Airflow

Favorite	☐
Archived	☐
Fleeting	☐
Area/Resource	
Project	
Type	
Review Date	
Image	
URL	
Created	@February 7, 2023 4:04 PM
Updated	@February 10, 2023 1:23 PM
Root Area	
Project Area	
Updated (short)	02/10/2023
Pulls	
Resource Pulls	
Project Archived	
URL Base	
🍳 Recipe Divider	🥗🥗🥗 RECIPE BOOK PROPERTIES 🥗🥗🥗
🍳 Recipe Tags	
📚 Book Divider	📚📚📚 BOOK TRACKER PROPERTIES 📚📚📚
📚 Author	
📚 Date Started	

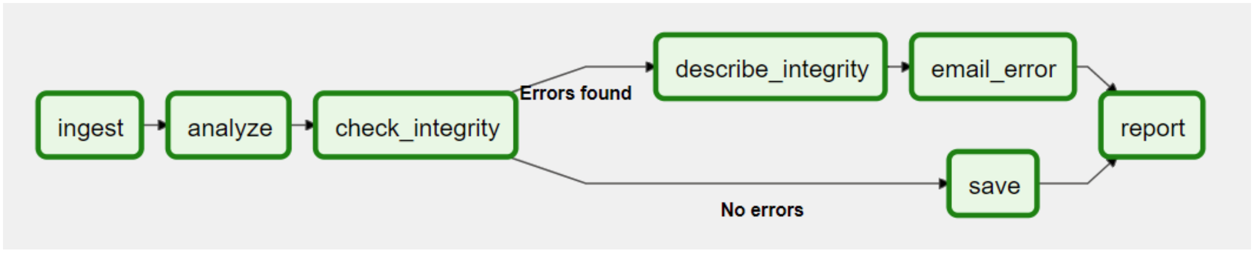
📚 Date Finished	
📚 Book Status	
📚 Rating	

Workflow orchestration

💡 When working with large complicated apps or workflow it can become complicated to manage the dependencies of different functions. Orchestration helps improve the efficiency, reliability and scalability of data pipelines by providing an intuitive way to visualize workflows, and trigger jobs based on the completion of previous necessary jobs. It can also help with monitoring and collaboration.

https://www.youtube.com/watch?v=K9AnJ9_ZAXE

Example workflow

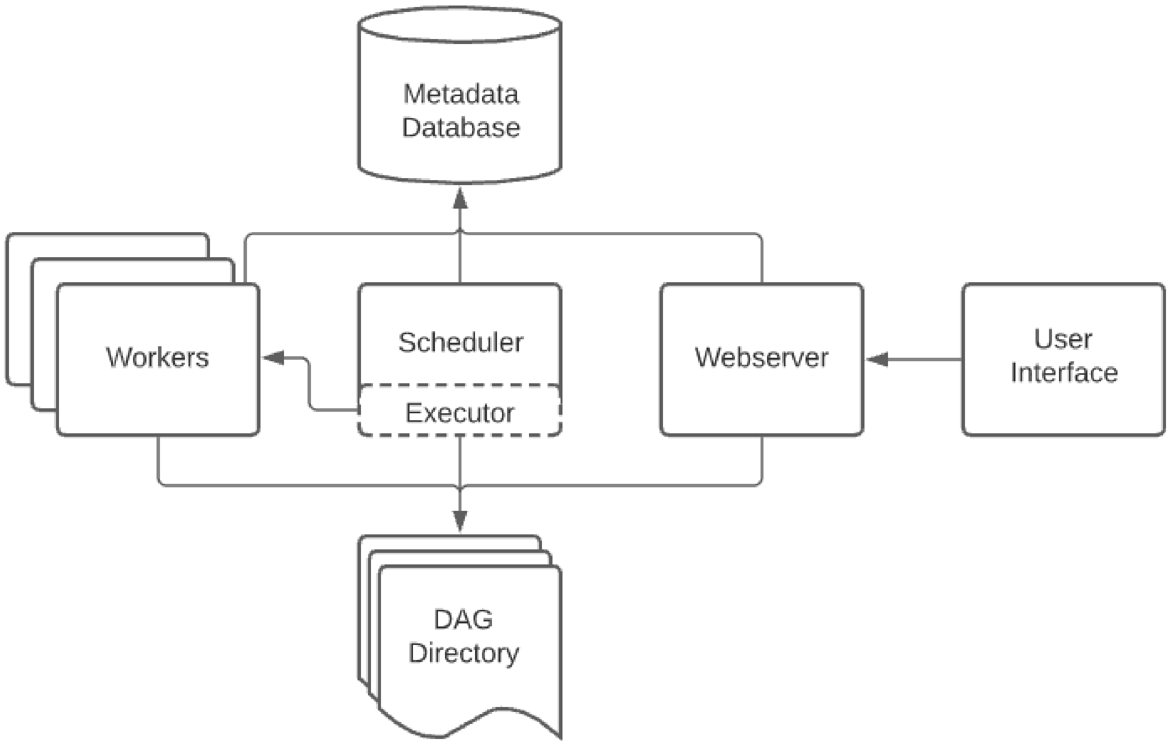


💡 Here we have an example of a workflow. As you can see this is a Directed Acyclic Graph as there is a direction in which the dependencies are moving, and no tasks has any loops which trigger itself. Each task represents a script which depends on the actions or output of the scripts before it. Each task or job has it's own set of parameters and there also may be global parameters which are the same for all jobs.

Airflow

💡 Airflow is an apache open-source framework for workflow orchestration, created by Airbnb it aimed to help solves problems relating to managing large and complex workflows.

Architecture



Web server: Graphical user interface to help handle and visualize workflows and can trigger dags, as well as view logs and status updates. It can be run on your local host or VM.

Scheduler: This component schedules and triggers tasks in the workflow. It keeps track of task dependencies and ensures the tasks are executed in the correct order.

> 💡 It is designed to run as a separate process and is written in python. It periodically checks the state of the DAGs and tasks, and triggers the execution of tasks as necessary. It also updates the status of tasks in the metadata database and maintains a record of the task execution history.

Workers: These are the actual nodes that complete the required work as defined by airflow. It can be run on a single machine or on a cluster depending on the workload.

Database: This component stores information about the workflow, including task state, execution dates and tasks logs. It is used by the scheduler and workers to coordinate activities.

Task definitions: This component defines the individual tasks that can make up the workflow. Tasks can be written in python or any other language and can perform a wide range of operations, including data processing, transfer and database operations

Direct Acyclic Graphs (DAG): This component defines the relationships between tasks in the workflow, including dependencies and execution order.

Tools

SQLite: Is used as the backend database to run locally and handle testing environment tasks, however if you try and scale up they recommend PostgreSQL for production tasks

Redis: Is used as distributed cache which makes it easier to share information across a cluster of Airflow workers. Redis can be used as a message queue. A message queue is a software component that enables the exchange of messages between different parts of a system, typically between producer and consumer applications. The message queue provides a way to decouple the producer and consumer applications, allowing them to run independently at their own pace. It handles pub/sub messaging.

CeleryExecutor: A type of executor that allows you to run tasks in parallel on a group of worker nods. It is designed to scale the number of nodes horizontally, allowing you to spread workflows across several machines. The celery executor uses a message queue such as Redis to distribute tasks to worker nodes, and to coordinate the execution of those tasks. When a task is ready to be run, the scheduler places a message in the queue, and a worker node picks up the messages, run the task and updates the metadata in the database. This allows for a highly scalable and resilient workflow environment

Setup

Pre-Requisites

1. Store your gcp service account file to google-credentials.json and store in Home directory
2. I may need to upgrade my docker compose version and set the memory for my docker engine to minimum 5GB. if not enough memory the webserver will keep restarting

Airflow setup

1. Create new sub directory called airflow in your project dir
2. Import the official image and setup the latest airflow version
    a. I used the docker-compose way

    There are two ways to setup airflow:

    1. Docker
        a. If you install with docker you should know how to use docker compose, as it relies on using several containers i.e. airflow and the metadata base. The advantage of using docker to import airflow is it suitable for production environments.
    2. Pip
        a. If you install with pip you will have to set up your own database so that airflow metadata database can be stored. You will also have to create and manage the database schema.

Pip

1. Set the airflow home and airflow user environment variables
    a. AIRFLOW_HOME is needed to specify a root directory for the airflow installation. It is used to store the configuration, logs and metadata of an airflow

instance.

    i. Setting an home variable allows you to separate the configuration and other data for different airflow instances on the same machine, allowing you to run multiple instances of airflow with different configurations

    ii. When you run airflow db init, it is in the airflow home directory where the data will be stored

2. Run airflow webserver -p 8080 to launch webserver
    a. You will run the airflow webserver on port 8080
    b. When you try to log into the server you will need a username and password, so you have to create one first
3. Run airflow scheduler to initialize the airflow scheduler

Docker

1. Download docker-compose file
2. Change celeryexecutor to localexecutor
    a. CeleryExecutor is for horizontal scaling. It is a way of running python processes in a distributed fashion. To optimize for flexibility and availability, the Celery executor works with a pool of independent workers and uses messages to delegate tasks
    b. We don't need distributed functionality since we are running everything locally
    c. We can also remove the Redis dependency and definition as well as celery worker and flower.
3. Create folders for airflow dags logs and plugins
4. Run docker compose up
    a. This will create users and the create the users and containers
5. Run docker compose up -d which will run the containers

What is airflow

Dag

💡 A dag is a collection of tasks which runs operators on them. The dependencies between several tasks creates a dag. A dependency can only go downstream and not upstream which makes it directed.

Task

💡 A task is a unit of computer which has a goal to achieve a specific thing. It uses an operator. Each task is a implementation of an operator by defining specific values for that operator. Each task has dependencies.

Operator

💡 In airflow there are many types of operators, such as BashOperator, PythonOperator and you can create your own customized ones.

Execution date

💡 Logic date and time in which the dag runs.

Task run

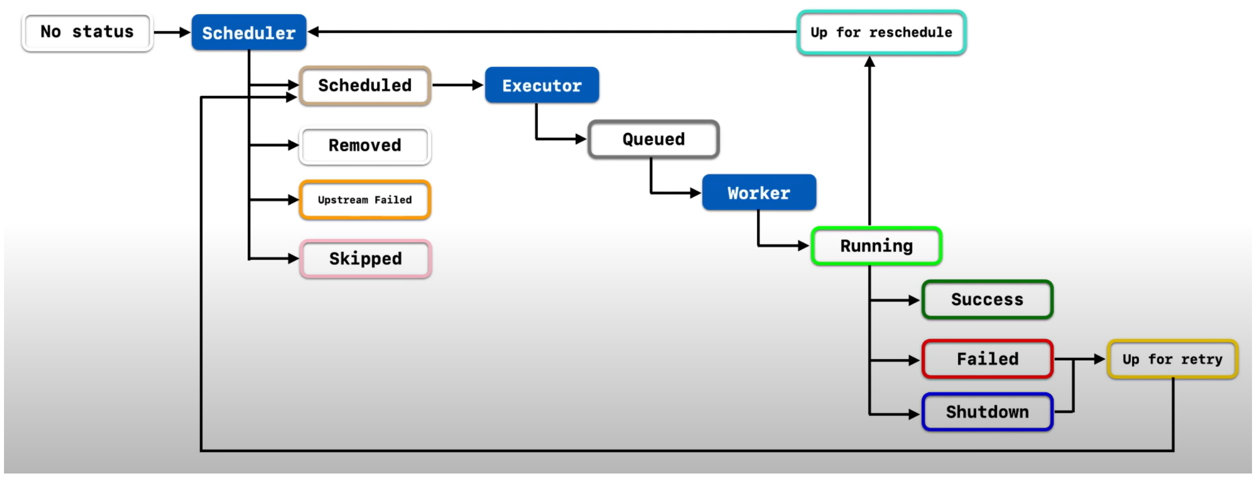
💡 A task run is a task instance running at a particular point in time.

Dag run

💡 A dag run is a dag which is triggered by it's execution date

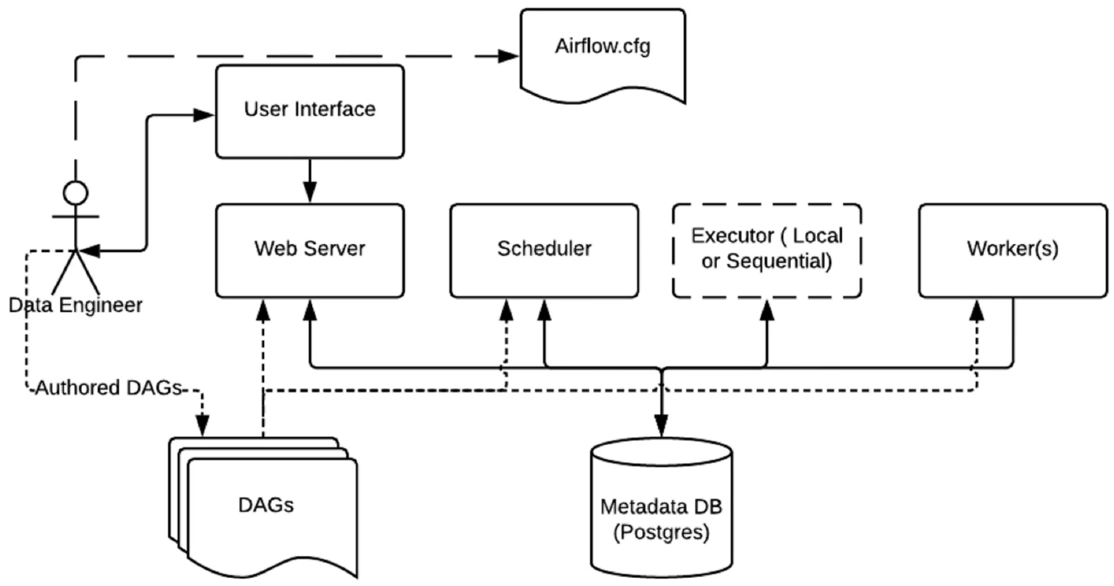
Task lifecycle

💡 Each task has a particular state showing in which stage it is running. Each task starts with no_status



up_for_reschedule: Task will be rescheduled every time interval. Check for a file in a bucket every 10 seconds is an example.

Architecture



Creating dags

💡 All airflow dags are defined in the dags folder. A dag implementation is an instantiation of the airflow class DAG.

```
from datetime import datetime
from airflow import DAG
from airflow.operator.bash import BashOperator

default_args = {
  'owner': 'alex',
  'retries': 5,
  'retry_delay': timedelta(minute=2)
}

with DAG(
  dag_id='our_first_dag',
  default_args=default_args,
  description='This is our first dag that we write',
  start_date=datetime(2021, 7, 30, 2),
  scheduler_interval='@daily'
) as dag:
  task_1 = BashOperator( # A task is an instance of an Operator
    task_id='first_task',
    bash_command='echo hello world, this is the first task'
)
```

Airflow XComs

💡 We can push information from one task to another using XComs. By default every functions return value would be pushed into Xcoms. Maximum size of XCOM is 48kb.

Airflow task API

💡 The task flow API is another way to write DAGs using the airflow API syntax methods.

Airflow backfill and catchup

💡 DAGs in airflow will catch with the scheduled intervals all the way from the declared start_date.

Schedule with con expression

💡 Airflow provides you with preset intervals. However you can provide cron expressions as time intervals for your dag to be run. You can use crontab.guru to the right cron time for you.

Install python dependencies

There are two ways to install python dependencies to your airflow docker container:

1. Image extending
    a. Only requires basic knowledge of docker

2. Image customizing

Image extending