AIrflow

Orchestration Service

JERIN SAM

Table of Contents

[Airflow 3](#_Toc182234898)

[Core Components 4](#_Toc182234899)

[Core Components 4](#_Toc182234900)

[Types of Airflow Architecture: 6](#_Toc182234901)

[How does Airflow work 8](#_Toc182234902)

[Execution Steps 8](#_Toc182234903)

[Important Points to Remember 11](#_Toc182234904)

[Different Views to Monitor DAG 12](#_Toc182234905)

[Code Data Pipeline 15](#_Toc182234906)

[Setup Environment to write Python code 16](#_Toc182234907)

[Define a DAG 16](#_Toc182234908)

[DAG 16](#_Toc182234909)

[DAG Python Script Details 18](#_Toc182234910)

[Define Operator 18](#_Toc182234911)

[Operator 18](#_Toc182234912)

[Providers 18](#_Toc182234913)

[Operator Python Script Details 19](#_Toc182234914)

[Test the Task 21](#_Toc182234915)

[Define Sensors 22](#_Toc182234916)

[Sensors 22](#_Toc182234917)

[Sensors Python Script Details 22](#_Toc182234918)

[Additional Operators 23](#_Toc182234919)

[Define Hook 24](#_Toc182234920)

[Hook 24](#_Toc182234921)

[Hook Python Script Details 25](#_Toc182234922)

[Define Dependency 26](#_Toc182234923)

[Dependency 26](#_Toc182234924)

[Dependency Python Script Details 26](#_Toc182234925)

[DAG and Tasks - Python Solution 27](#_Toc182234926)

[Test DAG and Its Tasks 27](#_Toc182234927)

[Scheduling DAGs using Datasets / Data Awareness 28](#_Toc182234928)

[Dataset Schedule/ Trigger 28](#_Toc182234929)

[Important concepts while implementing Dataset Triggers 29](#_Toc182234930)

[Python Script Snippet 30](#_Toc182234931)

[Dataset - Python Solution 31](#_Toc182234932)

[Executor 31](#_Toc182234933)

[Executor 31](#_Toc182234934)

[Airflow Configuration File: 33](#_Toc182234935)

[Executor’s Configuration 35](#_Toc182234936)

[Monitor Tasks with Flower 37](#_Toc182234937)

[CeleryExecutor – Python Implementation 38](#_Toc182234938)

[Advanced Concepts 39](#_Toc182234939)

[Group Tasks 39](#_Toc182234940)

[SubDag 39](#_Toc182234941)

[SubDag – Python Implementation 39](#_Toc182234942)

[TaskGroups 40](#_Toc182234943)

[TaskGroups – Python Implementation 40](#_Toc182234944)

[Share Data Between Tasks 41](#_Toc182234945)

[XComs 41](#_Toc182234946)

[Branch Operator 42](#_Toc182234947)

[Trigger Rules 42](#_Toc182234948)

[XComs, Branch Operator & Trigger Rules– Python Implementation 44](#_Toc182234949)

[Airflow Context 46](#_Toc182234950)

[Python Script 46](#_Toc182234951)

[Params 47](#_Toc182234952)

[Python Script 47](#_Toc182234953)

[Variables 48](#_Toc182234954)

[Python Script 49](#_Toc182234955)

[Dynamic DAGs – Jinja Template 50](#_Toc182234956)

[Python Script to generate dynamic DAGs 51](#_Toc182234957)

[Appendix 53](#_Toc182234958)

[Important Links 53](#_Toc182234959)

[Remove Example DAGs from Airflow 54](#_Toc182234960)

# 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

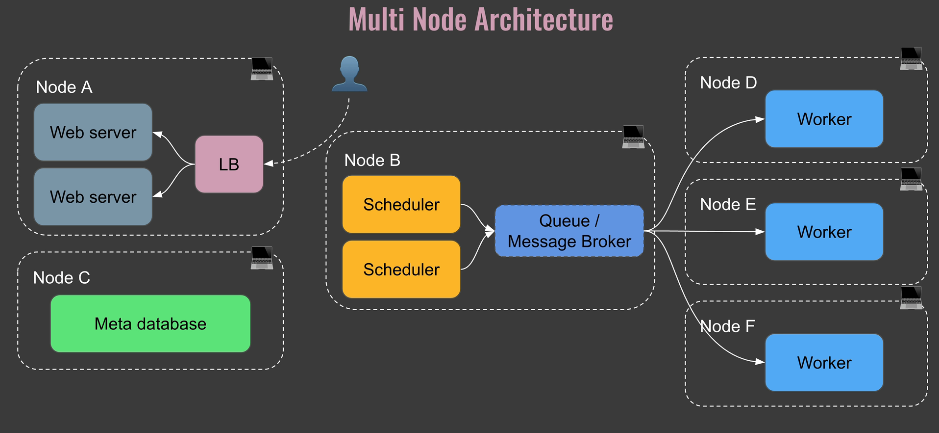
Description automatically generated

***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:***

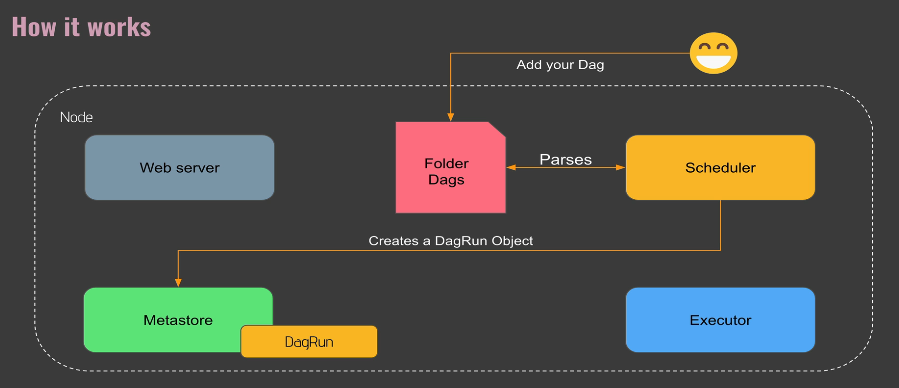
* It is easy to scale.
* Provides fault tolerance (Reliability) i.e. if one of the worker nodes goes down then its tasks will be handled by other worker nodes.
* This architecture also improves the performance by distributing the tasks to multiple workers.

## How does Airflow work

### Execution Steps

**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

### Important Points to Remember

Whenever a DAG is executed, a DAGRun object is created, which has timestamp information ***data\_interval\_start*** and ***data\_interval\_end.***

***data\_interval\_start:***

* It marks the beginning of the data interval, or the range of data that a task is responsible for processing.
* It essentially marking the start of the ***logical interval*** *also known as* ***logical data***.
* ***execution date*** was the older term for ***logical date*** - meaning they all represent the start of the data interval, not the actual time the DAG is executed.

***data\_interval\_end:***

* It marks the end of the data interval, or the end of the data collection period for a DAG run.

A diagram of a diagram

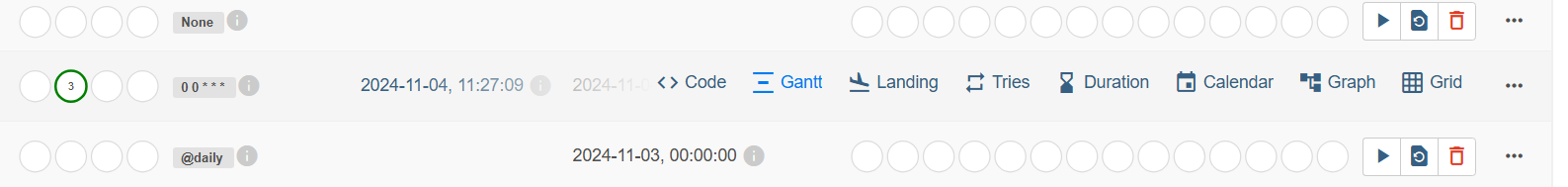
Description automatically generated

In the above example shown in the snippet, if the DAG is deployed on 2022-01-01 and scheduled to execute at 2022-01-02 midnight then –

1. During 1st scheduled execution - ***data\_interval\_start*** will be ***2022-01-01*** and ***data\_interval\_end*** will be ***2022-01-02.***
2. During 2nd scheduled execution - ***data\_interval\_start*** will be ***2022-01-02*** and ***data\_interval\_end*** will be ***2022-01-03.***

# 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

In this exercise, A DAG will be created along with following tasks –

1. A task to create a table in Postgres.
2. A task to check the availability of API.
3. Extract data from API.
4. Push data to Postgres Table.

Dependencies of the tasks will be defined after the creation of tasks.

## 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

### DAG

While defining DAG, following parameters need to be passed –

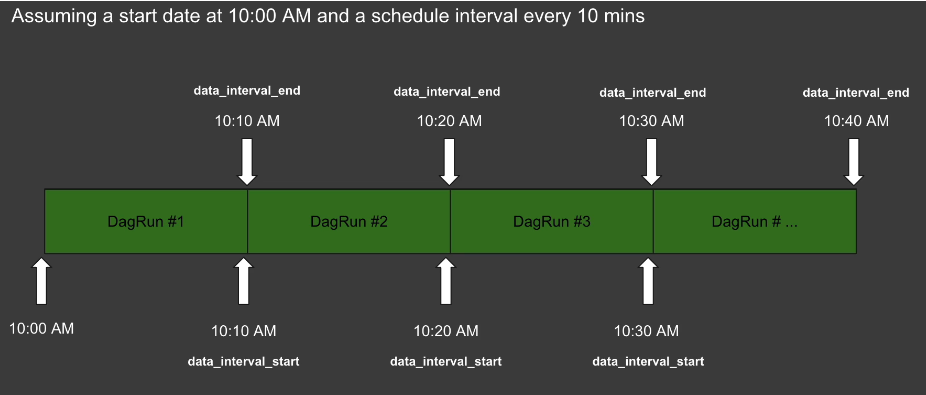
* ***dag\_id***: Provide unique name to the DAG.
* ***start\_date:*** Assign execution Start Date of the DAG.
* ***schedule\_interval:*** Assign Schedule Interval to execute the DAG at a set frequency – It accepts CRON Format.

There is another parameter called ***timetable*** which can be used to schedule the DAG using Airflow’s *timetable* *class*.

Follow below link to go through Airflow’s scheduler

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

A DAG is triggered after the ***start\_date/last\_run+ schedule\_interval***



In the above example, let’s say DAG *start\_date is 2022-01-01* and is scheduled to execute every 10 mins.

Initially Scheduler will wait for 10 mins before the actual execution so, *data\_interval\_start = 2022-01-01* 10 AM and after 10 mins i.e. at 10:10 AM DAG is executed, so *data\_interval\_end = 10:10 AM*

Now during 2nd execution schedule, *data\_interval\_start =* 10:10 AM and after 10 mins i.e. at 10:20 AM DAG is executed again, so *data\_interval\_end = 10:20 AM*

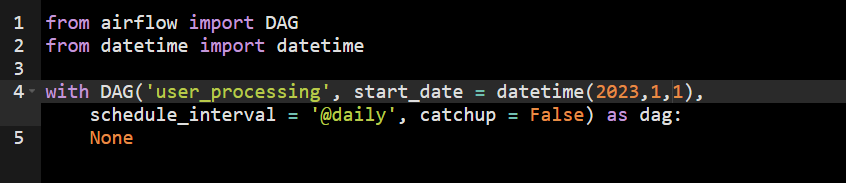
* ***catchup:*** This is used to backfill the data. For Example, If the DAG's ***start\_date*** is in the past, setting ***catchup=True*** will cause the DAG to run for each interval between the start date and today.

To avoid this backlog of runs, set ***catchup=False***. This ensures the DAG only runs from the current date forward, preventing unnecessary historical runs.

A screenshot of a video game

Description automatically generated

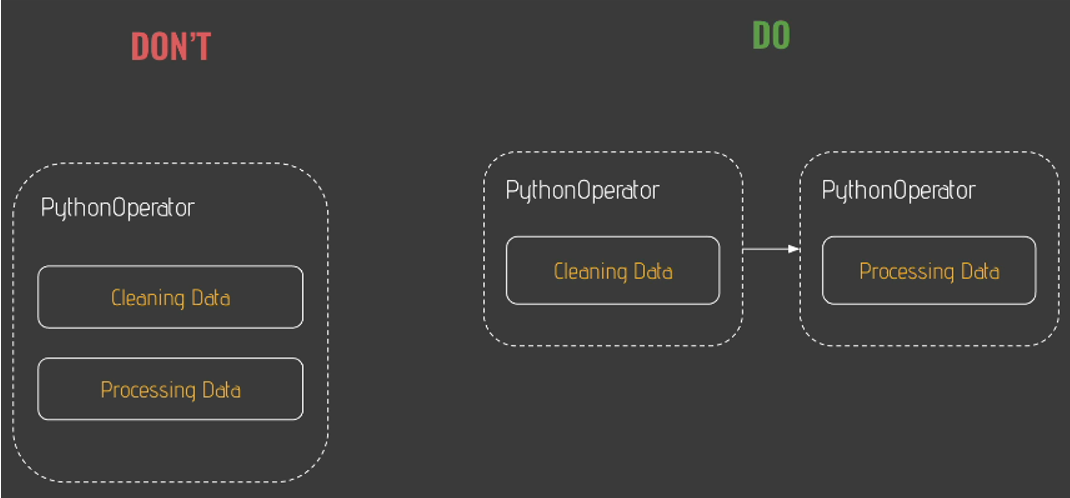
### DAG Python Script Details



## Define Operator

### 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

***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.

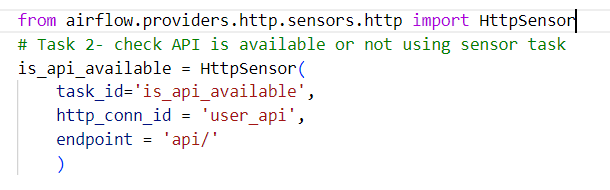
#### Airflow Web UI Setup HTTP Connection using below details:

* *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

* Process data, which is extracted from API, PythonOperator is used to manage it.

A screenshot of a computer code

Description automatically generated

## Define Hook

### Hook

***Hook*** is an abstract layer for interacting with external systems and services.

Hooks provide a way for Airflow tasks to connect to, authenticate, and interact with different types of data sources, such as databases, cloud storage, APIs, and message brokers.

Behind the scenes Operators talk to Data Source using Hooks, as shown in below snippet.

A screenshot of a diagram

Description automatically generated

*Hook provides access to certain functions, which is not accessible via Operators. In the below snippet copy\_expert function in PostgresHook is an example of such function which is not accessible via PostgresOperator*

### Hook Python Script Details

A screenshot of a computer

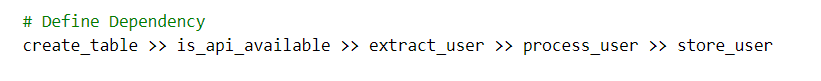
Description automatically generated

## Define Dependency

### Dependency

Dependency is defined using ***>>***, **<<** operator and ***set\_downstream*** and ***set\_upstream*** methods can also be used. This helps to define which task is dependent upon which another task.

### Dependency Python Script Details

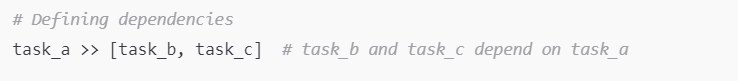


*Web UI graph view snippet to view dependency between tasks*

A screen shot of a computer

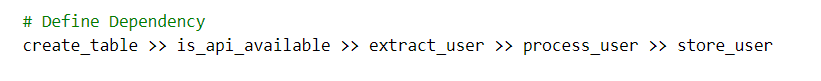
Description automatically generated

***Example 1:***



The above dependency denotes, ***task\_b and task\_c depend on task\_a***

***Example 2:***



*The above dependency can also be written as follows:*

A screenshot of a computer program

Description automatically generated

## DAG and Tasks - Python Solution

Follow below path for the python script implemented-

* ***./main/airflow-working-folder/dags/user-processing.py***

## Test DAG and Its Tasks

* 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 Postgres 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-postgres-1/bin/bash*** |

* Execute below script to open Postgres Server with User Airflow .

|  |
| --- |
| ***psql -UAirflow*** |

* Execute select statement to check data in the Table

|  |
| --- |
| ***select \* from users;*** |

* If data is populated after executing select statement, then DAG has successfully executed.

# Scheduling DAGs using Datasets / Data Awareness

## Dataset Schedule/ Trigger

***Note: schedule\_interval*** *and* ***timetable parameters*** *will be* ***deprecated*** *and is* ***replaced*** *with parameter* ***“schedule”***

***Data Awareness*** refers to the concept of making Airflow aware of data dependencies and availability for scheduling and executing workflows.

***Data-aware scheduling*** is handled by using ***Dataset***.

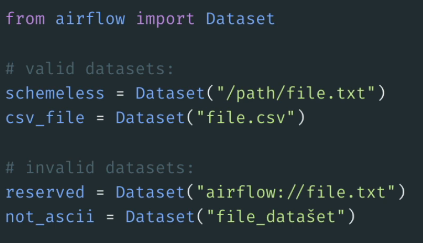
In addition to using CRON expressions to schedule DAGs, Airflow offers an alternative method called ***Dataset*** Scheduling.

This approach enables a DAG to be triggered automatically after a dependent DAG successfully produces or updates a specific ***dataset***.

***Airflow dataset is a logical grouping of data***. **Upstream producer tasks can update datasets, and dataset updates contribute to scheduling downstream consumer DAGs**.

There are 2 important properties to a Dataset:

* ***URI –*** 
  + Unique Identifier of the dataset
  + Path to the dataset
  + Supports only ASCII Characters
  + Case Sensitive
  + URI schema cannot be airflow



* ***extra –*** 
  + To provide more details to the Dataset Extra is used.

A computer screen with text

Description automatically generated

## Important concepts while implementing Dataset Triggers

* ***@task*** decorator in Airflow is specifically designed for creating tasks with the ***PythonOperator*** or similar operators that use Python functions directly. This decorator is part of the **Task Flow API.**
* ***Inlets and Outlets in Apache Airflow Tasks***
  + ***Outlets*** –

A task parameter that contains the list of ***datasets*** a specific task produces updates to, as soon as it completes successfully.

All outlets of a task are shown in the DAG graph in the Airflow UI, as well as reflected in the dependency graph of the Datasets tab as soon as the DAG code is parsed, i.e. independently of whether or not any dataset events have occurred.

Note that Airflow is not yet aware of the underlying data. It is up to you to determine which tasks should be considered producer tasks for a dataset. As long as a task has an outlet dataset, Airflow considers it a producer task even if that task doesn't operate on the referenced dataset.

* + ***Inlets:***

A task parameter that contains the list of datasets a specific task has access to, typically to access ***extra*** information from related dataset events. Defining inlets for a task does not affect the schedule of the DAG containing the task and the relationship is not reflected in the Airflow UI.

## Python Script Snippet

*Producer.py*

A screenshot of a computer program

Description automatically generated

*consumer.py*

A screen shot of a computer code

Description automatically generated

*DAG in Web UI showing schedule as Dataset for Consumer*

A screenshot of a computer

Description automatically generated

## Dataset - Python Solution

Follow below path for the python script implemented-

* ***/main/airflow-working-folder/dags/producer.py***
* ***/main/airflow-working-folder/dags/consumer.py***

In this implementation, consumer DAG is dependent on producer DAG, and consumer DAG will be triggered when producer DAG writes in the dataset file.

# Executor

## Executor

Executor does not execute a task but decides “how to run a task”. There are 3 types of executors –

1. ***SequentialExecutor***

Only 1 task will be executed at a time. It uses SQLLite DB which can handle only 1 writer at a time.

A diagram of a computer program

Description automatically generated

1. ***LocalExecutor***

This can execute multiple tasks at the same time, it uses Postgres to handle the executions. It can not be scaled horizontally.

A diagram of a software program

Description automatically generated with medium confidence

1. ***CeleryExecutor***

This can execute multiple tasks at the same time, it uses Postgres to handle the executions. It can be scaled horizontally.

This Executor has additional components like ***Queue*** and ***Workers.*** Tasks will be queued for the execution in Queue and once Workers are available to execute then Tasks will be moved from queue to the Workers for the execution.

Generally ***Redis*** is used as Queue.

A screenshot of a computer

Description automatically generated

## Airflow Configuration File:

Config file of Airflow can be found in scheduler docker container’s file location ***/opt/airflow/airflow.cfg***

Below Docker Copy script to copy the Config file from the docker to Host machine.

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

***# code snippet****: docker cp <container\_name>:<file\_path> <destination\_host\_path>*

|  |
| --- |
| ***docker cp 2-airflow-docker-configure-airflow-scheduler-1:/opt/airflow/airflow.cfg .*** |

config file is divided into sections, which can be identified using []. In below snippet, ***[core]*** denotes core config of the Airflow. Executor config is denoted in the [core] section itself.

Since, Docker container is being used therefore these config properties can be changed using Environmental Variables, In the below snippet, as can be seen in the comment variable “*AIRFLOW\_\_CORE\_\_EXECUTOR*” is used to change the executor property present in core section of the config file.

*Config [core] section snippet:*

A screenshot of a computer

Description automatically generated

*Config Executor propery and Environmental Variable snippet:*

A screenshot of a computer

Description automatically generated

Environmental Variable can be changes using Docker-Compose.yaml file, as shown in the below snippet. ***CeleryExecutor*** is used in the docker file.

A screenshot of a computer code

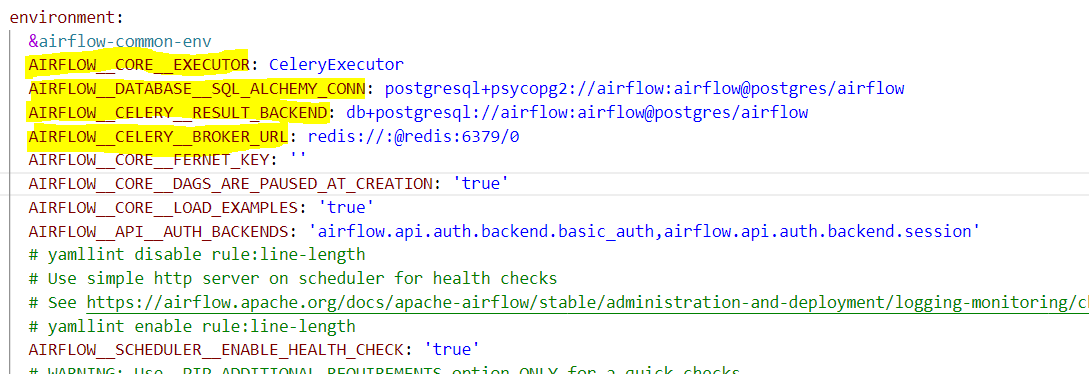
Description automatically generated

## Executor’s Configuration

For each type of Executor, following config property needs to be changed:

1. SequentialExecutor
   * ***executor = SequentialExecutor***
2. LocalExecutor
   * ***executor = SequentialExecutor***
   * **sql\_alchemy\_conn=postgresql+psycopg2://<user>:<password>@<host>/<db**>
3. CeleryExecutor
   * ***executor=SequentialExecutor***
   * ***sql\_alchemy\_conn=postgresql+psycopg2://<user>:<password>@<host>/<db> :*** This is used as Metadata DB
   * ***celery\_result\_backend= postgresql+psycopg2://<user>:<password>@<host>/<db> :*** This is used to store state of the tasks, task results etc.
   * ***celery\_broker\_url=redis://:@redis:6379/0 :*** This is used for redis queue

The configuration property mentioned above, can be changed in Airflow Docker Containeres using Environmental Variables mentioned in Docker Compose file.



Docker Compose file also contains the services details which will be used by Airflow like Redis, Postgres, Airflow Workers etc.

*Docker Compose Snippet showing Redis and Postgres Services:*

A screenshot of a computer program

Description automatically generated

*Docker Compose Snippet showing Airflow Worker:*

A screenshot of a computer

Description automatically generated

*Note:* ***celery worker*** *command is used to assign a node/ system as worker to the Executor.*

Currently in the Docker Compose file only 1 worker is used in the executor, if additional worker is required then existing worker code can be duplicated and change the service name. Ref. below snippet.

A screenshot of a computer code

Description automatically generated

In the above snippet, the command ***“celery worker -q high\_cpu”*** means –

1. **“*celery worker***”: This command will add the node/ system to the Executor as a worker node
2. **“-q high\_cpu”:** This command will create a new queue called high\_cpu and all the tasks in this queue will be executed by this worker.

## Monitor Tasks with Flower

***Flower*** is a web-based tool for monitoring and managing Celery clusters.

Execute below Docker Compose command to run Flower.

|  |
| --- |
| ***docker-compose down***  ***docker-compose --profile flower up -d*** |

After executing the above script, navigate to [http://*localhost:5555*](http://localhost:5555)to access Flower Web UI.

A screenshot of a computer

Description automatically generated

## CeleryExecutor – Python Implementation

Python Script can be found at –

* **/main/ airflow-working-folder/dags/*parallel\_dag.py***

In this exercise, a DAG with 5 BashOperator sleep tasks are created, to check –

1. Multiple tasks getting executed at once.

*Airflow Web UI - Tasks load\_b and load\_a are running parallel:*

A screenshot of a computer

Description automatically generated

1. Send a task to a specific queue i.e. high\_cpu queue

*Code Snippet – Adding Queue in Tasks:*

A screen shot of a computer program

Description automatically generated

*Flower UI – Showing worker associated with high\_cpu queue handling a task:*

A screenshot of a computer

Description automatically generated

# Advanced Concepts

## Group Tasks

### SubDag

***SubDag*** is a type of DAG that is embedded within a parent DAG as a task.

It allows to modularize complex workflows by splitting a large DAG into smaller, more manageable DAGs, helping to improve readability and simplify maintenance.

***SubDagOperator*** is used to implement ***SubDag***.

*NOTE: The main function of SubDag was to group the tasks together so a new feature is introduced therefore SubDag is deprecated in Airflow 2.0 and instead the new feature TaskGroups is introduced.*

### SubDag – Python Implementation

* Follow below path for the python script -
  + *SubDag Function 1****: /main/airflow-working-folder/dags/* *subdags/* *subdag\_downloads.py***
  + *SubDag Function 2****: /main/airflow-working-folder/dags/*** ***subdags/*** ***subdag\_transforms.py***
  + *Main Dag****: /main/airflow-working-folder/dags/*** ***parent\_dag\_for\_subdags.py***
* A Folder is created to store SubDAGs, and a function is created where DAG will be defined and will be later called by the main DAG implementation.

*SubDag - Downloads*

A screenshot of a computer

Description automatically generated

* In the main DAG implementation, the above SubDag functions will be called using *SubDagOperator*.

*Main DAG implementation*

A screenshot of a computer code

Description automatically generated

### TaskGroups

*NOTE: The main function of SubDag was to group the tasks together so a new feature is introduced therefore SubDag is deprecated in Airflow 2.0 and instead the new feature TaskGroups is introduced.*

***TaskGroups*** groups related tasks within a DAG without creating a separate ***SubDag***.

### TaskGroups – Python Implementation

* Follow below path for the python script -
  + *SubDag Function 1****: /main/airflow-working-folder/dags/* *taskgroup/* *taskgroup\_downloads.py***
  + *SubDag Function 2****: /main/airflow-working-folder/dags/*** ***taskgroup/*** ***taskgroup\_transforms.py***
  + *Main Dag****: /main/airflow-working-folder/dags/*** ***parent\_dag\_for\_*** ***taskgroups.py***
* A Folder is created to store taskgroups, and a function is created where TaskGroup will be defined with all the tasks and will be later called by the main DAG implementation.

*TaskGroup – Downloads*

A screenshot of a computer

Description automatically generated

* In the main DAG implementation, the above TaskGroup functions will be called .

*Main DAG implementation*

A screenshot of a computer

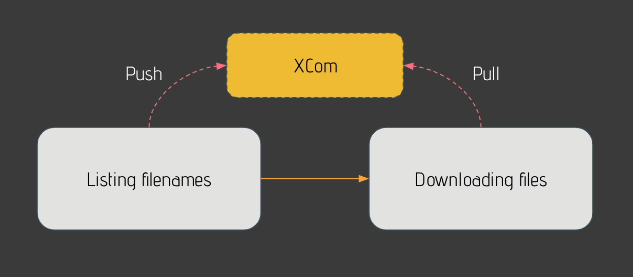
Description automatically generated

## Share Data Between Tasks

### XComs

To share data between 2 tasks, XComs can be used. XComs do cross communication between tasks and allows to exchange small amounts of data.

* If SQLLite is used, then upto 2 GB of data can be stored in xComs.
* If Postgres is used, then upto 1 GB of data can be stored in xComs.
* If MySQL is used, then upto 64 KB of data can be stored in xComs



### Branch Operator

Branch Operator is used to execute a task based on a condition. In the below snippet, based on a specific condition, either Task A 🡪 Task B will be executed or Task C 🡪 Task D.

Its dev syntax is similar to the implementation of other operators like PythonOperator.

A diagram of a task

Description automatically generated

### Trigger Rules

In the below scenario, based on a branch operator condition Task t2 is executed and Task t3 is skipped, however Task t4 was expected to be executed and since its dependent on task t2 and t3 and since t3 is skipped therefore Airflow skipped t4 as well.

A diagram of a branch

Description automatically generated

To handle such scenarios, ***Trigger Rules*** are used.

***Trigger Rules*** define the conditions under which a task should run based on the state of its upstream tasks.

This allows for complex dependencies and helps control task execution in scenarios where some upstream tasks may fail or skip.

Different types of Trigger Rules –

* ***all\_success***: (default) all parents/ upstream tasks have succeeded.

*In below snippet, for task t4 since one upstream task, t3 is skipped therefore t4 is also skipped*.

A screenshot of a computer screen

Description automatically generated

* ***all\_failed***: all parents/ upstream tasks are in a failed or upstream\_failed state
* ***all\_done***: all parents/ upstream tasks are done with their execution, regardless of their states.
* ***all\_skipped***: all parents/ upstream tasks are skipped.
* ***one\_failed***: fires as soon as at least one parent/ upstream task has failed, it does not wait for all parents to be done
* ***one\_success***: fires as soon as at least one parent/ upstream tasks succeed, it does not wait for all parents to be done
* ***one\_done***: fires as soon as at least one parent/ upstream task has either succeeded or failed.
* ***none\_failed***: all parents have not failed (failed or upstream\_failed) i.e. all parents/ upstream tasks have succeeded or been skipped
* ***none\_skipped***: no parent/ upstream task is in a skipped state, i.e. all parents are in a success, failed, or upstream\_failed state.
* ***none\_failed\_min\_one\_success***: The task runs only when all parent/ upstream tasks have not failed or upstream\_failed, and at least one upstream task has succeeded.

*In below snippet, for task t4 since one upstream task, t3 is skipped therefore t4 is also skipped*.

A screenshot of a computer screen

Description automatically generated

* ***dummy***: dependencies are just for show, trigger at will.

### XComs, Branch Operator & Trigger Rules– Python Implementation

* Follow below path for the python script -
  + *SubDag Function 1****: /main/airflow-working-folder/dags/* *taskgroup/* *taskgroup\_downloads.py***
* *XCom Script*

XCom push and pull methods are used to push the data to an XCom key and to pull the value back from the key.

***Note:***

* + *If a Python callable function returns a value, then it will be pushed to XComs.*
  + *If 2 XComs have same key then while accessing that key then the XCom with recent execution date will be pulled.*

XComs can be accessed through taks instance.



* *Branch Python Operator Script*

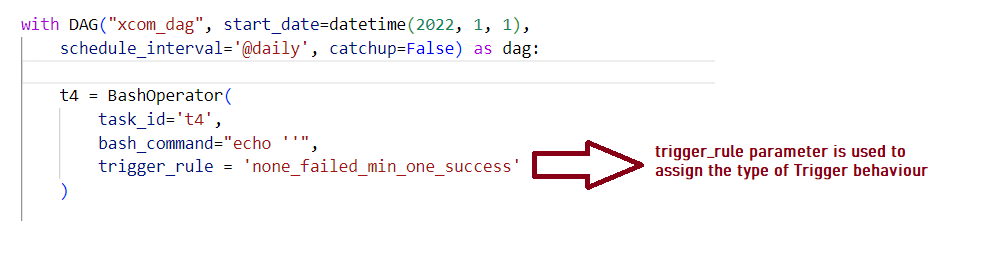
BranchPythonOperator is used to run tasks based on condition.

A screenshot of a computer program

Description automatically generated

* *Trigger Group Script*

***trigger\_group*** parameter is used to change the behaviour of triggering a task.



## Airflow Context

An ***Airflow context*** is essentially a dictionary that carries information between tasks in a DAG. It allows you to pass data, configuration parameters, or even dynamic values generated during execution.

Ref: <https://www.astronomer.io/docs/learn/airflow-context/>

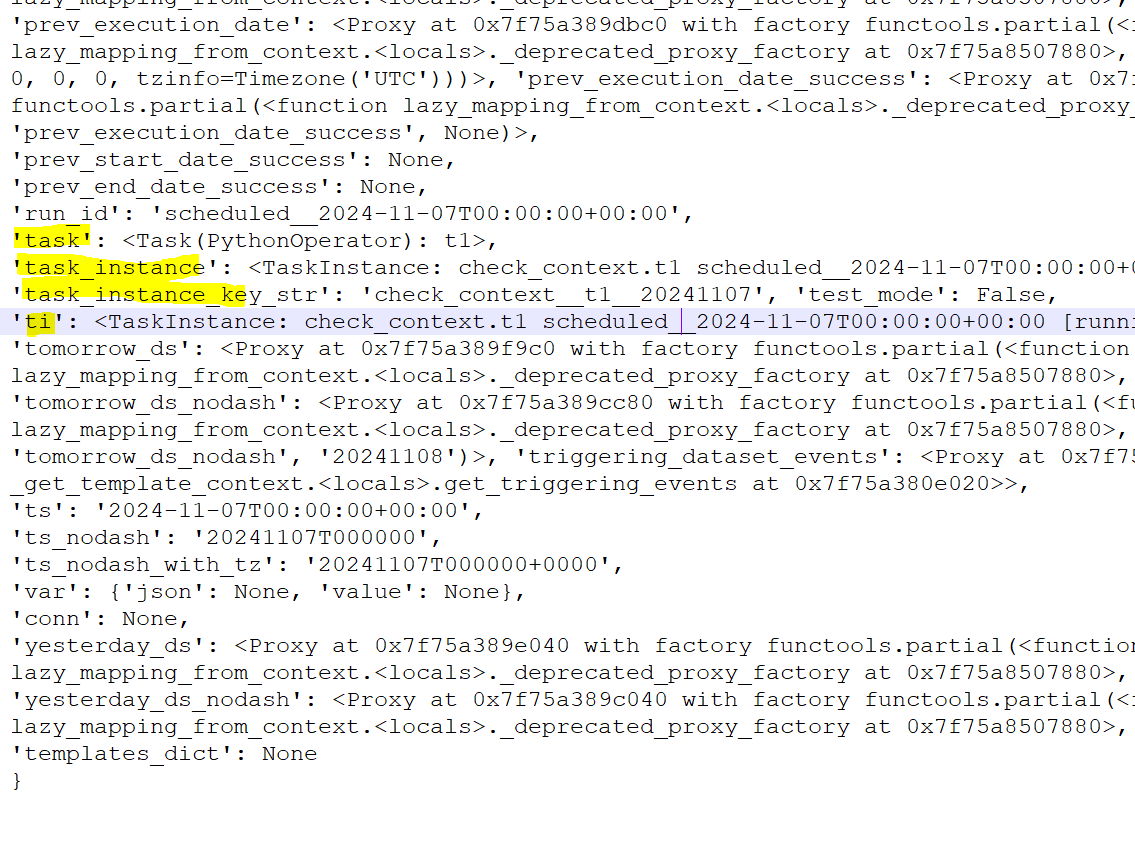
Python Script

*Python script can be found at:* ***/main/airflow-working-folder/dags/check\_context.py***

*Code Snippet:*



*Content of context dictionary:*



## Params

* Params are arguments which you can pass to an Airflow DAG or task at ***runtime*** and are stored in the *Airflow context dictionary* for each DAG run.
* Params are set at DAG Level, whenever a DAG is executed from the UI, a form will appear to pass the params.
* *Refer*: <https://www.astronomer.io/docs/learn/airflow-params/>
* ***TriggerDagRunOperator*** can be used to execute a DAG from another DAG and params can be passed programmatically using ***conf*** parameter.

A computer screen shot of a program

Description automatically generated

In above example, a Dag Operator is used to execute another DAG “*tdro*”, parameter “*upstream\_color*”is set programmatically and “*trdo*” DAG will be executed using the value provided from main DAG.

Python Script

*Python script can be found at:* ***/main/airflow-working-folder/dags/check\_params.py***

*Code Snippet:*

*While defining param at DAG, either directly specify a default value or use the****Param****class to define a default value with additional attributes.*

A screenshot of a computer program

Description automatically generated

## Variables

* A ***Variable*** is a key-value pair that can be used to store information in Airflow environment. They are commonly used to store instance level information that rarely changes, including secrets like an API key or the path to a configuration file.
* Airflow variables store key-value pairs or short JSON objects, in python it is imported as ***“from airflow.models.variable import Variables”.***
* Airflow stores variables in the metadata database by default. When a variable is created using ***Variable.set()*** or fetched using ***Variable.get()***, Airflow retrieves the value from the metadata database.
* Variable Value ***gets stored in the context*** which can be further used in other tasks.
* It is necessary to avoid using Airflow variables outside of tasks in top-level DAG code, as they will create a connection to the Airflow metastore every time the DAG is parsed.
* To store sensitive information or secrets in your Variables, use ***Secret Backends***.

***Set Variables – Python Script:***

A screen shot of a computer code

Description automatically generated

***Get Variables– Python Script:***



*Aside from Airflow variables, there are other ways of storing information in Airflow –*

* ***Environment variables*** *store small pieces of information that are available to the whole Airflow environment.*

*There is no direct way to see environment variables in the Airflow UI but they can be accessed using****os.getenv("MY\_ENV\_VAR")****inside of Airflow DAGs and tasks.*

*Env Variables are also often used to store credentials.*

*When you create an Environment Variable, Airflow doesn’t store the value in the metadatabase but stays in the environment in which Airflow runs. That means you won’t make a connection to the database each time you fetch the variable.*

* ***Params****can be used to store information specific to a DAG or DAG run.*

*Params are not encrypted and should not be used to store secrets.*

* ***XComs****can be used to pass small pieces of information between Airflow tasks.*

### Python Script

*Set and Get Variable Value using Variable Class:*

* While setting the Variable values for JSON make sure, ***serialize\_json*** parameter is set as ***True***
* While getting the Variable values for JSON make sure, ***deserialize\_json*** parameter is set as ***True***

A screenshot of a computer

Description automatically generated

*Get Variable Value using Context:*

* Variable Value gets stored in the context as well therefore while extracting data from variable, context dictionary can also be used.
* While using task decorator, make sure while calling the task function it is assigned to a variable so that it can be further used to setup dependencies.

A screenshot of a computer program

Description automatically generated

## Dynamic DAGs – Jinja Template

Jinja templates can be used to dynamically generate DAGs or parts of DAGs at runtime based on configurations mentioned in YAML file.

### Python Script to generate dynamic DAGs

Follow below scripts in following order to understand the implementation of jinga2 template –

1. *DAG Jinja Template*: **/main/jinja-template/template\_dag.jinja2**

*{{dag\_id}}*: *{{}}* is the placeholder in the template which will be replaced by jinja2 processor

A screenshot of a computer

Description automatically generated

1. *Config File 1*: **/main/jinja-template/config\_company1.yml**

These are the YAML files, which needs to be executed on the template to create the DAG. This file will contain all the details of the placeholders like DAG config details, Operators etc.

A screenshot of a computer code

Description automatically generated

1. *Config File 1*: **/main/jinja-template/config\_company2.yml**

A screenshot of a computer program

Description automatically generated

1. *DAG Generator Code*: **/main/jinja-template/dynamic\_dag\_generator.py**

This is used to create DAGs using jinja2 python module and yaml configurations.

A screenshot of a computer program

Description automatically generated

*O/P after executing the DAG Generator*

A screenshot 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 Operators

<https://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/operators/>

* Airflow Schedules and Triggers

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

* Airflow Context

<https://www.astronomer.io/docs/learn/airflow-context/>

* Airflow Params

<https://www.astronomer.io/docs/learn/airflow-params/>

* Airflow Variables

<https://www.astronomer.io/docs/learn/airflow-variables/>

* Airflow Notification and Callbacks

<https://www.astronomer.io/docs/learn/error-notifications-in-airflow/>

<https://www.restack.io/docs/airflow-faq-administration-and-deployment-logging-monitoring-callbacks-01>

* Airflow Template – Dynamic DAGs

<https://www.youtube.com/watch?v=HuMgMTHrkn4>

* Airflow Dynamic Tasks

<https://airflow.apache.org/docs/apache-airflow/stable/authoring-and-scheduling/dynamic-task-mapping.html>

## Remove Example DAGs from Airflow

To remove the examples from the Airflow Docker container, change environmental variable “*AIRFLOW\_\_CORE\_\_LOAD\_EXAMPLES” to False* in docker-compose yaml file.

A screen shot of a computer code

Description automatically generated