of scheduling jobs by defining very simple pipelines which are little more than an execution schedule, offloading most of the execution logic to files in S3. Pipewelder uses Data Pipeline's concept of data staging to pull input files from S3 at the beginning of execution and to upload output files back to S3 at the end of execution.

If you follow Pipewelder's directory structure, all of your pipeline logic can live in version-controlled flat files. The included command-line interface gives you simple commands to validate your pipeline definitions, upload task definitions to S3, and activate your pipelines.

1.2 Installation

Pipewelder is available from PyPI via `pip` and is compatible with Python 2.6, 2.7, 3.3, and 3.4:

```
pip install pipewelder
```

The easiest way to get started is to clone the project from GitHub, copy the example project from Pipewelder's tests, and then modify to suit:

```
git clone https://github.com/SimpleFinance/pipewelder.git
cp -r pipewelder/tests/test_data my-pipewelder-project
```

If you're setting up Pipewelder and need help, feel free to email the author.

1.3 Development

To do development on Pipewelder, clone the repository and run `make` to install dependencies and run tests.

1.4 Directory Structure

To use Pipewelder, you provide a template pipeline definition along with one or more directories that correspond to particular pipeline instances. The directory structure looks like this (see test_data for a working example):

```
pipeline_definition.json
pipewelder.json <- optional configuration file
my_first_pipeline/
    run
    values.json
    tasks/
        task1.sh
        task2.sh
my_second_pipeline/
...
```

The `values.json` file in each pipeline directory specifies parameter values that are used modify the template definition including the S3 paths for inputs, outputs, and logs. Some of these values are used directly by Pipewelder as well.

A `'ShellCommandActivity <http://docs.aws.amazon.com/datapipeline/latest/DeveloperGuide/dp-object-shellcommandactivity.html>'__` in the template definition simply looks for an executable file named `run` and executes it. `run` is the entry point for whatever work you want your pipeline to do.

Often, your `run` executable will be a wrapper script to execute a variety of similar tasks. When that's the case, use the `tasks` subdirectory to hold these definitions. These tasks could be text files, shell scripts, SQL code, or whatever else your `run` file expects. Pipewelder gives `tasks` folder special treatment in that the CLI will make sure to remove existing task definitions when uploading files.

1.5 Using the Command-Line Interface

The Pipewelder CLI should always be invoked from the top-level directory of your definitions (the directory where `pipeline_definition.json` lives). If your directory structure matches Pipewelder's expectations, it should work without further configuration.

As you make changes to your template definition or `values.json` files, it can be useful to check whether AWS considers your definitions valid:

```
$ pipewelder validate
```

Once you've defined your pipelines, you'll need to upload the files to S3:

```
$ pipewelder upload
```

Finally, activate your pipelines:

```
$ pipewelder activate
```

Any time you change the `values.json` or `pipeline_definition.json`, you'll need to run the `activate` subcommand again. Because active pipelines can't be modified, the `activate` command will delete the existing pipeline and create a new one in its place. The run history for the previous pipeline will be discarded.

1.6 Acknowledgments

Pipewelder's package structure is based on python-project-template.

CHAPTER 2

Pipewelder Core API

The core Pipewelder API.

class pipewelder.core.**Pipeline**(*conn*, *s3_conn*, *template*, *dirpath*)

A class defining a single pipeline definition and associated tasks.

Create a Pipeline based on definition dict *template*.

dirpath is a directory containing a 'values.json' file, a 'run' executable, and a 'tasks' directory. *conn* is a DataPipelineConnection and *s3_conn* is an S3Connection.

activate()

Activate this pipeline definition in AWS.

Deletes the existing pipeline if it has previously been activated.

Returns `True` if successful.

api_objects()

Return a dict containing the pipeline objects in AWS API format.

api_parameters()

Return a dict containing the pipeline parameters in AWS API format.

api_tags()

Return a list containing the pipeline tags in AWS API format.

api_values()

Return a dict containing the pipeline param values in AWS API format.

create()

Create a pipeline in AWS if it does not already exist.

Returns the pipeline id.

delete()

Delete this pipeline definition from AWS.

Returns `True` if successful.

is_valid()

Returns `True` if the pipeline definition validates to AWS.

put_definition()

Put this pipeline definition to AWS.

Returns `True` if successful.

upload()
Uploads the contents of *dirpath* to S3.

The destination path in S3 is determined by 'myS3InputDirectory' in the 'values.json' file for this pipeline. Existing contents of the 'tasks' subdirectory are deleted.

Returns `True` if successful.

class pipewelder.core.**Pipewelder**(*conn*, *template_path*, *s3_conn=None*)
A collection of Pipelines sharing a definition template.

conn is a `boto.datapipeline.layer1.DataPipelineConnection` instance used to manipulate added pipelines, *s3_conn* is a `boto.s3.connection.S3Connection` used to upload pipeline tasks to S3, and *template_path* is the path to a local file containing the template pipeline definition.

activate()
Activate all pipeline definitions, deleting existing pipeline if needed.

Returns `True` if successful.

add_pipeline(*dirpath*)
Load a new `Pipeline` object based on the files contained in *dirpath*.

are_pipelines_valid()
Returns `True` if all pipeline definition validate with AWS.

delete()
Delete all pipeline definitions.

Returns `True` if successful.

put_definition()
Puts definitions for all pipelines.

Returns `True` if successful.

upload()
Upload files to S3 corresponding to each pipeline and its tasks.

Returns `True` is successful.

validate()
Synonym for `are_pipelines_valid()`.

pipewelder.core.**adjusted_to_future**(*timestamp*, *period*)
Return *timestamp* string, adjusted to the future if necessary.

If *timestamp* is in the future, it will be returned unchanged. If it's in the past, *period* will be repeatedly added until the result is in the future.

All times are assumed to be in UTC.

```
>>> adjusted_to_future('2199-01-01T00:00:00', '1 days')
'2199-01-01T00:00:00'
```

pipewelder.core.**bucket_and_path**(*s3_uri*)
Return a bucket name and key path from *s3_uri*.

```
>>> bucket_and_path('s3://pipewelder-bucket/pipewelder-test/inputs')
('pipewelder-bucket', 'pipewelder-test/inputs')
```

pipewelder.core.**definition_from_file**(*filename*)
Return a dict containing the contents of pipeline definition *filename*.

pipewelder.core.**definition_from_id**(*conn*, *pipeline_id*)
Return a dict containing the definition of *pipeline_id*.

conn is a DataPipelineConnection object.

pipewelder.core.**fetch_default**(*params*, *key*)
Return the default associated with *key* from parameter list *params*.

If no default, returns None. >>> p = [{'type': 'String', 'id': 'myParam', 'default': 'foo'}] >>> fetch_default(p, 'myParam') 'foo' >>> p = [{'type': 'String', 'id': 'myParam'}] >>> fetch_default(p, 'myParam')

pipewelder.core.**fetch_field_value**(*aws_response*, *field_name*)
Return a value nested within the 'fields' entry of dict *aws_response*.

The returned value is the second item from a dict with 'key' *field_name*.

```
>>> r = {'fields': [{'key': 'someKey', 'stringValue': 'someValue'}]}
>>> fetch_field_value(r, 'someKey')
'someValue'
```

pipewelder.core.**parse_period**(*period*)
Return a timedelta object parsed from string *period*.

```
>>> parse_period("15 minutes")
datetime.timedelta(0, 900)
>>> parse_period("3 hours")
datetime.timedelta(0, 10800)
>>> parse_period("1 days")
datetime.timedelta(1)
```

pipewelder.core.**parsed_object**(*conn*, *pipeline_id*, *object_id*)
Return an object dict as evaluated by Data Pipeline.

pipewelder.core.**parsed_objects**(*conn*, *pipeline_id*, *object_ids*)
Return a list of object dicts as evaluated by Data Pipeline.

pipewelder.core.**state_from_id**(*conn*, *pipeline_id*)
Return the *@pipelineState* string for object matching *pipeline_id*.

conn is a DataPipelineConnection object.

CHAPTER 3

Pipewelder Util

pipewelder.util.**cd**(**args*, ***kwds*)

Change to a different directory within a limited context.

CHAPTER 4

Pipewelder Command-Line Interface

The Pipewelder command-line interface.

pipewelder.cli.**build_pipewelder**(*conn*, *config*)
Return a Pipewelder object defined by *config*.

pipewelder.cli.**call_method**(*obj*, *name*)
Call the method *name* on *obj*.

pipewelder.cli.**entry_point**()
Zero-argument entry point for use with setuptools/distribute.

pipewelder.cli.**main**(*argv*)
Program entry point. :param argv: command-line arguments :type argv: list

pipewelder.cli.**pipewelder_configs**(*filename=None*, *defaults=None*)
Parse json from *filename* for Pipewelder object configurations.

Returns a dict which maps config names to dicts of options.

p