Fileflow Documentation

Release 0.0.3

Laura Lorenz, Miriam Sexton, David Barbarisi

Apr 12, 2017

Contents:

CHAPTER 1

Fileflow Overview

Fileflow is a collection of modules that support data transfer between Airflow tasks via file targets and dependencies with either a local file system or S3 backed storage mechanism. The concept is inherited from other pipelining systems such as Make, Drake, Pydoit, and Luigi that organize pipeline dependencies with file targets. In some ways this is an alternative to Airflow's XCOM system, but supports arbitrarily large and arbitrarily formatted data for transfer whereas XCOM can only support a pickle of the size the backend database's BLOB or BINARY LARGE OBJECT implementation can allow.

Installation

Fileflow has been tested on Python 2.7 and Airflow version 1.7.0.

You can install from github using pip:

```
pip install git+git://github.com/industrydive/fileflow.git#egg=fileflow
```

Or build from source by downloading the archive from github, unzipping, navigating to the root directory of the project and using setup.py:

```
python setup.py install
```

Concepts

The main components of fileflow are

- an operator superclass for your operators and sensors that works with multi-inheritance (*DiveOperator*)
- a task logic superclass that exposes the storage backend and convenience methods to serialize different data formats to text (*TaskRunner*)
- two storage drivers that handle the nitty gritty of passing your serialized data to either the local file system or S3 (*FileStorageDriver* or *S3StorageDriver*)

DiveOperator

The `DiveOperator` is a subclass of `airflow.models.BaseOperator` that mixes in the basic functionality that allows operators to define which task's data they depend on. You can subclass your own operators or sensors from `DiveOperator` exclusively, or mix it in via multi-inheritance with other existing operators to add the data dependency feature. For example, if you want to use the existing `airflow.operators.PythonOperator` and mix-in fileflow's file targeting feature, you could define your derived class as:

```
class DerivedOperator(DiveOperator, PythonOperator):
    ...
```

Given that definition, you can specify a given task's dependency on data output from an upstream task like so:

```
a_task = DerivedOperator(
        data_dependencies = {"dependency_key": "task_id_for_dependent_task"},
         ...
         )
```

Fileflow ships with exactly this type of derived operator, which we call `DivePythonOperator`, that works directly with the second component of fileflow, the `TaskRunner`, to make file detection of upstream targets easy.

TaskRunner

The `TaskRunner` is a superclass for you to use to define your task logic within your subclass's `run()` method. Many of the Airflow examples set up the task logic as plain functions that the `PythonOperator` calls; however our `DivePythonOperator` instead expects the class name of a subclass of `TaskRunner` and during operator execution will call `TaskRunner.run()` which should contain the actual logic of the task. To be more clear, see this example comparing the basic `PythonOperator` signature and our *`DivePythonOperator`* signature:

```
# vanilla airflow PythonOperator
normal_task = PythonOperator(
                task_id="some_unique_task_id",
                python_callable=a_function_name,
                dag=dag)

# fancy DivePythonOperator
fancy_task = DivePythonOperator(
                task_id="some_other_unique_task_id",
                python_object=AClassNameThatSubclassesTaskRunnerAndHasARunMethod,
                data_dependencies={"important_data": normal_task.task_id},
                dag=dag)
```

If you're a snowflake and don't like calling your main logic wrapper `run()` and, for example, want to call it `ninja_move()`, you can configure that on the operator in the DAG:

```
# fancy DivePythonOperator for someone who wants to be unique
fancy_and_unique_task = DivePythonOperator(
                task_id="yet_aother_unique_task_id",
                python_
↪object=AClassNameThatSubclassesTaskRunnerAndHasANinjaMoveMethod,
                python_method="ninja_move",
                data_dependencies={"important_data": normal_task.task_id},
                dag=dag)
```

All of this is to take advantage of the fact that we've done a bunch of work in `TaskRunner` to give it the ability to easily pass forward Airflow specific details to the storage driver to determine where it should write its target or where

its upstream task's wrote their targets. We've also written into `TaskRunner` several serialization methods that can serialize different file formats such as JSON, pandas DataFrames, and bytestreams for convenience. The idea is that by the time the `TaskRunner` has passed off some data to the appropriate storage driver, the data is already serialized into a single `str` representation or `BytesIO` object.

storage drivers

The two storage drivers shipped in `fileflow` deal with the nitty gritty of actually communicating with either the local file system in the case of *`FileStorageDriver`*, or with an S3 bucket in the case of *`S3StorageDriver`*. The storage driver needs to be able to

- derive a path or key name or names from the Airflow TaskInstance context data passed through by the TaskRunner for either upstream tasks (data dependencies) or the current task's target
- read and write to that path or key name

Since we're working with text I/O obviously this introduces a bunch of decisions the storage drivers have to be making regarding encoding/charsets, file read/write mode, path/key existence, and in the case of putting to S3 over HTTP, content types. All of this is handled by the respective storage driver; the interface for what a storage driver should implement is represented by the base *`StorageDriver`* class.

A full example

```
import logging
from datetime import datetime, timedelta

from airflow import DAG

from fileflow.operators import DivePythonOperator
from fileflow.task_runners import TaskRunner


# Define the logic for your tasks as classes that subclasses from TaskRunner
# By doing so it will have access to TaskRunner's convenience methods to read and␣
↪write files

# Here's an easy one that just writes a file.
class TaskRunnerExample(TaskRunner):
    def run(self, *args, **kwargs):
        output_string = "This task -- called {} -- was run.".format(self.task_
↪instance.task_id)
        self.write_file(output_string)
        logging.info("Wrote '{}' to '{}'".format(output_string, self.get_output_
↪filename()))


# Here's a more complicated one that will read the file from its upstream task, do␣
↪something, and then write its own file
# It also shows you how to override __init__ if you need to
class TaskRunnerReadExample(TaskRunner):
    def __init__(self, context):
        """
        An example how to write the init on a class derived from TaskRunner
        :param context: Required.
        """
```

```
        super(TaskRunnerReadExample, self).__init__(context)
        self.output_template = "Read '{}' from '{}'. Writing output to '{}'."

    def run(self, *args, **kwargs):
        # This is how you read the output of a previous task
        # The argument to read_upstream_file is based on the DAG configuration
        input_string = self.read_upstream_file("something")

        # An example bit of 'logic'
        output_string = self.output_template.format(
            input_string,
            self.get_input_filename("something"),
            self.get_output_filename()
        )

        # And write out the results of the logic to the correct file
        self.write_file(output_string)

        logging.info(output_string)


# Now let's define a DAG
dag = DAG(
    dag_id='fileflow_example',
    start_date=datetime(2030, 1, 1),
    schedule_interval=timedelta(minutes=1)
)

# The tasks in this DAG will use DivePythonOperator as the operator,
# which knows how to send a TaskRunner anything in the `data_dependencies` keyword
# so you can specify more than one file by name to be fed to a downstream task
t1 = DivePythonOperator(
    task_id="write_a_file",
    python_method="run",
    python_object=TaskRunnerExample,
    provide_context=True,
    owner="airflow",
    dag=dag
)

# We COULD set `python_method="run"` here as above, but "run" is the
# default value, so we're not bothering to set it
t2 = DivePythonOperator(
    task_id="read_that_file",
    python_object=TaskRunnerReadExample,
    data_dependencies={"something": t1.task_id},
    # remember how our TaskRunner subclass knows how to read the upstream file with␣
↪the key 'something'? This is why
    provide_context=True,
    owner="airflow",
    dag=dag
)
```

```
(fileflow)fileflow $ airflow run -sd fileflow/example_dags/ fileflow_example write_a_
↪file 2017-01-01
[2017-01-18 16:54:55,245] {__init__.py:36} INFO - Using executor SequentialExecutor
Sending to executor.
[2017-01-18 16:54:56,080] {__init__.py:36} INFO - Using executor SequentialExecutor
```

```
Logging into: /Users/llorenz/airflow/logs/fileflow_example/write_a_file/2017-01-
↪01T00:00:00
[2017-01-18 16:54:56,995] {__init__.py:36} INFO - Using executor SequentialExecutor
(fileflow)fileflow $ airflow run -sd fileflow/example_dags/ fileflow_example read_
↪that_file 2017-01-01
[2017-01-18 16:55:30,790] {__init__.py:36} INFO - Using executor SequentialExecutor
Sending to executor.
[2017-01-18 16:55:32,219] {__init__.py:36} INFO - Using executor SequentialExecutor
Logging into: /Users/llorenz/airflow/logs/fileflow_example/read_that_file/2017-01-
↪01T00:00:00
(fileflow)fileflow $ tree storage/
storage/
- fileflow_example
    - read_that_file
    |   - 2017-01-01
    - write_a_file
        - 2017-01-01

3 directories, 2 files
(fileflow)fileflow $ cat storage/fileflow_example/read_that_file/2017-01-01
Read 'This task -- called write_a_file -- was run.' from 'storage/fileflow_example/
↪write_a_file/2017-01-01'. Writing output to 'storage/fileflow_example/read_that_
↪file/2017-01-01'.
(fileflow)fileflow $ cat storage/fileflow_example/write_a_file/2017-01-01
This task -- called write_a_file -- was run.
(fileflow)fileflow $
```

CHAPTER 2

fileflow.operators package

fileflow.operators.dive_operator module

class fileflow.operators.dive_operator.**DiveOperator**(**args*, ***kwargs*)

Bases: airflow.models.BaseOperator

An operator that sets up storage and assigns data dependencies to the operator class.

This is intended as a base class for eg fileflow.operators.DivePythonOperator.

storage

Lazy load a storage property on access instead of on class instantiation. Something in the storage attribute is not deep-copyable which causes errors with airflow clear and airflow backfill which both try to deep copy a target DAG and all its operators, so we only want this property when we actually use it.

fileflow.operators.dive_python_operator module

class fileflow.operators.dive_python_operator.**DivePythonOperator**(*python_object*, *python_method='run'*, **args*, ***kwargs*)

Bases: *fileflow.operators.dive_operator.DiveOperator*, python_operator.PythonOperator

Python operator that can send along data dependencies to its callable. Generates the callable by initializing its python object and calling its method.

pre_execute(*context*)

CHAPTER 3

fileflow.task_runners package

fileflow.task_runners.task_runner module

class fileflow.task_runners.task_runner.**TaskRunner**(*context*)
Bases: object

get_input_filename(*data_dependency*, *dag_id=None*)
Generate the default input filename for a class.

Parameters

- **data_dependency** (*str*) – Key for the target data_dependency in self.data_dependencies that you want to construct a filename for.
- **dag_id** (*str*) – Defaults to the current DAG id

Returns File system path or S3 URL to the input file.

Return type str

get_output_filename()
Generate the default output filename or S3 URL for this task instance.

Returns File system path to output filename

Return type str

get_upstream_stream(*data_dependency_key*, *dag_id=None*)
Returns a stream to the file that was output by a seperate task in the same dag.

Parameters

- **data_dependency_key** (*str*) – The key (business logic name) for the upstream dependency. This will get the value from the self.data_dependencies dictionary to determine the file to read from.
- **dag_id** (*str*) – Defaults to the current DAG id.
- **encoding** (*str*) – The file encoding to use. Defaults to 'utf-8'.

Returns stream to the file

Return type stream

read_upstream_file(*data_dependency_key*, *dag_id=None*, *encoding='utf-8'*)
Reads the file that was output by a seperate task in the same dag.

Parameters

- **data_dependency_key** (*str*) – The key (business logic name) for the upstream dependency. This will get the value from the self.data_dependencies dictionary to determine the file to read from.
- **dag_id** (*str*) – Defaults to the current DAG id.
- **encoding** (*str*) – The file encoding to use. Defaults to 'utf-8'.

Returns Result of reading the file

Return type str

read_upstream_json(*data_dependency_key*, *dag_id=None*, *encoding='utf-8'*)
Reads a json file from upstream into a python object.

Parameters

- **data_dependency_key** (*str*) – The key for the upstream data dependency. This will get the value from the self.data_dependencies dict to determine the file to read.
- **dag_id** (*str*) – Defaults to the current DAG id.
- **encoding** (*str*) – The file encoding. Defaults to 'utf-8'.

Returns A python object.

read_upstream_pandas_csv(*data_dependency_key*, *dag_id=None*, *encoding='utf-8'*)
Reads a csv file from upstream into a pandas DataFrame. Specifically reads a csv into memory as a pandas dataframe in a standard manner. Reads the data in from a file output by a previous task.

Parameters

- **data_dependency_key** (*str*) – The key (business logic name) for the upstream dependency. This will get the value from the self.data_dependencies dictionary to determine the file to read from.
- **dag_id** (*str*) – Defaults to the current DAG id.
- **encoding** (*str*) – The file encoding to use. Defaults to 'utf-8'.

Returns The pandas dataframe.

Return type `pd.DataFrame`

run(**args*, ***kwargs*)

write_file(*data*, *content_type='text/plain'*)
Writes the data out to the correct file.

Parameters

- **data** (*str*) – The data to output.
- **content_type** (*str*) – The Content-Type to use. Currently only used by S3.

write_from_stream(*stream*, *content_type='text/plain'*)

write_json(*data*)
Write a python object to a JSON output file.

Parameters **data** (*object*) – The python object to save.

write_pandas_csv(*data*)
Specifically writes a csv from a pandas dataframe to the default output file in a standard manner.

Parameters **data** – the dataframe to write.

write_timestamp_file()
Writes an output file with the current timestamp.

CHAPTER 4

fileflow.storage_drivers package

Base StorageDriver class

class `fileflow.storage_drivers.storage_driver.`**`StorageDriver`**

Bases: `object`

A base class for common functionality and API amongst the storage drivers.

This is an example of mocking the read method inside a `DiveOperator`

```
# Inside a TestCase where self.sensor is a custom sensor

# Assign the desired input string to the return
# value of the 'read' method
attrs = {
     'read.return_value': 'output from earlier task'
}
self.sensor.storage = Mock(**attrs)

# Do stuff that will call storage.read()

# Later, we want to make sure the `read` method has been
# called the correct number of times
self.assertEqual(1, self.sensor.storage.read.call_count)
```

`execution_date_string`(*execution_date*)

Format the execution date per our standard file naming convention.

Parameters **`execution_date`** (*datetime.datetime*) – The airflow task instance execution date.

Returns The formatted date string.

Return type str

`get_filename`(*dag_id*, *task_id*, *execution_date*)

Return an identifying path or URL to the file related to an airflow task instance.

Concrete storage drivers should implement this method.

Parameters

- **dag_id** (*str*) – The airflow DAG ID.
- **task_id** (*str*) – The airflow task ID.
- **execution_date** (*datetime.datetime*) – The datetime for the task instance.

Returns The identifying path or URL.

Return type str

get_path(*dag_id*, *task_id*)

Return the path portion where files would be stored for the given task.

This should generally be the same as get_filename except without the filename (execution date) portion.

Parameters

- **dag_id** (*str*) – The DAG ID.
- **task_id** (*str*) – The task ID.

Returns A path to the task's intermediate storage.

Return type str

get_read_stream(*dag_id*, *task_id*, *execution_date*)

Parameters

- **dag_id** (*str*) – The airflow DAG ID.
- **task_id** (*str*) – The airflow task ID.
- **execution_date** (*datetime.datetime*) – The datetime for the task instance.

Returns

list_filenames_in_path(*path*)

Given a storage path, get all of the filenames of files directly in that path.

Note that this only provides names of files and is not recursive.

Parameters **path** (*str*) – The storage path to list.

Returns A list of only the filename portion of filenames in the path.

Return type list[str]

list_filenames_in_task(*dag_id*, *task_id*)

Shortcut method to get a list of filenames stored in the given task's path.

This is already implemented by depending on the unimplemented methods above. Override this if you need more flexibility.

Parameters

- **dag_id** (*str*) – The DAG ID of the task.
- **task_id** (*str*) – The task ID.

Returns A list of file names of files stored by the task.

Return type list[str]

read(*dag_id*, *task_id*, *execution_date*, *encoding*)
Read the data output from the given airflow task instance.

Concrete storage drivers should implement this method.

Parameters

- **dag_id** (*str*) – The airflow DAG ID.
- **task_id** (*str*) – The airflow task ID.
- **execution_date** (*datetime.datetime*) – The datetime for the task instance.
- **encoding** (*str*) – The encoding to use for reading in the data.

Returns The data from the file.

Return type str

write(*dag_id*, *task_id*, *execution_date*, *data*, **args*, ***kwargs*)
Write data to the output file identified by the airflow task instance.

Concrete storage drivers should implement this method.

Parameters

- **dag_id** (*str*) – The airflow DAG ID.
- **task_id** (*str*) – The airflow task ID.
- **execution_date** (*datetime.datetime*) – The datetime for the task instance.
- **data** (*str*) – The data to write.
- **args** – might be used in child classes, (currently used in S3StorageDriver)
- **kwargs** – same reasoning as args

write_from_stream(*dag_id*, *task_id*, *execution_date*, *stream*, **args*, ***kwargs*)
Write data to the output file identified by the airflow task instance.

Concrete storage drivers should implement this method.

Parameters

- **dag_id** (*str*) – The airflow DAG ID.
- **task_id** (*str*) – The airflow task ID.
- **execution_date** (*datetime.datetime*) – The datetime for the task instance.
- **data** (*stream*) – A stream to the data to write
- **args** – might be used in child classes, (currently used in S3StorageDriver)
- **kwargs** – same reasoning as args

exception fileflow.storage_drivers.storage_driver.**StorageDriverError**
Bases: exceptions.Exception

Base storage driver Exception.

fileflow.storage_drivers.file_storage_driver module

class fileflow.storage_drivers.file_storage_driver.**FileStorageDriver**(*prefix*)
Bases: *fileflow.storage_drivers.storage_driver.StorageDriver*

Read and write to the local file system.

check_or_create_dir(*dir*)
Make sure our storage location exists.

Parameters **dir** (*str*) – The directory name to look for and create if it doesn't exist..

Returns

get_filename(*dag_id*, *task_id*, *execution_date*)

get_path(*dag_id*, *task_id*)

get_read_stream(*dag_id*, *task_id*, *execution_date*)

list_filenames_in_path(*path*)

read(*dag_id*, *task_id*, *execution_date*, *encoding='utf-8'*)

write(*dag_id*, *task_id*, *execution_date*, *data*, **args*, ***kwargs*)

write_from_stream(*dag_id*, *task_id*, *execution_date*, *stream*, **args*, ***kwargs*)

fileflow.storage_drivers.s3_storage_driver module

class fileflow.storage_drivers.s3_storage_driver.**S3StorageDriver**(*access_key_id*, *secret_access_key*, *bucket_name*)
Bases: *fileflow.storage_drivers.storage_driver.StorageDriver*

Read and write to S3.

get_filename(*dag_id*, *task_id*, *execution_date*)

get_key_name(*dag_id*, *task_id*, *execution_date*)
Formats the S3 key name for the given task instance.

Parameters

- **dag_id** (*str*) – The airflow DAG ID.
- **task_id** (*str*) – The airflow task ID.
- **execution_date** (*datetime.datetime*) – The execution date of the task instance.

Returns The S3 key name.

Return type str

get_or_create_key(*key_name*)
Get a boto Key object with the given key name. If the key exists, returns that. Otherwise creates a new key.

Parameters **key_name** (*str*) – The name of the S3 key.

Returns A boto key object.

Return type boto.s3.key.Key

get_path(*dag_id*, *task_id*)

get_read_stream(*dag_id*, *task_id*, *execution_date*)

list_filenames_in_path(*path*)
This requires some special treatment. The path here is the full url path. For boto/s3 we need to strip out the s3 protocol and bucket name to only get the prefix.

Everywhere else in life the path is in the url form. This is done to mimic the full path in the file system storage driver.

read(*dag_id*, *task_id*, *execution_date*, *encoding='utf-8'*)

write(*dag_id*, *task_id*, *execution_date*, *data*, *content_type='text/plain'*, **args*, ***kwargs*)
Note that content_type is an argument not in parent method.

Parameters **content_type** (*string*) – The content-type. If set to None, it is not set.

write_from_stream(*dag_id*, *task_id*, *execution_date*, *stream*, *content_type='text/plain'*, **args*, ***kwargs*)

Parameters **content_type** (*string|None*) – pass None to not set

CHAPTER 5

fileflow.utils package

fileflow.utils.dataframe_utils module

fileflow.utils.dataframe_utils.**clean_and_write_dataframe_to_csv**(*data*, *filename*)

Cleans a dataframe of np.NaNs and saves to file via pandas.to_csv

Parameters

- **data** (pandas.DataFrame) – data to write to CSV
- **filename** (*str | None*) – Path to file to write CSV to. if None, string of data will be returned

Returns If the filename is None, returns the string of data. Otherwise returns None.

Return type str | None

fileflow.utils.dataframe_utils.**read_and_clean_csv_to_dataframe**(*filename_or_stream*, *encoding='utf-8'*)

Reads a utf-8 encoded CSV directly into a pandas dataframe as string values and scrubs np.NaN values to Python None

Parameters **filename_or_stream** (*str*) – path to CSV

Returns

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

o

s

t

u

Index

O

P

R

S

T

U

W