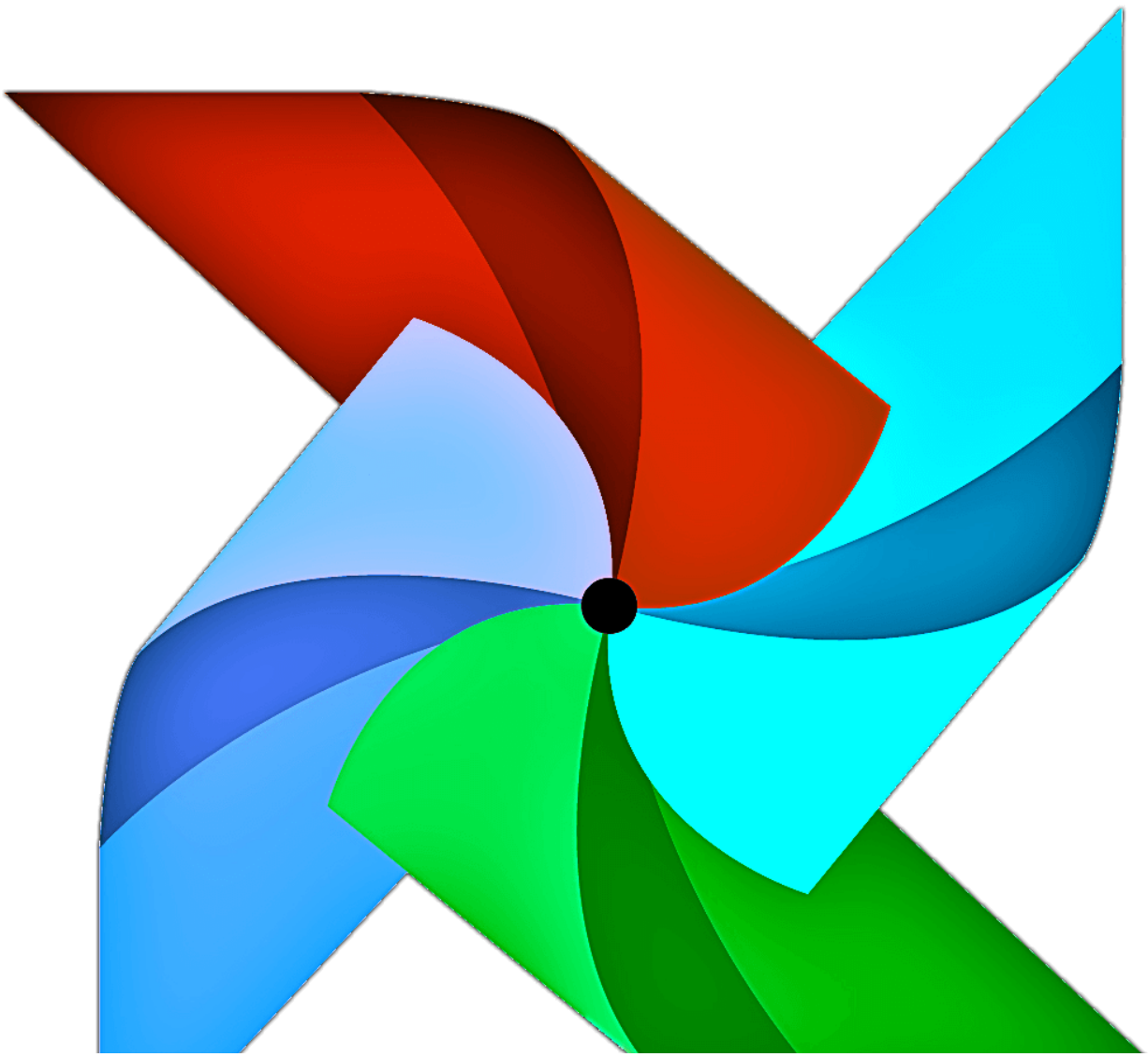
Setting Up Airflow Using Celery Executors in Docker

Craig Godden-Payne
Feb 7 · 5 min read ★



I've recently been tasked with setting up a proof of concept of Apache Airflow.

What is apache airflow?

Apache Airflow is an open-source tool for orchestrating complex computational workflows and data processing pipelines. An Airflow workflow is designed as a directed acyclic graph (DAG). That means, that when authoring a workflow, you should think how it could be divided into tasks which can be executed independently.

What did I aim to learn?

The aim of the proof of concept was to find out if it could be made:

- Highly available
- Recoverable
- Scalable
- Able to push and pull data from different AWS accounts, so the ability to run under a role

First of all, I needed to set up airflow within a containerised environment, to be able to tick off some of the points. This was relatively straightforward, although the official airflow docker image wasn't the easiest to setup, I ended up building my own dockerfile, using https://github.com/puckel/docker-airflow as a guideline.

```
FROM python:3.7-slim-stretch

ENV DEBIAN_FRONTEND noninteractive
RUN echo 1 > /dev/null
RUN apt-get update -yqq && apt-get upgrade -yqq && \
    apt-get install -yqq apt-utils && \
    apt-get install -yqq --no-install-recommends \
        freetds-dev libkrb5-dev libsasl2-dev libssl-dev libffi-dev
libpq-dev git \
        freetds-bin \
        build-essential \
        default-libmysqlclient-dev \
        cron && \
```

```
        apt-get autoremove -yqq --purge && \
        apt-get clean && \
        rm -rf \
            /var/lib/apt/lists/* \
            /tmp/* \
            /var/tmp/* \
            /usr/share/man \
            /usr/share/doc \
            /usr/share/doc-base

    RUN useradd -ms /bin/bash -d /usr/local/airflow airflow && \
        pip install -U pip setuptools wheel && \
        pip install pytz pyOpenSSL ndg-httpsclient pyasn1 awscli boto3

    RUN pip install apache-
    airflow[crypto,celery,postgres,hive,jdbc,mysql,ssh,redis,dynamodb]

    COPY airflow.cfg /usr/local/airflow/airflow/airflow.cfg
    RUN chown -R airflow: /usr/local/airflow
    COPY dags/* /usr/local/airflow/airflow/dags/

    EXPOSE 8080 5555 8793

    USER airflow
    WORKDIR /usr/local/airflow

    USER root
    COPY init.sh /usr/local/airflow/init.sh
    RUN chmod 755 /usr/local/airflow/init.sh

    USER airflow

    ENTRYPOINT ["/bin/sh", "-c", "/usr/local/airflow/init.sh"]
```

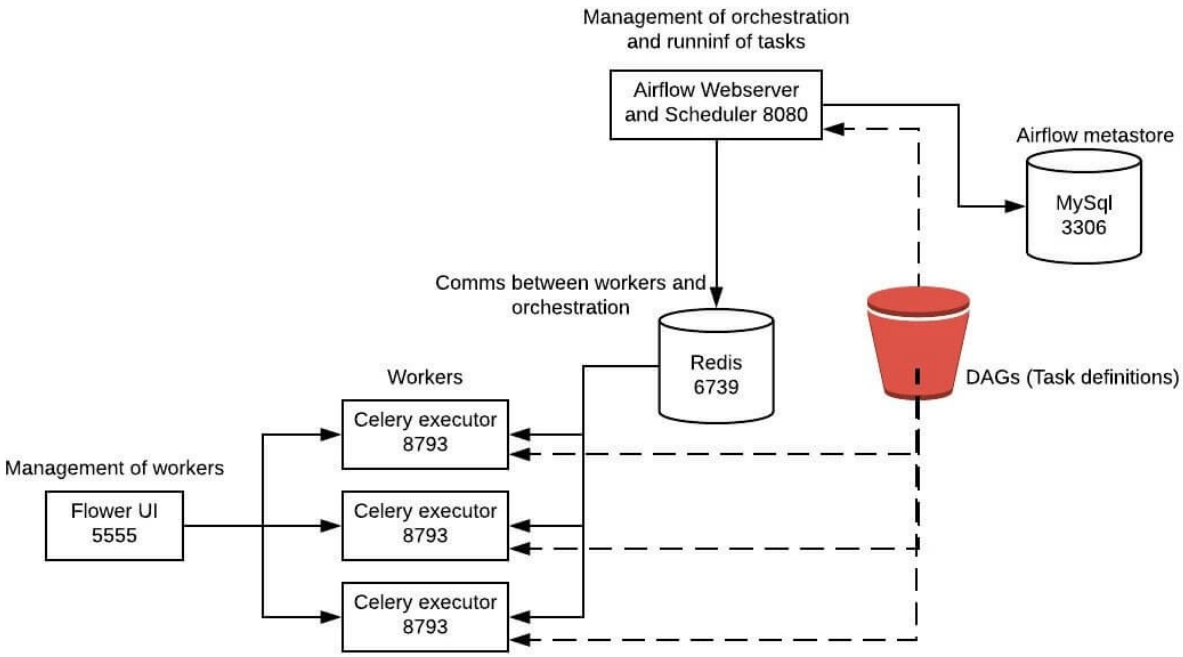
By default, it seems that airflow is built in a way to support running all its tasks running on a single instance, and I needed it to support running across multiple nodes. After a quick google, I found that it is possible to run using `celery` which is a concept familiar to python devs, but since I haven't written a load of python, it was new to me!

What is celery?

Celery is a Distributed Task Queue, an asynchronous task queue/job queue based on distributed message passing. The execution units, called tasks, are executed concurrently on a single or more worker servers using multiprocessing, eventlet, or gevent.

Within airflow, it's possible to use redis or rabbitMQ out of the box for communication between the workers and master. Since my proof of concept is in AWS, I naturally used

redis, since AWS offers redis as a service, and rabbitMQ I would have to host my own.



init script

Since airflow runs using the same base and different executables for the worker, webserver etc. I updated the init script to be able to inject in the type of process I wanted to run but maintained the same base image.

```
if [ -z "$AIRFLOW_TYPE" ]
then
  echo "Set the airflow type to either 'WEBSERVER', 'FLOWER' or
'WORKER'"
  exit 1
fi

# start airflow
if [ "$AIRFLOW_TYPE" = "WORKER" ]
then
  airflow worker

elif [ "$AIRFLOW_TYPE" = "WEBSERVER" ]
then
  airflow scheduler & airflow webserver -p8080

elif [ "$AIRFLOW_TYPE" = "FLOWER" ]
then
  airflow flower
fi
```

There idea being that there should be one or many WORKERS, but only one of WEBSERVER or FLOWER instances

Findings

It was easy to set up and as long as you use a database backend and configure remote logging, it seemed to cover all areas we needed, i.e.

- Highly available (hosted within ecs in AWS which has 99.9% SLA)
- Recoverable — immutable containers, all connecting to a database instance and sharing work via redis
- Scalable — the ability to increase the number of containers indefinitely, based on the workload needed (and should be straightforward to automate this based on stats from the running container)
- Ability to push and pull data from different AWS accounts — This part comes from the tasks themselves, and I've read a lot online about how to do this. So it is technically possible, although I haven't had a chance to prove this yet.

I then went on to build an example DAG, which would allow me to pull a CSV file from S3, convert to JSON, and then store the result within dynamodb storage.

```
import datetime
import csv
import json
import os
import string
import random
from airflow import DAG
from airflow.hooks.S3_hook import S3Hook
from airflow.contrib.hooks.aws_dynamodb_hook import AwsDynamoDBHook
from airflow.operators.python_operator import PythonOperator

s3 = S3Hook(aws_conn_id="s3_bucket")
dynamo_db = AwsDynamoDBHook(aws_conn_id="dynamo_db")
bucket_name = "airflow-poc-data"
csv_key_name = "example-data.csv"
tmp_filename = "/tmp/example-data.json"

def random_string():
    """ Generate a random string of 20 characters"""
    letters = string.ascii_lowercase
    return ''.join(random.choice(letters) for i in range(20))
```

```
def s3_csv_to_json(**kwargs):
    """ convert file from s3, from csv to json """
    print(kwargs)
    csv_content = s3.read_key(csv_key_name, bucket_name)
    print("fetched " + csv_key_name + " from " + bucket_name + ".
length is " + str(len(csv_content)))

    tmp_csv_filename = random_string() + ".tmp"
    with open(tmp_csv_filename, 'w') as csv_file:
        csv_file.write(csv_content)

    s3_tmp_filename = random_string() + ".json.tmp"
    tmp_json_filename =
convert_csv_file_to_json_file(tmp_csv_filename)
    with open(tmp_json_filename, 'rb') as file:
        s3.load_file_obj(file, s3_tmp_filename, bucket_name)

    # cleanup
    os.remove(tmp_csv_filename)
    os.remove(tmp_json_filename)
    return s3_tmp_filename

def convert_csv_file_to_json_file(csv_filename):
    tmp_json_filename = random_string() + ".tmp"
    with open(csv_filename, 'rt') as csv_file:
        with open(tmp_json_filename, 'w') as json_file:
            reader = csv.DictReader(csv_file.readlines(), ('Id',
'Name', 'Enabled', 'FavouriteNumber'))
            for rows in reader:
                json_file.write(json.dumps(rows))
    return tmp_json_filename

def save_to_dynamo(**kwargs):
    """ save json from previous task to dynamo """
    print(kwargs)
    s3_tmp_filename =
kwargs['ti'].xcom_pull(task_ids='fetch_file_from_s3')
    print("fetching previous state: key:" + str(s3_tmp_filename) + "
bucket:" + str(bucket_name))
    json_data = s3.read_key(s3_tmp_filename, bucket_name)
    print("processing json data. length is " + str(len(json_data)))
    dynamo_db.write_batch_data(json_data)
    s3.delete_objects(bucket_name, [s3_tmp_filename])

args = {
    'owner': 'Craig Godden-Payne',
    'retries': 10,
    'start_date': datetime.datetime(2019, 8, 15),
    'retry_delay': datetime.timedelta(minutes=1)
}

with DAG(dag_id='csv_to_json_example', default_args=args) as dag:
    convert_csv_to_json =
PythonOperator(task_id='fetch_file_from_s3', provide_context=True,
python_callable=s3_csv_to_json)
    save_to_dynamo = PythonOperator(task_id='save_to_dynamo',
provide_context=True, python_callable=save_to_dynamo)
```

```
convert_csv_to_json >> save_to_dynamo
```

Gotchas

There were a few gotchas I noticed, and worth a mention;

- If running locally using sqllite, you cannot run scheduler and webserver at the same time due to parallelism issues.
- When adding new DAGs, it takes a period of time before the dags appear within the webserver, it isn't instant, and it's not broke!
- The scheduler can trigger single tasks more than once over multiple workers, so it's important to make the DAGs idempotent. I didn't see this for myself during the POC, although I have read a lot about it.

Written on August 20, 2019.

Originally published on: https://craig.goddenpayne.co.uk/airflow/

Airflow Celery Python Docker