

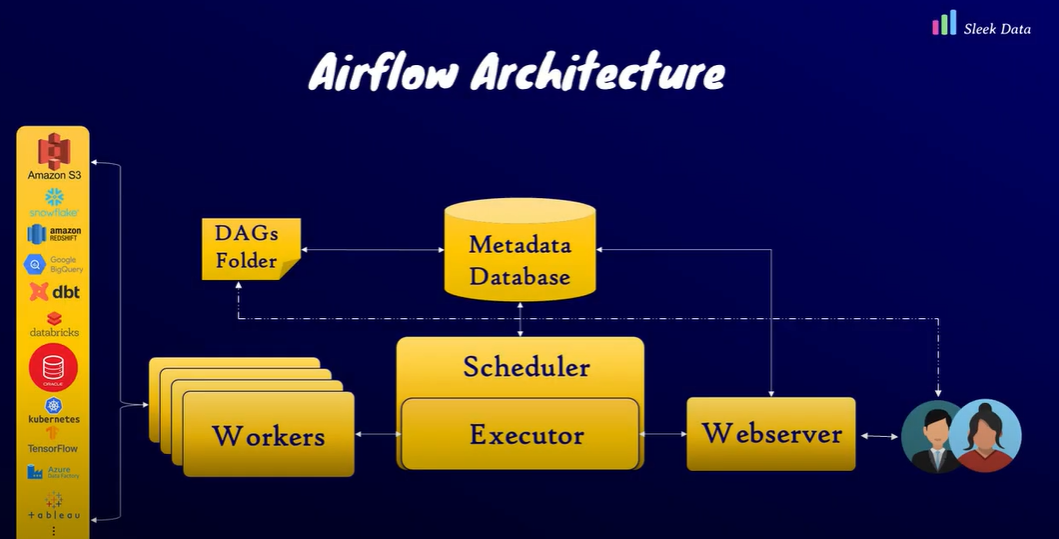
**Dag** –representation of Workflows. A DAG is a data-pipeline

**Directed** –dependencies between the tasks have a specified direction.

**Acyclic** - No cycles or loops.

**Graph** – diagram consists of nodes and edges. Each Node represents tasks and edges represent dependencies.

**DAG** consists of one or more task and each DAG is created by operator. An operator is an abstraction that defines what need to be done in the tasks



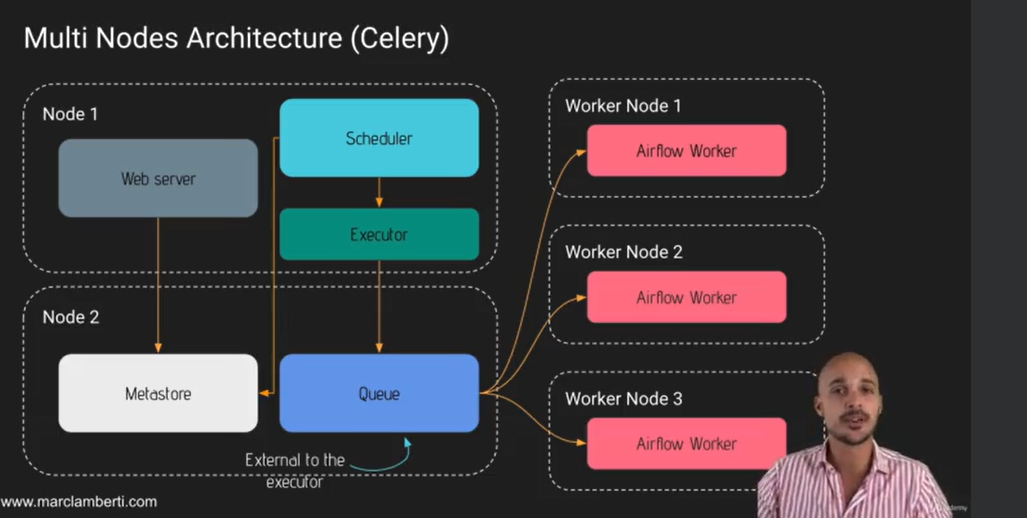
**Web server** –servers the web interface and that allows user to monitor and manage workflows (UI).

**Database(Meta Store)** – is used to store the metadata about workflows, tasks and their dependencies and it also saves the status of your workflows and tasks i.e. running failed or succeeded

**Scheduler** – is responsible for scheduling your workflows to run

**Executor** – receiving tasks from the scheduler and assigning them to the workers and monitors their execution.Executor has an **internal Queue**

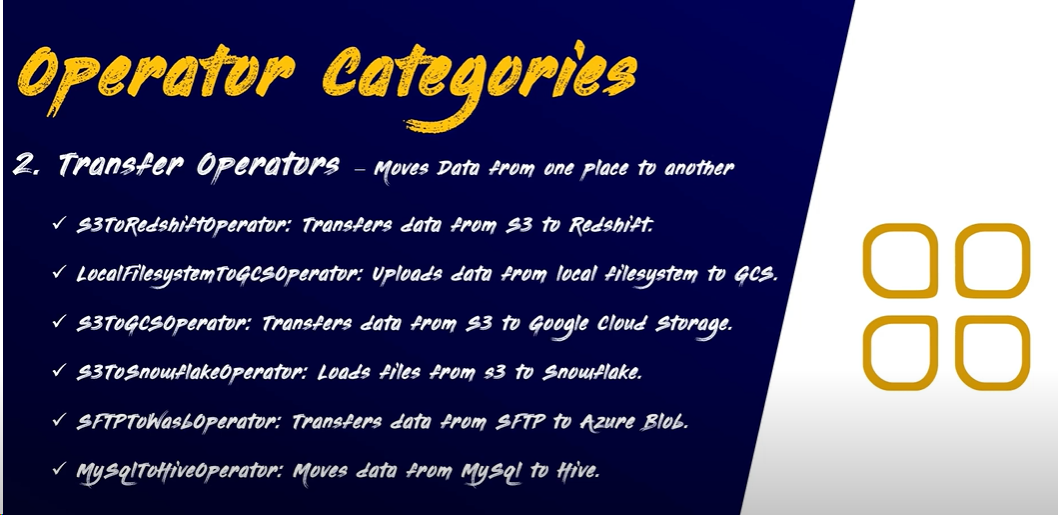
**Workers** – are the compute layers that run the tasks. They are responsible for fetching tasks from the executors and finally reporting them to the executors. Based on the specified operators and parameters workers complete their tasks





**Python Operator:**

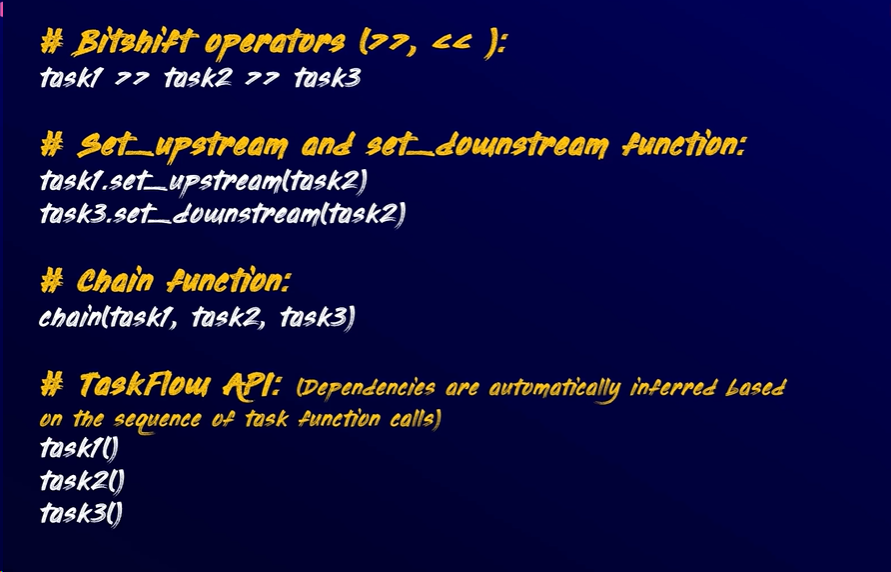


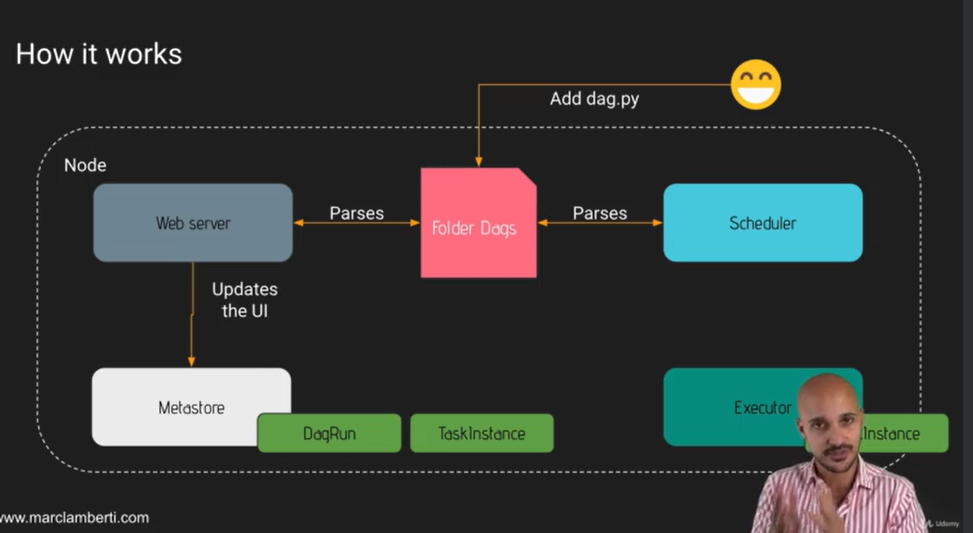






**Task Dependencies:**





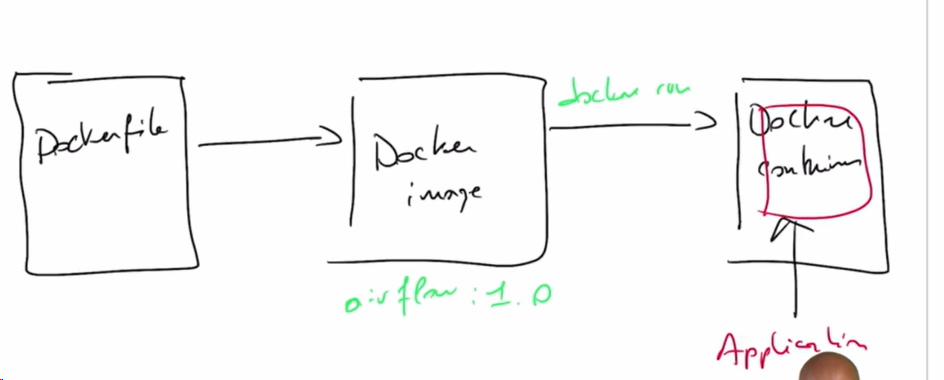
###########################################################

docker ps

docker run --rm -d -p 8081:8080 airflow-basic = to run the existing docker image

**Docker**: with docker you are able to run your application on any of the **os** without worrying about the dependencies.

* + - You have your application and in order to install your application to run it, there are some instructions that you have to run. For example, installing Python, installing wget.Then you need to create a user and so on. Some instructions that are needed by your application to run it.So those instructions actually to run your application, you are going to put all of them into a file which is called a **Docker file.**
    - **Docker file** – corresponds to all instructions to install and run your application
    - **Docker Image** – build an image based on docker file
    - **Docker Container -** Based on the image to obtain a Docker container where your application is running inside that container. Concretely, it means that inside the container, your application is running. And you have the operating system right there. The dependency is installed in the Docker container.

****

**Docker compose** – allows you to run multi-container docker applications.

* + - All the services that are inside the **compose file** are **triggered** by this **docker compose file** andall of those containers are run on same network.

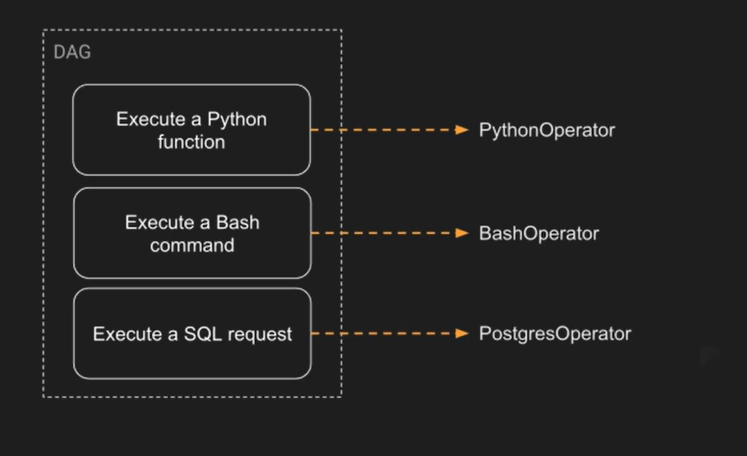
**Practical in Airflow:**

**Catchup = false** means you won't backfill or run any **missed** **runs** for the intervals between the DAG's start date and the current date.

* + - Setting **catchup** to **False** is useful in scenarios where you don't want to **run** the **DAG** for **historical** **data** or if you **don't** want to **catch** **up** on **missed** **runs**.

**Operator**: is a task in your DAG. You can think an operator is an **Object** encapsulating the **logic** and the job you want to execute

An **Operator** is conceptually a **template** for a predefined [**Task**](https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/tasks.html) that you can just define declaratively inside your **DAG**:



**3 types of operator:**

**Action operator**: is used to execute something

* + - Python operator
    - Bash operator

**Transfer operator**: is used to transfer data from source to destination

* + - Mysql etc...

**Sensor operators:** waits for something to happen before moving to next task.

* For example, you want to wait for a **file** to land at a **specific location** in your file system. One thing you can do is to use the **file sensor,** which will wait for the file to land at the specific Location in your system before moving to the **next desk.**
* So every **60** seconds, by default, the **file sensor** will check if the corresponding finding the file you are expecting exists or not, if just then it moves forward. If not, then it checks **again** every **60** seconds.
* Keep in mind, sensors allow you to verify if a condition is met or not before moving forward.

**Providers**: our airflow is divided into different providers according to their usage from airflow 2.0 above version.

**task\_id** = is different for different tasks.

**http\_conn\_id** = we need to create

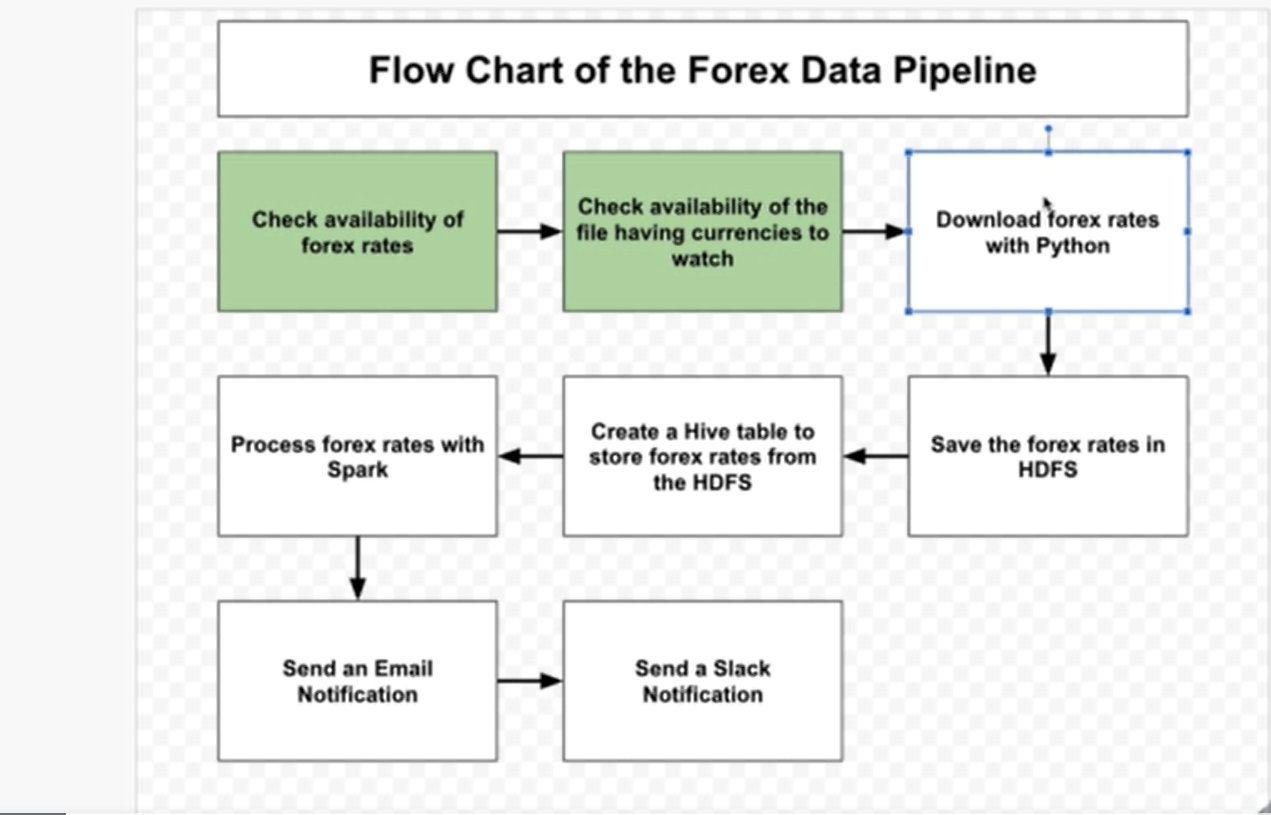
**poke\_interval** = In that case, every five seconds you are going to verify if the **Forex API** is available or not.

./start.sh = to run the dag for 1st time use this command.

**Path of local file system:**

Well, remember that airflow is running inside a Docker container, so that path corresponds to the

file system in the Docker container of airflow.



**Note**: each operator should have different **task** **id** within a day.

File Sensor:

Check the file is available or not in the specified location

Create a connection id:

**Connection name**

**Connection type** -File path

**Extras** – location - {“path”:”/opt/airflow/dags/files”} - path of the file in our airflow container

Python operator: with python operator we are going to execute **python function** and **python script**.

**Python\_callable** = this is where you specify python function or script that u want to execute on operator

**op\_kwargs** = if you want to add dict of keywords

**op\_args** = if you want to pass argument to your operator without a key

**HDFS (Hadoop distributed file system):** with these we are going to store the data in distributed file system.

**Bash Operator**: The bash operator allows you to execute a bash command and that's exactly what you are going to do in order to push the file into the Hdfs.

* **Hue** allows you to explore to query the Hdfs, the distributed file system.

So it is like a nice user interface to interact with the **Hdfs**.

**Hive**: With Hive, you are going to create a **table** on top of your **data**, on top of your **files** (json or any) so that you will be able to query your data by executing SQL like queries.

**Create hive connection to interact with hive:**

Hive conn\_id = name

Hive conn\_type =hive server 2thrift

Login = hive

Pass = hive

Port = 10000

Docker ps

Docker exec –it contained\_id /bin/bash = go inside the airflow bash session

Airflow tasks test dag\_name - to test the airflow tasks

**Spark:** with spark we able to process the data and store in Hive tables.

**Process forex rates with Spark:**

**Insert the forex\_rates.json file in HDFS into forex\_rates table in HIVE using Spark.**

Spark conn\_id = name

Conn\_type = spark

Host: spark://spark-master

Port = 7077

**Note**: When you work with Spark and Hive integration, the **metadata** of actual **Hive** **tables** is stored in **a Hive Metastore**. The Hive Metastore is a centralized repository that manages metadata information for Hive tables, such as table schemas, column names, data types, and the locations of the actual data in **HDFS**.

**Sending an Email:**

Edit the **airflow.config** file under the section [**SMTP**]

To send an email then we need to modify the SMTP settings in airflow.config file.

Every time we made changes in config file we need to restart the airflow server

* + - **docker-compose restart airflow**

smtp\_host = smtp.gmail.com

smtp\_starttls = True

smtp\_ssl = False

# Example: smtp\_user = airflow

smtp\_user = dhruvan.gowda@williamoneilindia.com

# Example: smtp\_password = airflow

smtp\_password = sfztxhtiqapojqtw #app –password of Google or Microsoft.

smtp\_port = 587

smtp\_mail\_from = dhruvan.gowda@williamoneilindia.com

smtp\_timeout = 30

smtp\_retry\_limit = 5

**Sending a Slack Notification:**

<https://slack.com/> **=** create a workspace = it is a hub of channels

Api.slack.com/apps= creates a workspace and create channel.

<https://api.slack.com/apps> = set a slack API and enable incoming webhook.

App name = is our channel name where your slack messages will go to send.

* + - Verify that the future incoming webhooks is activated as it allows you to post messages from external sources into Slack.

**Defining Task Dependencies in Airflow:**

There 2 methods

1. set\_downstream = parent(child)
2. set\_upstream = chid(parent)
3. Right bitshift operator >> equal to Set downstream
4. Left bitshit operator << equal to Set upstream

**How to define Scheduling interval:**

* + - Either using **Cron** **expressions(recommended)** or using **Timedelta** Object
    - Start date and scheduled interval

**crontab.guru** = to check the cron timings = next

**Backfilling and Catch-up in order to trigger the dag run that haven’t been started:**

**Note**: we can also edit catchup\_by\_default in airflow config file to true or false.

* + - As you can see, it is set to true. If we set it to **False**, the **Scheduler** won’t try to backfill non triggered DagRuns by default anymore.
    - Only the most recent DAG Run in late will be run and not the previous ones.

**Command: airflow backfill -s 2023-12-01 –e 2023-12-07 - -rerun\_failed\_tasks –B backfill**

* + - It will backfill all the failed dags run if catch-up is false also.
    - We need to run the above command inside the bash session of the airflow
    - Run inside the **bash** container.
    - Docker logs –f **container\_id --- > to the logs of the container**

**Schedule Interval in Airflow:**

\*5/ \* \* \* \* this means it will run every 5 min.

**If you want to add external parameters then we can use this:**

**op\_kwargs** = {‘conn\_id’: ‘my\_conn\_id’}

**Timezone**: for a specific local timezone

Import pendulum

India\_tz = pendulum.timezone (“Asia/Kolkata”)

We need to specify in the default args: (time\_zn = India\_tz)

**depends\_on\_past = True**

It is used to set up the tasks based on the previous history. If the task is failed then it will not trigger for the next run because it will **depends** on **past** **run**. If the previous run failed, the current run will be held until the previous run succeeds. It is of dag level and we need change the status to **mark as success**

**Ex: python\_task\_2** = PythonOperator (task\_id='python\_task\_2', python\_callable=second\_task, **depends\_on\_past** = **True**)

**Wait\_for\_downstream = True =** if we marked any tasks as Wait\_for\_downstream = True means current task will wait until all the others tasks to complete

* + - **.airflowignore:** where you will keep your ignore files of the airflow

**Organizing your airflow dag’s folder:**

* + - sudo apt install zip – install in ubuntu
    - Create a folder functions and create a python files **helpers.py** and add the functions
    - Create a file **\_\_init\_\_.py** helps to import the module to the main file called **packaged\_dag.py** from **helpers.py**
    - To import external file from different folder specify by
    - **from foldername.filename import function\_names**
    - to package a file into zip file:
      1. zip –**rm** file-name.zip file\_name.py functions/ (it is folder)
      2. zip –rm **packaged\_dag.zip packaged\_dag.py functions/**
    - It will zip all the files and folders
    - **From airflow.operators.python import** **ShortCircuitOperator -** in Apache Airflow is used to conditionally **skip** **downstream** **tasks** based on a **Python** **callable**. If the Python callable returns **true**, the downstream tasks are skipped; otherwise, they are executed as **usual**.
    - From ocm\_airflow.timezone\_helper import **LA**\_**TIMEZONE** as local\_tz - representing the timezone for **Los Angeles**.
    - Well, do you remember the timeout parameters? That’s why **workers** are refreshed. If a worker **timeout** for any reasons, **Airflow** will start a new one in order to keep the same number of running **workers** as defined in **airflow.cfg**. By default, every **worker\_refresh\_interval**, the web server creates a new worker and because the number of workers becomes greater than the number of workers defined in the parameter workers, the oldest worker is stopped.

