AIrflow

Orchestration Service

JERIN SAM

Table of Contents

[Airflow 1](#_Toc181617594)

# Airflow

Airflow is an open-source platform to programmatically author, schedule and monitor workflows.

***Benefits:***

* It is a python-based framework, which can help to create dynamic tasks, workflows and branching (execute tasks based on certain conditions).
* Airflow is highly scalable.
* Its Web UI provides good visibility on the workflows.
* Extensibility: Customize Web UI, create and add new connectors etc.

***What Airflow isn’t:***

Airflow is an orchestrator and not any processing framework or Realtime streaming solution or any storage solution. Process your gigabytes of data outside of Airflow (i.e. You have a Spark cluster, you use an airflow operator to execute a Spark job, and the data is processed in Spark).

***Use case where Airflow is not the Best:***

* High Frequency, sub-minute scheduling.
* Directly processing large datasets.
* Real-time data streaming.
* Simple workflow with less dependency – CRON jobs can be used instead of Airflow

## Core Components

1. ***Web Server***: Helps to view, manage and monitor the workflows using a Web UI.
2. ***Scheduler***: Schedules tasks; It ensures tasks run at the right time and in correct order.
3. ***Meta Database***: Stores the information of the workflows, users, tasks and status of the tasks. Airflow uses Meta DB to remember which tasks have been executed, when it was executed and its status.
4. ***Trigger***: Trigger is responsible for managing deferrable tasks – tasks that wait for external events.
5. ***Executor:*** It determines how the tasks will be executed, deciding whether to run in parallel or in sequence and on which system.
6. ***Queue:*** It’s a list of tasks waiting to be executed.
7. ***Worker***: It is a process which is executing the task, it does the work defined in the tasks.

## Core Components

1. ***DAG (Directed Acyclic Graph)***: Collection of all the tasks that need to be executed, it also defines organizing these tasks in a way that reflect the dependency.

**i.e.** It helps to define the entire structure of the workflow, defining which tasks (T1, T2, T3 and T4) need to be executed before others, as shown in below snippet.

A screenshot of a computer

Description automatically generated

It supports only non-cyclic graph, won’t support cyclic nature of the graph. As can be seen in the above snippet, task T1 depends on T4 and T4 depends on T1 causing cyclic graph, which is not supported by Airflow.

Since DAG is a graph, therefore ***Task*** is also called as ***Node*** and ***Dependency*** is also called as ***Edge***.

A diagram of a diagram

Description automatically generated

1. ***Operator***: Operators allow to break down the workflows in discrete, manageable piece of work (Taks). It defines a single, idempotent, task in the DAG

Follow below link to get the list of Providers which is used to create Operators:

1. [**https://registry.astronomer.io/providers**](https://registry.astronomer.io/providers)

***\*\* idempotent*** means for a give input, output will remain same irrespective of the number of executions of the tasks.

***\*\* Providers:*** Providers can be considered as source connectors which is used to configure operators.

1. ***Task/ Task Instance***: Task is a specific instance of an Operator. When an Operator is assigned to a DAG, it becomes a Task.

Tasks are the actual unit of work that gets executed when DAG runs.

1. ***Sensors*** are specialized tasks designed to wait for a particular condition to be met before allowing the DAG to proceed.

***Sensor "sense" external events or states***, such as checking if a file exists in a specific location, confirming that data is available, or verifying that a process in an external system is complete.

Sensors are useful when your workflow depends on resources or events outside of Airflow's immediate control.

1. ***Workflow:*** A Workflow is the entire process defined by DAG, including all tasks and its dependencies, as can be seen in below snippet.

A screenshot of a computer

Description automatically generated

## Types of Airflow Architecture:

1. ***Single Node Architecture:***

In Single Node Architecture, all components of Airflow run on a single node.

As can be seen below snippet of Single Node Architecture, all components are running on a single machine.

It is used during development or running small scale workflows.

A diagram of a computer

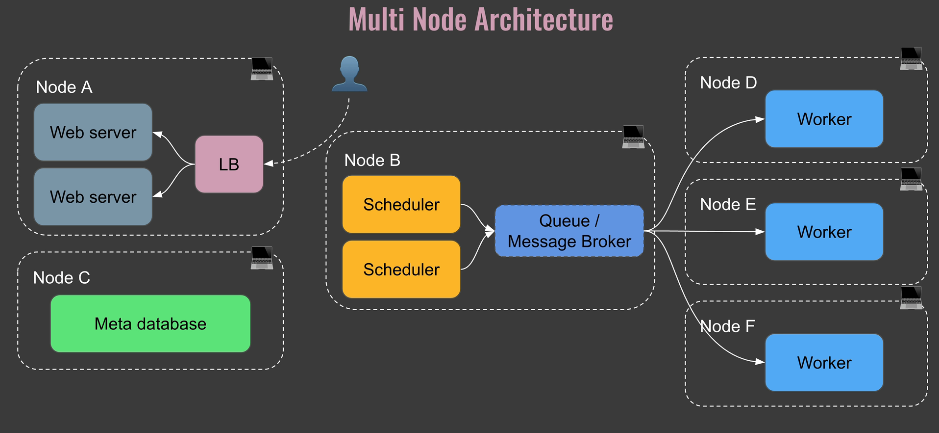
Description automatically generated

***Scheduler and Queue is generally part of Executor.***

1. ***Multi Node Architecture:***

In Multi Node Architecture, components of Airflow run on different nodes.

***Details of multi-node architecture:***



One node will have one or multiple Web Servers with Load Balancer.

* Dedicated node for Scheduler and Queue, Executor is generally part of Scheduler and for Queue Message Brokers are used like RabbitMQ.
* Each worker will have its own node.
* Meta Database will be hosted on separate node.

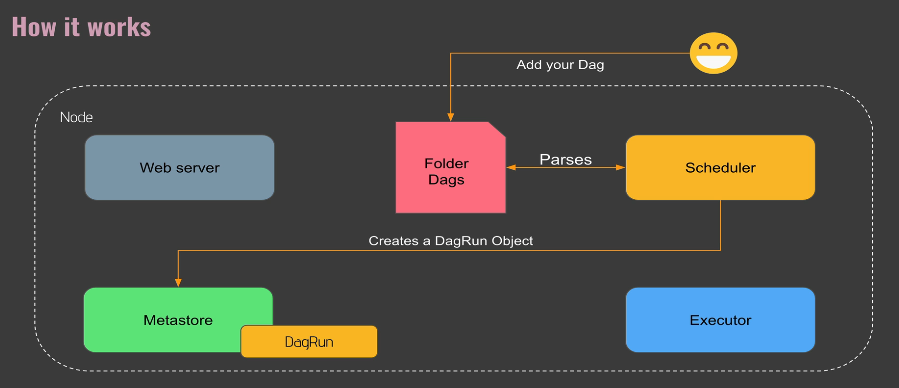
***Pros of using multi node architecture:***

* It is easy to scale.
* Provides fault tolerance (Reliability) i.e. if one of the worker nodes goes down then its tasks will be handled by other worker nodes.
* This architecture also improves the performance by distributing the tasks to multiple workers.

## How does Airflow work

**Step 1:**

User submits the python DAG file in the DAG Folder, which will be further parsed by the scheduler and creates a DAGrun object which will be stored in the Metastore and mark it as scheduled.



**Step 2:**

Once DAGrun is scheduled, now scheduler will create TaskInstance object and store it in Metastore and mark it as scheduled .

A diagram of a diagram

Description automatically generated

**Step 3:**

Once the task instance is scheduled now Scheduler will submit the TaskInstance Object to Executor, which will further decide how to execute the Tasks and it will assign a worker (not shown in the instance) to execute the same.

A diagram of a computer

Description automatically generated

**Step 4:**

Once the task is successfully executed the Worker, Executor will update the status of the tasks on Metastore.

A diagram of a computer

Description automatically generated

**Step 4:**

Once the all the tasks associated to a DAG are successfully executed then DAGRun object will be marked as Executed.

A diagram of a workflow

Description automatically generated

**Step 4:**

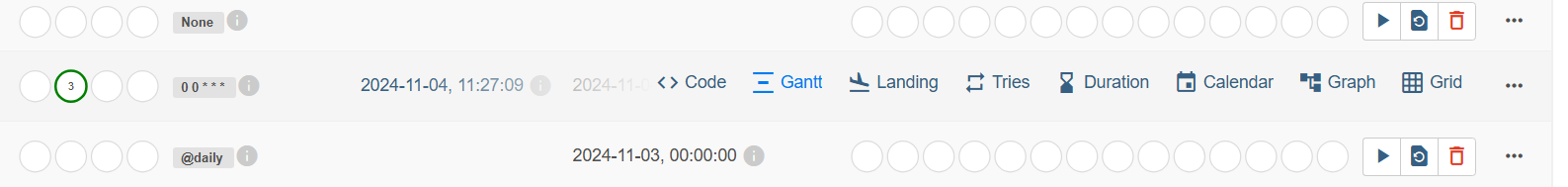
Web Server will continuously take to Metastore and get the latest update of the DAGs i.e. Scheduled DAGs, Executing DAGs/ Tasks, Failed Tasks etc..

A diagram of a dog

Description automatically generated

# Different Views to Monitor DAG

There are multiple views provided by Airflow Web UI to monitor DAGs.



***Below is the list of views:***

1. Grid View

It provides the details of the DAG and its tasks like Operator used in the tasks, execution time etc.

A screenshot of a computer

Description automatically generated

1. Graph View

Graph view will help to understand the dependencies among tasks.

A screenshot of a computer

Description automatically generated

1. Gantt View

Gantt View will help to understand the start and end time taken by the tasks to execute; this view is helpful to identify which tasks are consuming most time for the execution.

A screenshot of a computer

Description automatically generated

1. Code View

Code view helps to view the python code used to create the DAG and the associated tasks. This is helpful to check if the changes done to the Python code is applied to the DAG.

A screenshot of a computer

Description automatically generated

1. Landing View

Landing View is used to visualize the overall time taken by the DAG and its Tasks to execute over the period of time.

A screenshot of a graph

Description automatically generated

A screenshot of a computer

Description automatically generated

1. Calendar View

Calendar View helps to check the status of the DAGs.

A screenshot of a computer

Description automatically generated

# Code Data Pipeline

## Setup Environment to write Python code

Create a python script in *./airflow-working-folder/dags* folder with name “***user-processing.py***”

Use the above python file for adding DAG and Operators logics, as this folder is mounted with Airflow Docker’s DAG folder structure. Airflow’s scheduler will pick the python file from that Docker’s DAG folder location to execute.

Details of the mount point can be found in docker-compose.yaml file.

## Define a DAG

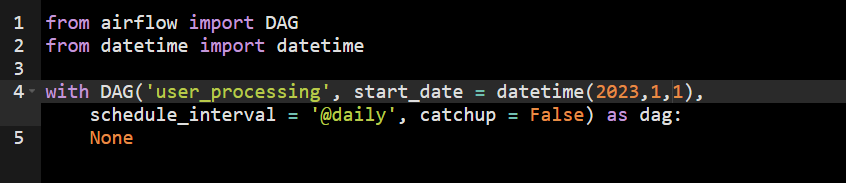
While defining DAG –

* Provide unique name to the DAG
* Assign Execution Start Date of the DAG
* Assign Schedule Interval to execute the DAG – It accepts CRON Format.

Follow below link to go through Airflow’s scheduler

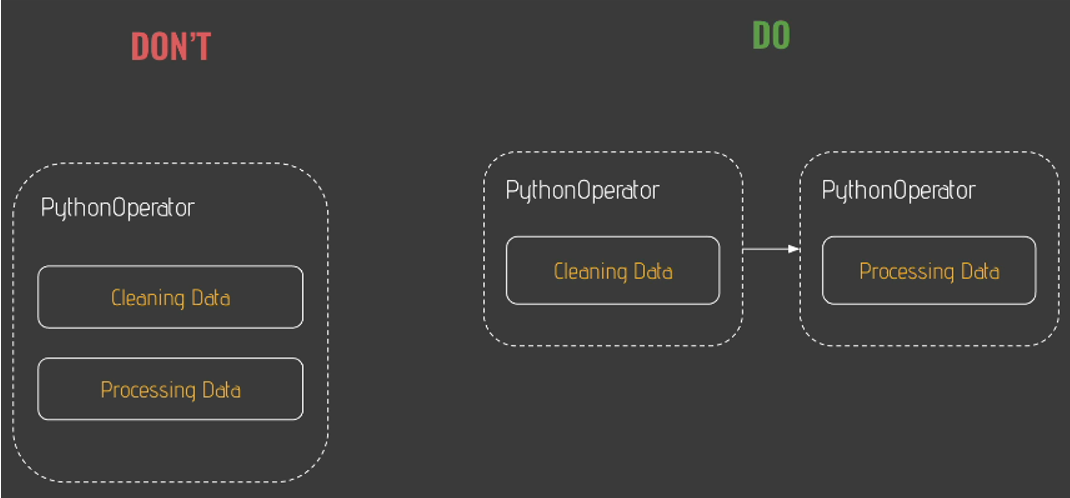
<https://airflow.apache.org/docs/apache-airflow/1.10.10/scheduler.html#dag-runs>

* If the DAG's start date is in the past, setting ***catchup=True*** will cause the DAG to run for each interval between the start date and today. To avoid this backlog of runs, set ***catchup=False***. This ensures the DAG only runs from the current date forward, preventing unnecessary historical runs.



## Define Operator

While creating operator don’t combine multiple steps in 1 operator as shown in below snippet. Break multiple processing steps into multiple Operators.



***There are 3 types of Operators:***

1. ***Action Operator***: Used to execute an action like set of code.
2. ***Transfer Operator***: Used to transfer data from point A to B.
3. ***Sensor Operator***: Wait for a condition to be fulfilled.

***Providers:***

To enable Apache Airflow to interact with various data sources, specific providers for each source needs to be installed.

***\*\**** *Providers can be considered as source connectors which is used to configure operators.*

***Install core Airflow***: pip install apache-airflow

***Install Snowflake Provider***: pip install apache-airflow-snowflake

A screenshot of a computer

Description automatically generated

*While configuring an Operator, a connection needs to be established in Airflow’s Web UI and connection\_id used in the connection will be referenced while configuring the Operator in Python Script.*

***Operator Python Script Details:***

In this example, Postgres Operator is used to create a table in Postgres DB.

Operator requires certain parameters, one of such parameters is connection Id, to get the connection id, set up Postgres connection in Airflow Web UI and use the connection name provided while setting up the connection in the python script.

*Airflow Web UI Setup Postgres Connection using below details:*

* *Host: postgres*
* *DB: airflow*
* *Login: airflow*
* *Password: airflow*
* *Port: 5432*

A screenshot of a computer

Description automatically generated

*Python Code Snippet:*

A screen shot of a computer

Description automatically generated

To understand the parameters used in the operator use the documentation associated to that operator. Use the link below to access the documentation and navigate to Python API section.

* All the providers can be found at

<https://airflow.apache.org/docs/>

* Postgres provider’s operator documentation:

<https://airflow.apache.org/docs/apache-airflow-providers-postgres/5.8.0/_api/airflow/providers/postgres/operators/postgres/index.html>

A screenshot of a computer

Description automatically generated

***Test the Task:***

* Go to the folder path where docker-compose.yaml file is kept and execute below script to list all active containers.

|  |
| --- |
| ***docker-compose ps*** |

* Open the Airflow docker shell from the Host Machine, using the below script.

***# code snippet****: docker exec -it <container\_name> /bin/bash*

|  |
| --- |
| ***docker exec -it 2-airflow-docker-configure-airflow-scheduler-1 /bin/bash*** |

* Test the shell by using below script, this will help to check whether airflow is accessible or not.

|  |
| --- |
| ***airflow -h*** |

* Test Airflow tasks using below script:

***# code snippet****: airflow tasks test <dag\_id> <task\_id> <past-date>*

|  |
| --- |
| ***airflow tasks test user\_processing create\_table 2022-01-01*** |

* After executing above script, if Success message comes then Tasks are successfully created.

## Define Sensors

***Sensors*** has 2 main parameters, to control the frequency and duration of checks for a condition -

1. ***poke\_interval*** 
   * The amount of time (in seconds) between each check for the condition.
   * If *poke\_interval=60*, the sensor will check every 60 seconds to see if the target condition has been satisfied.
   * Default value is 60 seconds.
2. ***timeout*** 
   * The maximum amount of time (in seconds) the sensor will wait for the condition to be met.
   * If *timeout=600*, the sensor will wait up to 10 minutes. If the condition is not met within that time, the sensor fails, and the task is marked as failed.
   * Default value is 7 days.

*While configuring an Operator, a connection needs to be established in Airflow’s Web UI and connection\_id used in the connection will be referenced while configuring the Operator in Python Script.*

***Sensors Python Script Details:***

In this exercise, a sensor task is created to check whether API is available or not.

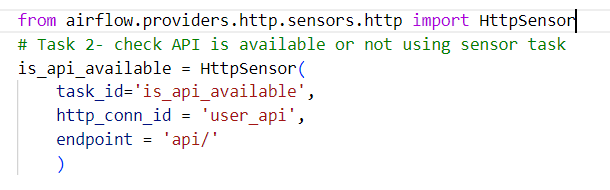
*Airflow Web UI Setup HTTP Connection using below details:*

* *Connection\_id: user\_api*
* *Connection Type: HTTP*
* *Host:* [*https://randomuser.me/*](https://randomuser.me/)

A screenshot of a computer

Description automatically generated

*Python Code Snippet:*



## Additional Operators

* Create an Operator to extract users from the API

A screen shot of a computer code

Description automatically generated

# Appendix

## Important Links

* Astronomer Providers

<https://registry.astronomer.io/providers>

* Airflow Providers

<https://airflow.apache.org/docs/>

* Airflow Schedules and Triggers

<https://airflow.apache.org/docs/apache-airflow/1.10.10/scheduler.html#dag-runs>