Week 2 learnings

Online copy of this doc is available at : <https://docs.google.com/document/d/1Ew_LCwvhQEfjkdIvw5r4Zxa33qwMLCnJUT69HX-iqxE/edit?usp=sharing>

# **Q - What is Apache Airflow and Why it is used?**

* It is a platform to programmatically author(Create), Schedule and Monitor workflows(Sequence of actions).
* It is used to create workflows to **automate tasks** using python programming language and these workflows can be monitored with it.

# **Q - What are the different alternatives available?**

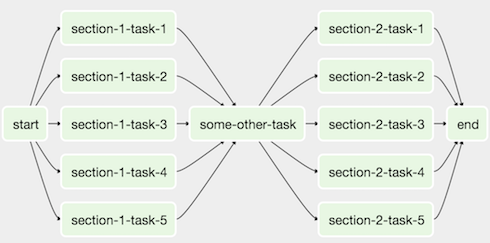
* Luigi
* Jenkins
* Kafka
* Argo
* AWS step functions

# **Q - How to install Airflow?**

* On windows machine I followed below steps
  + Open **“Turn windows features on and off”** option and enable **“Windows Subsystem for Linux”.**
  + Open **“Microsoft Store”** and install **“Ubuntu”**. Then restart the system.
  + Open **Ubuntu** terminal and set username and password.
  + Switch to root user by typing **“sudo su”** and enter the password.
  + Now, install **Python3 and pip** using the below command.
    - apt-get install python3-pip
  + Then using pip install apache-airflow
    - pip install apache-airflow
  + Once done, check whether the **“airflow”** directory got created or not under **/root** dir.
  + Run below command to initiate airflow DB, by default SQLite database is used.
    - **airflow initdb**
  + Run below command to initialize the UI of airflow.
    - **airflow webserver -p 8888**
    - Note : If the port is used we can use any other unused port while initializing the webserver.

# **Q - What are DAGs?**

* DAG stands for Directed Acyclic Graph. DAG is a collection of all the tasks you want to run and organized in a way that reflects their relationships and dependencies.

.

* Above graph is the representation of DAGs. It shows the relations and dependencies of tasks. **“Some-other-task”** is dependent on 5 tasks.
* **DagRun**
  + It is an instance of a DAG.It is a metadata of a DAG which is stored in the database. This tells us **how many times a dag has been executed.**
  + It is created by Scheduler or we can manually trigger a DAG to create it’s DagRun. We can also say it’s a log history of previously executed dag.
* **Task**
  + Task is a node in DAG.
  + Tasks are nothing but an action/work in Airflow.
  + Tasks can be defined by Operator, Sensor or a hook.

# **Q - Create and run some sample DAGs and try to schedule some python scripts in it.**

* To create DAG, we need to create **.py** file in **/airflow/dags dir**
* In the **.py** file, we need to write code to create DAG by following 5 below steps.
  + Import packages
  + Create a dict of default\_args.
  + Initialize DAG.
  + Define tasks.
  + Define dependencies of tasks.
* Once the DAG is created, it takes by default 300 seconds to reflect on airflow UI.
  + We can update the refresh time(in secs) in the **airflow.cfg** file.
    - dag\_dir\_list\_interval = 60
* While creating a DAG object in a .py file, We can schedule, when we want that DAG to execute.
  + dag = DAG(dag\_id='Second\_dag\_python\_operator',

default\_args=default\_args,

# schedule\_interval='\*/2 \* \* \* \*'

schedule\_interval=None

)

* Using python operator, we can run python methods.
  + **Step 1 : from airflow.operators.python\_operator import PythonOperator**
  + **Step 2 : define python method eg : printHelloWorld()**
  + **Step 3 : define task eg : t1 = PythonOpertaor(task\_id=’python\_task\_1’,**

**python\_callable=’printHelloWorld’,dag=dag)**

# **Q - User Authentication on Airflow**

* Refer this video for setting up authentication - <https://www.youtube.com/watch?v=IBi0P1hmiUw&ab_channel=MarcLamberti>
* There are 3 steps to set un and pw of airflow UI.
  + Open **airflow.cfg** and update/add below 2 lines under **[webserver]** section.
    - [webserver]

# Set to true to turn on authentication:

# https://airflow.apache.org/security.html#web-authentication

authenticate = True

auth\_backend = airflow.contrib.auth.backends.password\_auth

* + Install below package
    - pip3 install flask-bcrypt
  + create/add user via python REPL by writing below script.

root@DESKTOP-UM554UI:~/airflow# python3

Python 3.8.5 (default, Jul 28 2020, 12:59:40)

[GCC 9.3.0] on linux

Type "help", "copyright", "credits" or "license" for more information.

>>> from airflow import models, settings

>>> from airflow.contrib.auth.backends.password\_auth import PasswordUser

>>> user = PasswordUser(models.User())

>>> user.username = 'admin'

>>> user.email = 'vishalkashyap1895@gmail.com'

>>> user.password = ’n0ob007'

>>> session = settings.Session()

>>> session.add(user)

>>> session.commit()

>>> session.close()

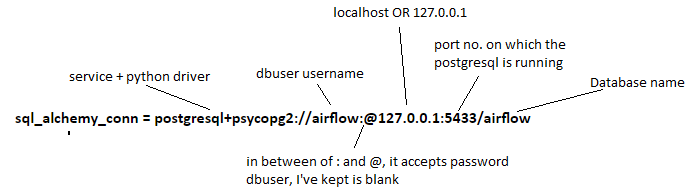
>>> exit()

root@DESKTOP-UM554UI:~/airflow#

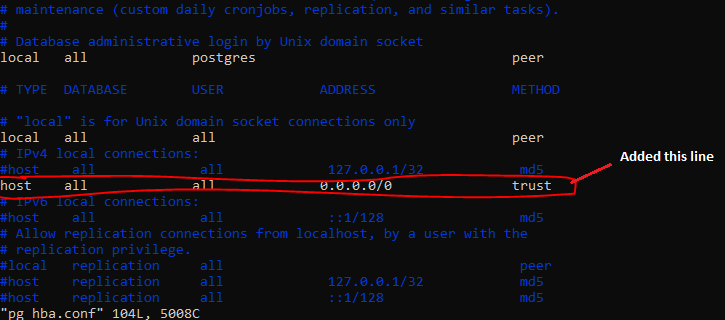
# **Q - Use** Mysql**/postgresql as back end db for airflow**

* I have used postgresql as a backend for apache airflow. By default the backend was SQLite DB.
* Follow below steps to change the backend.
  + Initially after airflow installation, when we initialize the apache airflow webserver and initdb, **airflow.cfg** files get created with default configurations.
  + Step 1 :
    - Install postgresql and psycopg2 onto the machine.
    - **apt-get update**
    - **apt-get install postgresql postgresql-contrib**
    - **apt-get install libpq-dev**
    - **pip3 install psycopg2-binary**
    - **pip3 install psycopg2**
    - **service postgresql start #[To start the database]**
  + Step 2 :
    - Create Database user, db and grant all permissions
    - **sudo -i -u postgres** # to login as admin user in postgresql
    - **psql** # to go to database prompt
    - **CREATE USER airflow;** # Created user without password
    - **CREATE DATABASE airflow;** # Created database
    - **GRANT ALL PRIVILEGES ON DATABASE airflow TO airflow;** # granting permission to access the airflow(DB) by airflow(user)
    - **postgres=# \list** # this command will list the databases.
    - **postgres=# \du** # this command will list the available users in DB.
  + Step 3 : Update airflow.cfg file
    - Open **/airflow/airflow.cfg** file and search for **“sql\_alchemy\_conn”**
    - By default the value would be **“sqlite:////root/airflow/airflow.db”**
    - We need to comment default line and change it to

**sql\_alchemy\_conn = postgresql+psycopg2://airflow:@127.0.0.1:5433/airflow**



* + Step 4 - Update **/etc/postgresql/12/main/pg\_hba.conf** file.
    - This is how my pg\_hba.conf file looks



These are the links that I referred for changing backend

1. <https://stackoverflow.com/questions/58380835/implementing-postgres-sql-in-apache-airflow>
2. <https://stackoverflow.com/questions/59383798/airflow-postgresql-backend-psycopg2-operationalerror-fatal-ident-authenticat>
3. <https://www.ryanmerlin.com/2019/07/apache-airflow-installation-on-ubuntu-18-04-18-10/>

# **Q - Different operators in Airflow**

For more operators refer :- <https://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/operators/index.html>

There are many operators in apache airflow, some of them are as follows :-

* **BashOperator** - Used to run bash scripts or commands.

For e.g. :

from airflow.operators.bash\_operator import BashOperator

bash\_task\_1 = BashOperator(task\_id=’bash\_task\_1 ’, bash\_command = ‘echo hello world!’, dag = dag)

* **PythonOperator** - Used to run the python scripts/methods as a task.

For e.g. :

from airflow.operators.python\_operator import PythonOperator

python\_task\_1 = PythonOperator(task\_id=’python\_task\_1’, python\_callable = ‘some\_python\_method’, dag = dag)

* **DummyOperator** - This operator does nothing, as the name suggests it is just used to group the tasks and create dummy relations.

For e.g. :

from airflow.operators.dummy\_operator import DummyOperator

dummy\_task\_1 = DummyOperator(task\_id=dummy\_task\_1’, dag = dag)

* **Sensor operator -** This operator is used to sense or check for the particular criteria. For e.g : Check for a xyz.py file in /dir1/dir2 folder, if there is a file then execute the downstream task else wait for specified time. There are many Sensor operators.

1. <https://www.astronomer.io/guides/what-is-a-sensor>
2. <https://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/operators/sensors/index.html>

* **Email Operator** - create a task to send an email on specified ids.

# **Q - Explore different commands for airflow cmd**

There are many commands for apache airflow that we can run via CLI.

* **airflow initdb** - For initializing the Database.
* **airflow webserver** - For initializing the webserver on which the airflow UI runs.
* **airflow scheduler** - The process that checks and schedules tasks based on the interval set in DAG.
* **airflow list\_dags** - To list all the available dags.
* **airflow test <dag\_id> <task\_id> <date>** - To run a particular task in a dag manually.
* **airflow info** - Will print out all the details like DB, airflow version, directories etc.
* **airflow trigger\_dag <dag\_id>** - This command will trigger dag externally.
* **airflow next\_excution <dag\_id>** - Will give next scheduled time of dag.

For more commands, refer : <https://airflow.apache.org/docs/apache-airflow/stable/cli-ref>

# **Q - Create/Define dependencies between Task/Dags**

**For more info, refer this :** <https://www.astronomer.io/guides/managing-dependencies>

This logic can be set three ways:

* d1.set\_downstream(d2)  
  d2.set\_downstream(d3)  
  d3.set\_downstream(d4)
* In above e.g. : d4 task is dependent on d3, d3 is dependent on d2, and d2 is dependent on d1. **Creating graphs in a top down approach.**
* d4.set\_upstream(d3)  
  d3.set\_upstream(d2)  
  d2.set\_upstream(d1)
* In above e.g. : d4 task is dependent on d3, d3 is dependent on d2, and d2 is dependent on d1. **Creating graphs in a bottom up approach.**
* d1 >> d2 >> d3 >> d4
* d4 << d3 << d2 << d1

# **Q - Cross Task communication using xcom**

<https://www.youtube.com/watch?v=nVT5SgdvD0Q&t=323s>

* Basically, it is use to interchange the data between tasks. Xcom is a table in database which stores the data push(sent) from 1 task and pull(received) into another task.
* Below Dag shows the implementation of data exchange via xcom.
* Step 1 : Import statements
  + from airflow.operators.bash\_operator import BashOperator
  + from airflow.operators.python\_operator import PythonOperator
  + from datetime import datetime
  + from airflow import DAG
  + from random import random

dag = DAG(dag\_id='Second\_dag\_python\_operator\_with\_xcom',

* + default\_args=default\_args,
  + # schedule\_interval='\*/2 \* \* \* \*'
  + schedule\_interval=None
  + )
* Step 2 : Create default\_args dict
  + default\_args = {
  + 'owner':'Vishal Kashyap',
  + 'start\_date': datetime(2020,12,2)
  + }
* Step 3 : Initialize DAG
  + dag = DAG(dag\_id='Second\_dag\_python\_operator\_with\_xcom',
  + default\_args=default\_args,
  + # schedule\_interval='\*/2 \* \* \* \*'
  + schedule\_interval=None
  + )
* Step 4 : Create tasks using python\_operator
  + def push\_random\_value\_to\_xcom(\*\*context):
  + random\_value = random()
  + context['ti'].xcom\_push(key='random\_value',value=random\_value)
  + print("Random Value generated in python and pushed to xcom -

{}".format(random\_value))

* + def pull\_random\_value\_from\_xcom(\*\*context):
  + ## 'ti' = task\_instance
  + received\_random\_value = context['ti'].xcom\_pull(key='random\_value')

print("Random value received from xcom - {}".format(received\_random\_value))

* + python\_task\_push\_random\_value\_to\_xcom = PythonOperator(

task\_id='python\_task\_push\_random\_value\_to\_xcom',

provide\_context = True,

python\_callable=push\_random\_value\_to\_xcom,

dag=dag)

* + python\_task\_pull\_random\_value\_from\_xcom = PythonOperator(

task\_id='python\_task\_pull\_random\_value\_from\_xcom',

provide\_context = True,

python\_callable=pull\_random\_value\_from\_xcom,

dag=dag)

* Step 5 : Create dependencies
  + python\_task\_push\_random\_value\_to\_xcom >> python\_task\_pull\_random\_value\_from\_xcom

# **Q - Running executors as local or using celery**

1. <https://www.astronomer.io/guides/airflow-executors-explained>
2. <https://airflow.apache.org/docs/apache-airflow/stable/executor/celery.html>

## What is an Executor?

* + An executor is responsible for running the tasks in which is defined in DAGs.
  + Once a DAG is defined (perhaps with the help of an Operator), the following needs to happen in order for a single or set of "tasks" within that DAG to execute and be completed from start to finish:
    - The *Metadata Database* keeps a record of all tasks within a DAG and their corresponding status (queued, scheduled, running, success, failed, etc) behind the scenes.
    - The *Scheduler* reads from the Metadatabase to check on the status of each task and decide what needs to get done (and in what order).
    - The *Executor* works closely with the Scheduler to figure out what resources will actually complete those tasks (via a worker process or otherwise) as they're queued.

## What is a LocalExecutor?

* + Running Apache Airflow on a LocalExecutor exemplifies single-node architecture.
  + The LocalExecutor completes tasks in parallel that run on a single machine (think: your laptop, an EC2 instance, etc.) - the same machine that houses the Scheduler and all code necessary to execute. A single LocalWorker picks up and runs jobs as they’re scheduled and is fully responsible for all task execution.

## What is a CeleryExecuter?

* + CeleryExecutor is one of the ways you can scale out the number of workers. For this to work, you need to setup a Celery backend (RabbitMQ, Redis, …) and change your **airflow.cfg** to point the executor parameter to CeleryExecutor and provide the related Celery settings.
  + Celery itself is a way of running python processes in a distributed fashion. To optimize for flexibility and availability, the CeleryExecutor works with a "pool" of independent workers across which it can delegate tasks, via messages.
  + If a worker node is ever down or goes offline, the CeleryExecutor quickly adapts and is able to assign that allocated task or tasks to another worker.
  + If you're running native Airflow, adopting a CeleryExecutor means you'll have to set up an underlying database to support it (RabbitMQ/Redis).

# **Q - Run airflow web server and scheduler as daemon service on your local rather than manually running start and stop command.**

service start airflow

service stop airflow

service restart airflow

1. <https://towardsdatascience.com/how-to-run-apache-airflow-as-daemon-using-linux-systemd-63a1d85f9702>
2. <https://linuxhandbook.com/system-has-not-been-booted-with-systemd/>

# **Q - Keep the DAG in git. No need to manually edit DAG on the server. So keep dag in git in a different folder than the default DAG folder i.e in the source code itself.**

<https://www.youtube.com/watch?v=AjBADrVQJv0&t=16s&ab_channel=MarcLamberti>

# **Q - Check with different integrations supported by Airflow**

## We can integrate airflow with many platforms like :

* Microsoft Azure
* Google cloud platform
* Amazon AWS
* Databricks
* <https://airflow.apache.org/docs/apache-airflow/stable/integration.html>

# **Q - Go through the google cloud composer service by GCP.**

<https://www.youtube.com/watch?v=gFQVmsRRY_A&ab_channel=GoogleCloudPlatform>