Overview on Apache Airflow:

- Manage scheduling and running jobs and data pipelines.
- Ensures jobs are ordered correctly based on dependencies.
- Provides mechanisms for tracking the state of jobs and recovering from failure.
- Airflow is using the Python programming language to define the pipelines. Users can take full advantage of that by using for loop to define pipelines, executing bash commands, using any external modules like pandas, sklearn or GCP or AWS libraries to manage cloud services and much, much more.

Apache Airflow can be used to schedule:

- ETL pipelines that extract data from multiple sources and run Spark jobs or any other data transformations.
- Training machine learning models.
- Report generation.
- Backups and similar DevOps operations.

Components of Airflow:

- **Task**: a defined unit of work (these are called operators in Airflow)
- **Task instance**: an individual run of a single task. Task instances also have an indicative state, which could be "running", "success", "failed", "skipped", "up for retry", etc.
- **DAG**: Directed acyclic graph, a set of tasks with explicit execution order, beginning and end
- **DAG run**: individual execution/run of a DAG
- **Operator**: implementation of a task.
- **Web Server**: The GUI. This is under the hood a Flask app where you can track the status of your jobs and read logs from a remote file store.
- **Scheduler:** This component is responsible for scheduling jobs. This is a multithreaded Python process that uses the DAG object to decide what tasks need to be run, when and where.
- **Executor:** The mechanism that gets the tasks done
- **Metadata Database :**Stores the airflow states

Instructions to Run Apache airflow on local machine:

Prerequisite:

- Install Docker desktop on your system.

  Follow below links to install Docker:

  URL: https://docs.docker.com/docker-for-windows/install/

- Before Docker installation enable windows subsystem for Linux.(only for windows user)

You most wondering, what is the Docker? Let's play around with it.

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and deploy it as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.

I. Setting up your local Airflow server

Step 1

Create local directory and place ***docker-compose.yaml*** in the directory

Step 2-a | first time build:

Build the images by running the following commands:

```
SET VOLUME_HOST_PATH=[Path to directory on host]
```

```
docker-compose up airflow-init
```

- Replace *[Path to directory on host]* with a path to a directory on your local machine where you'd like to create the volume. This directory will be shared by your containers filesystem and local machines filesystem
- i.e. I use ***SET VOLUME_HOST_PATH=/c/tmp*** on my local machine since I do not have Admin access to create volumes under my username directory. Ideally, you should create the volume in the same directory where you're keeping the *docker-compose.yaml* file since it will be better organized, as everything will live in one location

Step 2-b:

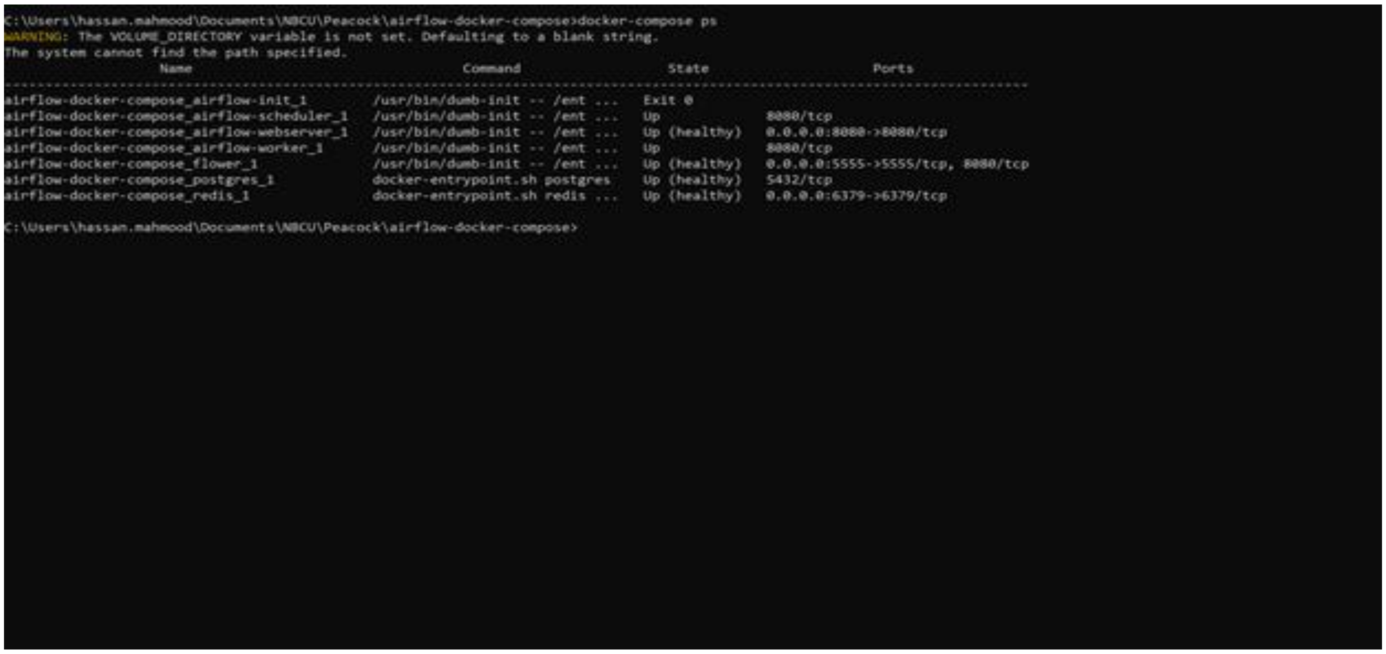
If you have already created the service and you would like to get it up and running, navigate to the directory where you have your *docker-compose.yaml* file located and run the following command:

```
docker-compose up
```

Step 3:

Once the service is up, you will see there are a handful of containers running in the background, which are interacting with each other to standup your Airflow server

```
docker-compose ps
```



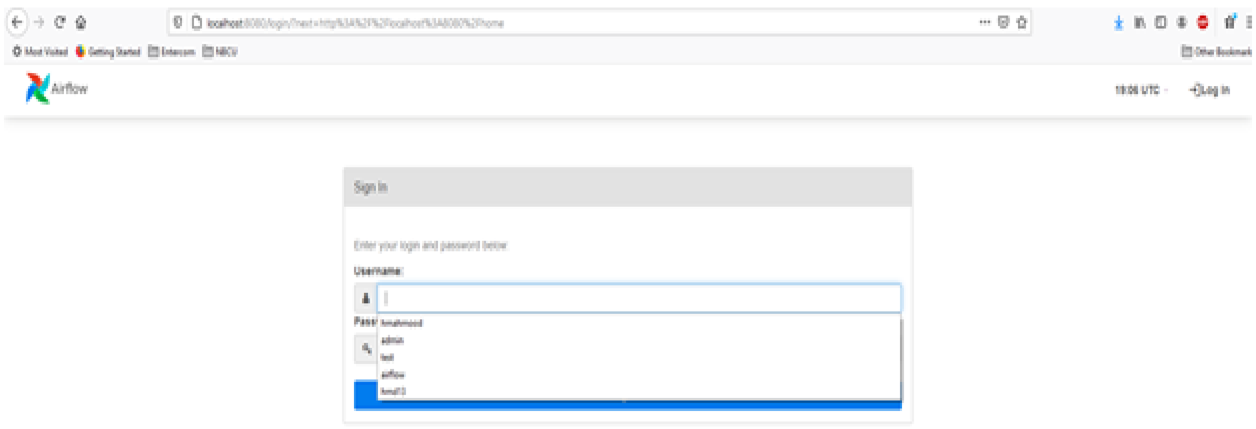
```
C:\Users\hassan.mahmood\Documents\NBCU\Peacock\airflow-docker-compose>docker-compose ps
WARNING: The VOLUME_DIRECTORY variable is not set. Defaulting to a blank string.
The system cannot find the path specified.
                 Name                               Command                  State                     Ports
--------------------------------------------------------------------------------------------------------------------------
airflow-docker-compose_airflow-init_1        /usr/bin/dumb-init -- /ent ...   Exit 0
airflow-docker-compose_airflow-scheduler_1   /usr/bin/dumb-init -- /ent ...   Up              8080/tcp
airflow-docker-compose_airflow-webserver_1   /usr/bin/dumb-init -- /ent ...   Up (healthy)    0.0.0.0:8080->8080/tcp
airflow-docker-compose_airflow-worker_1      /usr/bin/dumb-init -- /ent ...   Up              8080/tcp
airflow-docker-compose_flower_1              /usr/bin/dumb-init -- /ent ...   Up (healthy)    0.0.0.0:5555->5555/tcp, 8080/tcp
airflow-docker-compose_postgres_1            docker-entrypoint.sh postgres    Up (healthy)    5432/tcp
airflow-docker-compose_redis_1               docker-entrypoint.sh redis ...   Up (healthy)    0.0.0.0:6379->6379/tcp

C:\Users\hassan.mahmood\Documents\NBCU\Peacock\airflow-docker-compose>
```

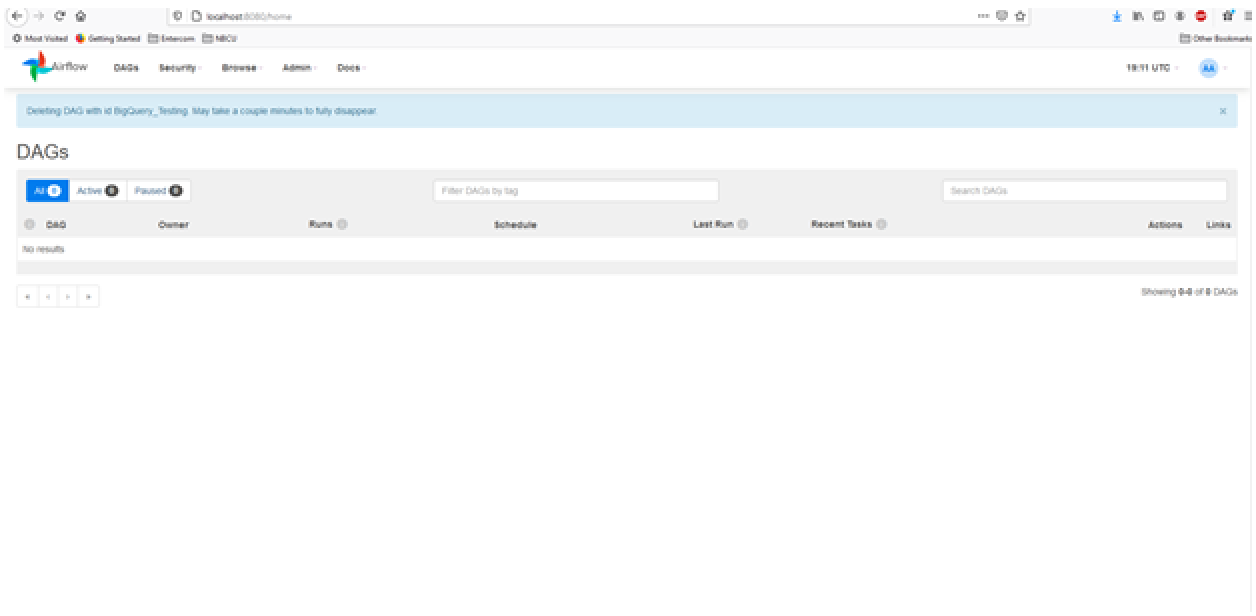
Step 4:

Open your web-browser and navigate to ***localhost:8080*** (also know as ***127.0.0.1:8080***), which should bring you to the login page. The default username and password are set to "airflow" and "airflow". You can see this on lines 118 and 119 in the *docker-compose.yaml* file, under the *airflow-init* block's environment variables.

A successfully login with bring you the Airflow UI, shown in the subsequent image.



Enter your login and password below



Deleting DAG with id BigQuery_Testing. May take a couple minutes to fully disappear

No results

Showing 0-0 of 0 DAGs

Documentation:

docker-compose	https://docs.docker.com/compose/reference/up/

Understanding Airflow Pipelines:

An Airflow pipeline is essentially a set of parameters written in Python that define an Airflow Directed Acyclic Graph (DAG) object. Various tasks within a workflow form a graph, which is Directed because the tasks are ordered. To avoid getting stuck in an infinite loop, this graph does not have any cycles, hence Acyclic.

Let’s create your first DAG:

```
🗁 airflow # airflow root directory
        🗁 ├─ dags          # the dag root folder
                ├─ first_dag.py       #  where you put your first task
```

Steps to write an Airflow DAG:

- A DAG file which is basically just a python script, is a configuration file specifying the DAG’s structure as code.
- There are only 5 steps to write an Airflow DAG
  - Step 1:importing modules
  - Step 2:Default Arguments
  - Step 3:Instantiate a DAG
  - Step 4:Creating Tasks
  - Step 5:Setting up Dependencies

Step 1:Importing Modules

```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import PythonOperator
```

Step-2: Default Arguments

```
default_args = {
```

```
                              # Basically  just the name of the DAG owner
                              "owner": "airflow",
                              # is a Boolean Value. If you set it to true the current running
test
                                  Instance will rely on the previous task's status
                             "depends_on_past": False,
                             # determines the execution day of the first DAG  task instant
                             "start_date": datetime(2020, 9, 15),


                              # is just where you will receive the email notification from
                              "email": ["airflow@example.com"],
                            # is used to define whether you want to receive the notification if a
                               failure happens
                              "email_on_failure": False,
                           #is used to define whether you want to receive an mail every time
                             a retry happens
                              "email_on_retry": False,
                         # dictates the number of times Airflow will attempt to retry a failed
                            task
                              "retries": 1,
                       # is the duration between consecutive retries
                              "retry_delay": timedelta(minutes=2),
}
```

Step 3: Instantiate a DAG

This step is about instantiating a DAG by giving it a name and passing in the default argument to your DAG here: default_args=default_args.

Then set the schedule interval to specify how often DAG should be triggered and executed.

```
dag = DAG(
        "hello_world",
         description="first DAG",
          schedule_interval="@daily",
          default_args=default_args,
          catchup=False,
      )
```

Different type of Schedule_interval:

@once : schedule once and only once

@hourly:Run once an hour at the beginning of the hour(cron: 0 * * * *)

@daily:Run once a day at midnight(cron:0 0 * * *)

@weekly:Run once a week at midnight on Sunday morning(cron:0 0 * * 0)

@monthly:Run once a month at midnight of the first day of the month Cron(0 0 1 * *)

@yearly:Run once a year at midnight of January 1(cron:0 0 1 1*)

Step 4: Creating tasks

```
task_a = DummyOperator(task_id="task_a")
task_b = DummyOperator(task_id="task_b")
task_c = DummyOperator(task_id="task_c")
task_d = DummyOperator(task_id="task_d")
```

Step 5: Setting up Dependencies

```
task_a >> [task_b, task_c]
task_c >> task_d
```

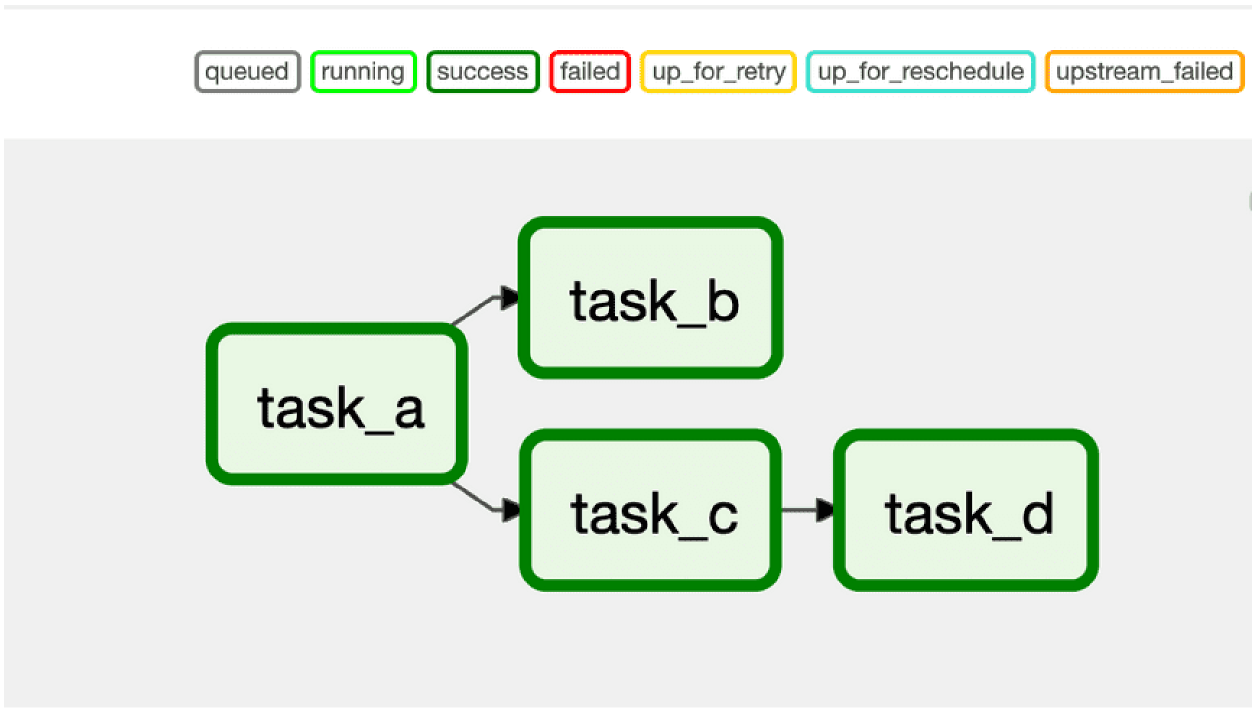
- Every task in a Airflow DAG is defined by the operator (we will dive into more details soon) and has its own task_id that has to be unique within a DAG. Each task has a set of dependencies that define its relationships to other tasks. These include:

  Upstream tasks — a set of tasks that will be executed before this particular task.

  Downstream tasks — set of tasks that will be executed after this task.

- In our example task_b and task_c are downstream of task_a. And respectively task_a is in upstream of both task_b and task_c. A common way of specifying a relation between tasks is using the >> operator which works for tasks and collection of tasks (for example list or sets).

- This is how a graphical representation of this DAG looks like:



Different types of Operator:

- Python operator
- Bashoperator

- DummyOperator
- JdbcOperator
- MysqlOperator
- BigQueryToGCSOperator
- BigQueryExecuteQueryOperator
- GCSToBigQueryOperator
- EC2StartInstanceOperator
- S3CreateBucketOperator
- S3FileTransformOperator
- SparkJDBCOperator
- SparkSqlOperator
- SparkSubmitOperator etc.

As mentioned already, each task in Airflow DAG is defined by an operator. Every operator is a pythonic class that implements the `execute` method that encapsulates the whole logic of what is executed. Operators can be split into three categories:

- Action operators — for example, BashOperator (executes any bash command), PythonOperator (executes a python function) or TriggerDagRunOperator (triggers another DAG).
- Transfer operators — designed to transfer data from one place to another, for example GCSToGCSOperator which copies data from one Google Cloud Storage bucket to another one. Those operators are a separate group because they are often stateful (the data is first downloaded from source storage and stored locally on a machine running Airflow and then uploaded to destination storage).
- Airflow ships with few built-in operators like the mentioned `BashOperator`. However, users can easily install additional operators using provider's packages. In this way Airflow makes it easy to use hundreds of operators that allow users to easily integrate with external services like Google Cloud Platform, Amazon Web Services or common data processing tools like Apache Spark.