

There is the meta database or the meta store.

This is typically a database compatible with SQL alchemy such as Postgres, MySQL, Oracle, DB and

so on, where all the metadata related to your tasks, data pipelines and so on will be stored.

You need a database to run airflow.

The trigger is in charge of managing the deferrable operators.

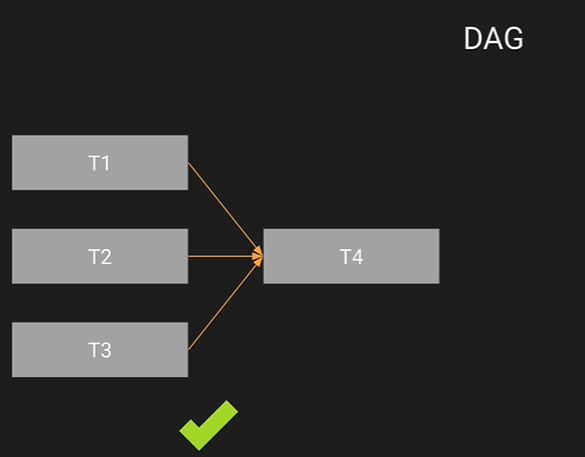
If you don't know what is a deferrable operator, think of it as a task that has this ability to suspend itself and to resume itself.

The executor doesn't execute your tasks.

The worker executes your tasks, not the executor.

The executor defines how and on which system to execute your tasks.

Whereas the worker, which can be a machine or a process, executes your tasks



A Dag means directed acyclic graph and basically it's the data pipeline in the context of airflow.

On the left you have a Dag, for example, the nodes are the tasks T1, T2, T3, T4, the directed

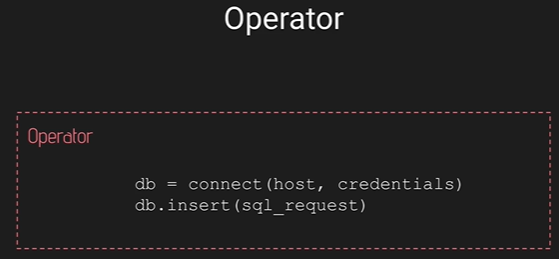
edges are the dependencies between the tasks.

You can see that T4 depends on T1, T2, T3 and finally, there is no cycle in this data pipeline.

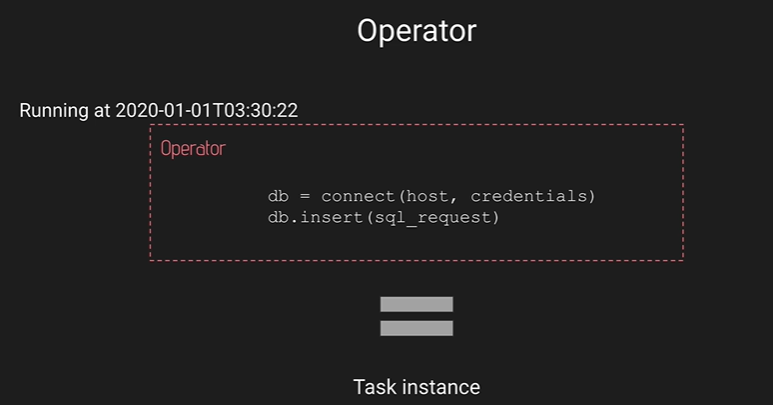
A task is an operator. Whenever you want to create a task in airflow, you will use an operator.

An operator encapsulates the logic that you want to achieve in this task.

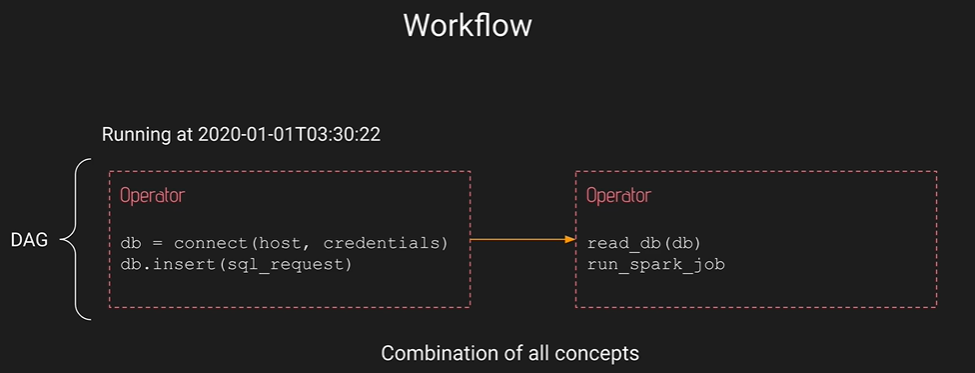
For example, you have the action operators to execute a python function with the python operator, a bash script with the bash operator or a SQL request with the Postgres operator.







So with those operators, as soon as you trigger a task, that task becomes a task instance. Basically, a task instance is an instance of an operator with a specific date.

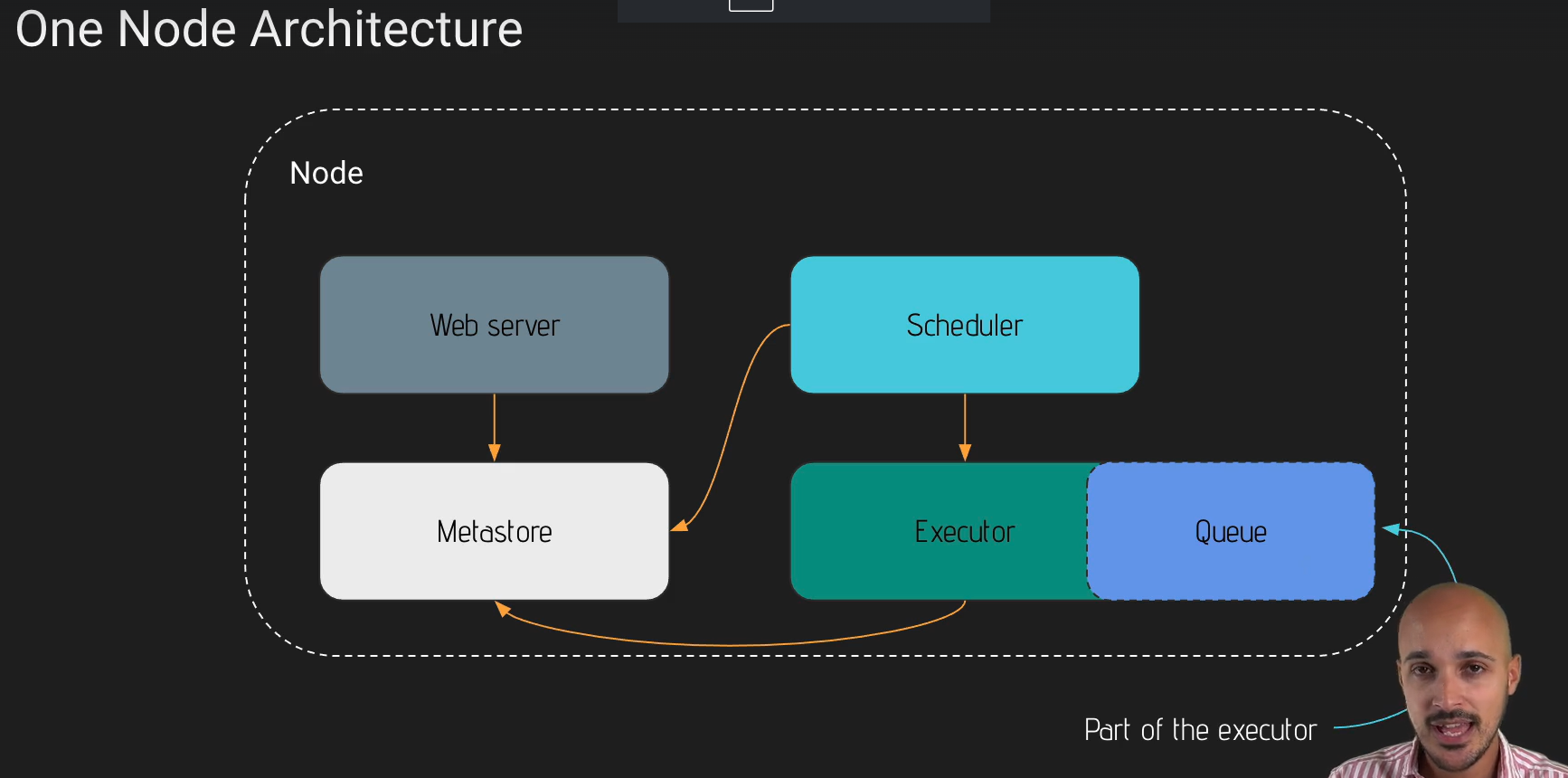


A work flow is a data pipeline, and a data pipeline is a Dag in airflow.

WHAT AIRFLOW IS NOT.

Airflow is not a data streaming solution or a data processing framework.If you need to process your data, don't use airflow.Airflow should be used as a trigger.Typically, you have spark to process your data and you will use airflow to trigger the spark job that processes your data, but you don't process your data in airflow.

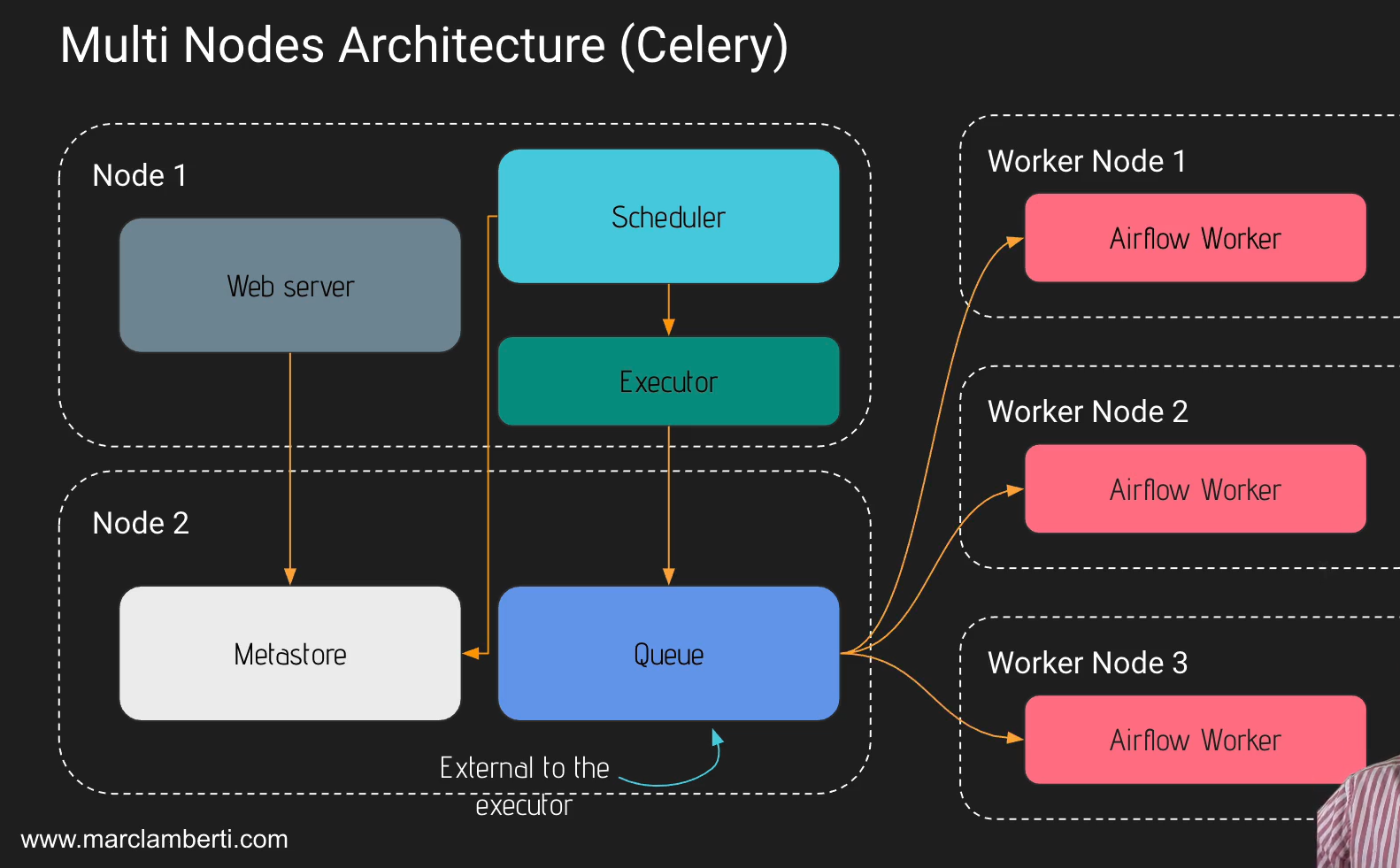
AIRFLOW ARCHITECTURE



The Web server fetches some metadata from the meta database of effort in order to display information corresponding to your Dag's, your task instances or your users on the user interface.

The scheduler interacts with the meta database and the executer in order to trigger your Dag's, in order to trigger your tasks. Finally, the executer interacts also with the meta database in order to update the tasks that just have been completed.

However if we need to scale operations, then we need to use the multimode architecture.

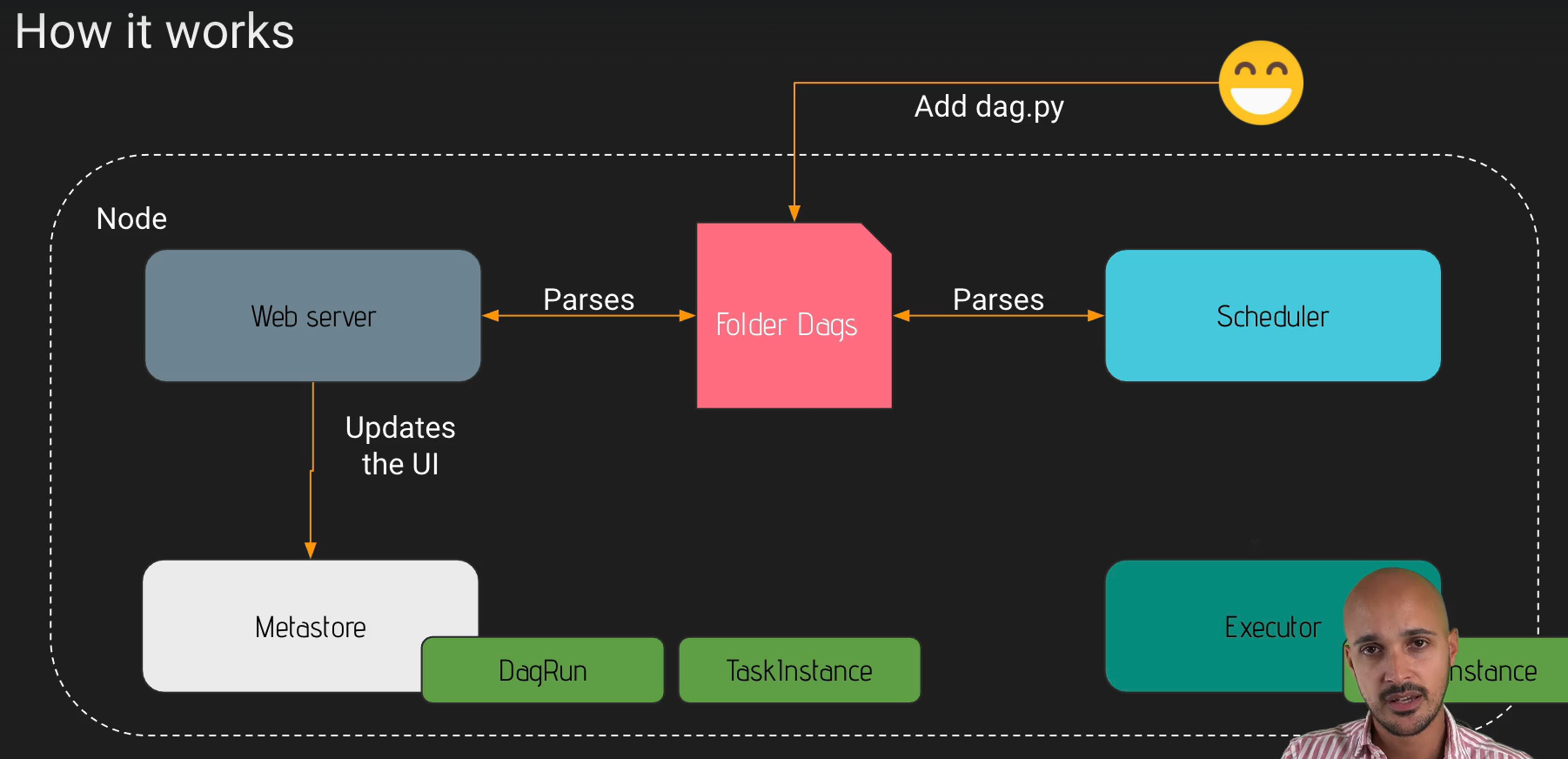


You have two nodes.

The first one contains some components of airflow, such as the webserver, the scheduler and the executor.

We also have as second node again for the components of airflow- the Meta-database and Queue. This Queue is external to the executor.

Q will be used in order to spread the work among multiple machines and among multiple Walker nodes. In each walker node you will have another component of airflow where the tasks will be executed.



Just remember that we have different statues that are given to our tasks and both the Web server and the scheduler parse our DAG.

Both the webserver and scheduler parse your DAGs. You can configure this parsing process with different configuration settings.

With the Scheduler:

min\_file\_process\_interval

Number of seconds after which a DAG file is parsed. The DAG file is parsed every min\_file\_process\_interval number of seconds. Updates to DAGs are reflected after this interval.

dag\_dir\_list\_interval

How often (in seconds) to scan the DAGs directory for new files. Default to 5 minutes. Those 2 settings tell you that you have to wait up 5 minutes before your DAG gets detected by the scheduler and then it is parsed every 30 seconds by default.

With the Webserver:

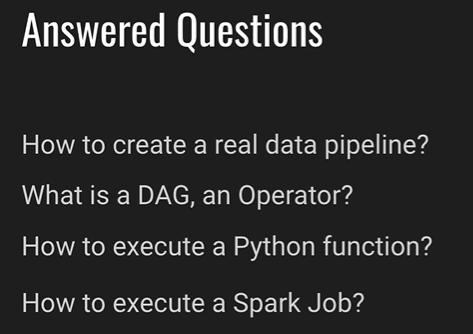
worker\_refresh\_interval

Number of seconds to wait before refreshing a batch of workers. 30 seconds by default. This setting tells you that every 30 seconds, the web server parses for new DAG in your DAG folder.

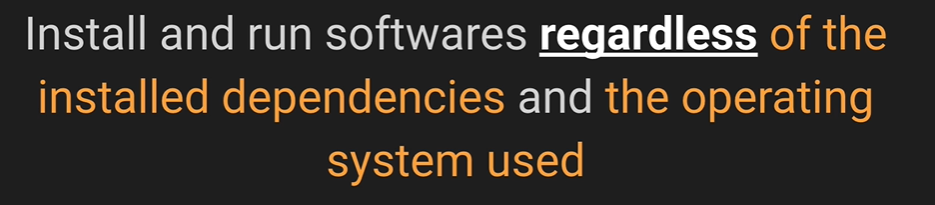
Remember

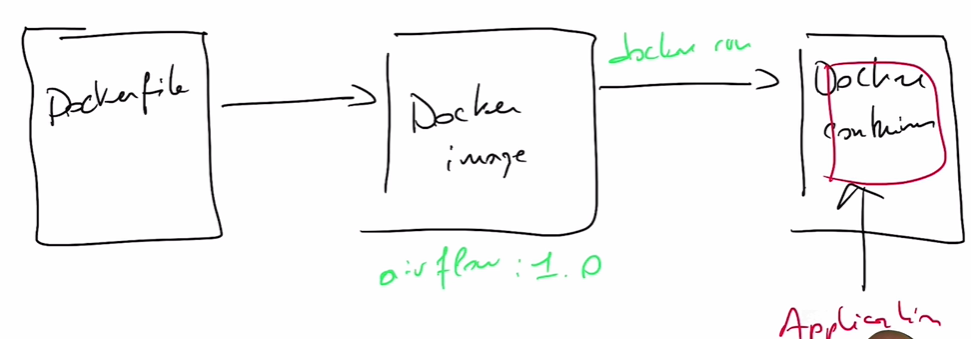
By default, when you add a new DAG you will have to wait up to 5 minutes before getting your DAG on the UI and then if you modify it, you will have to wait up to 30 seconds before getting your DAG updated.

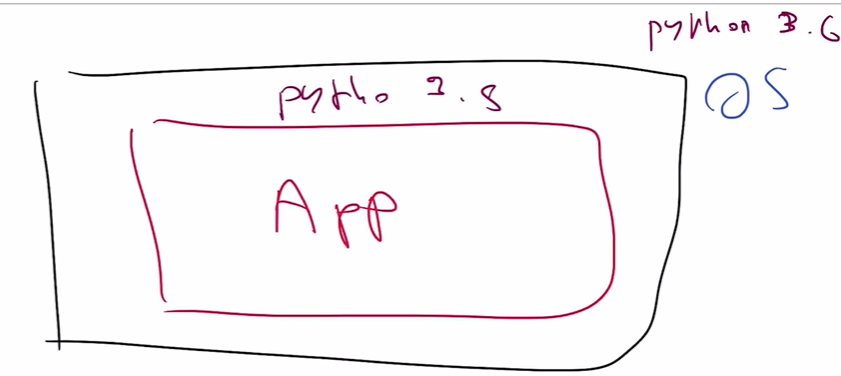
PROJECT



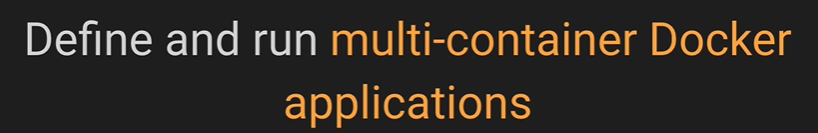
WHY DOCKER?

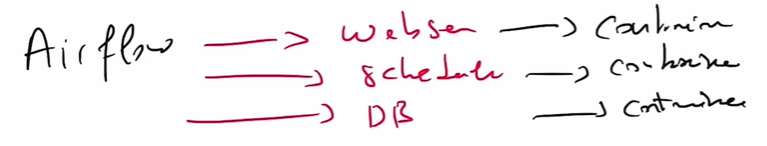






DOCKER COMPOSE



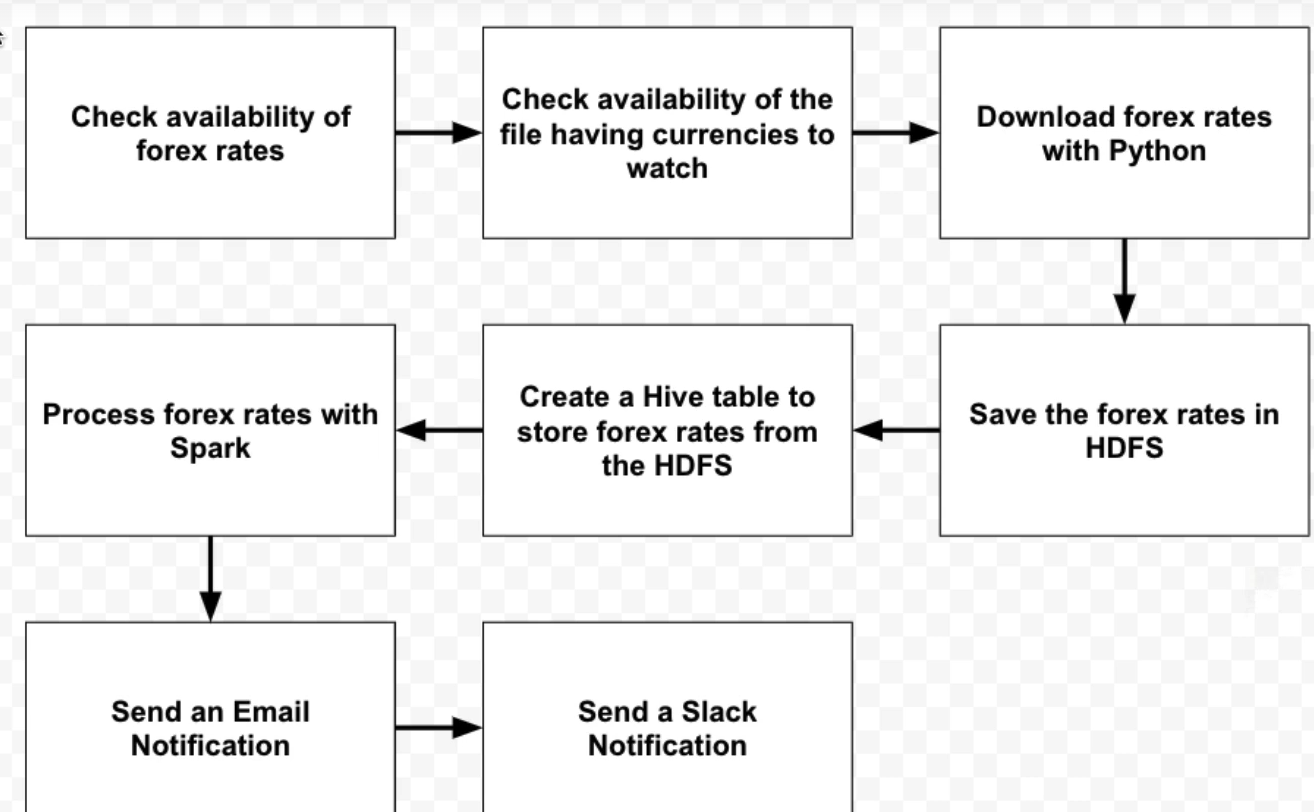


Airflow has multiple components. So each of these components is going to be docker container, so you will run one container for the webserver, one for the scheduler and one for the metastore.

We don't want to run everything into a single container.

Because if the website ever fails, then you will have to restart not only the Web server, but also the scheduler and the database.

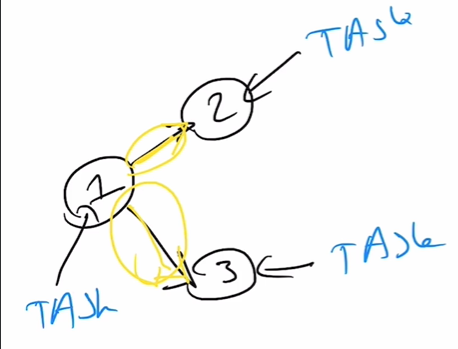
Flow Chart of Data Pipeline



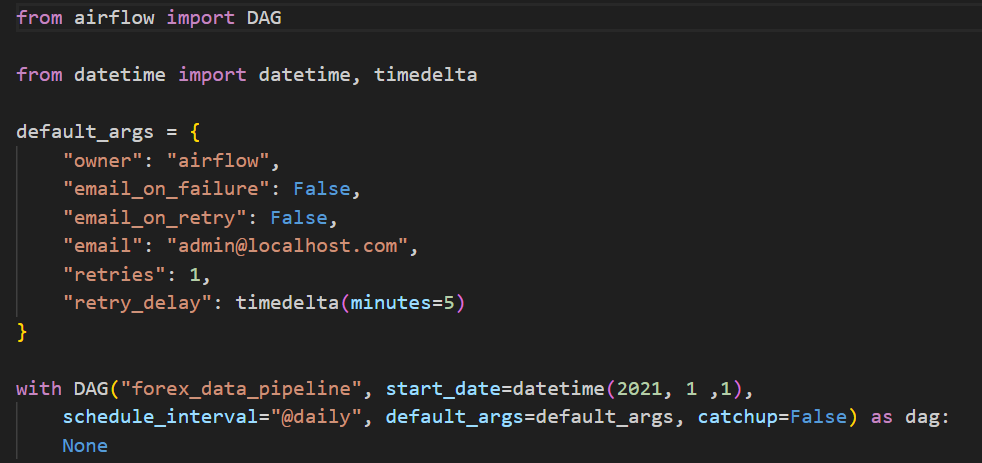


WHAT IS DAG





Edges are the dependencies between the tasks



First, you need to import a dag object, then a datetime object, as well as the timedelata if we use the timedelta object.

Then we can specify a dictionary for dags corresponding to the different arguments

that we want to apply to all of our tasks.

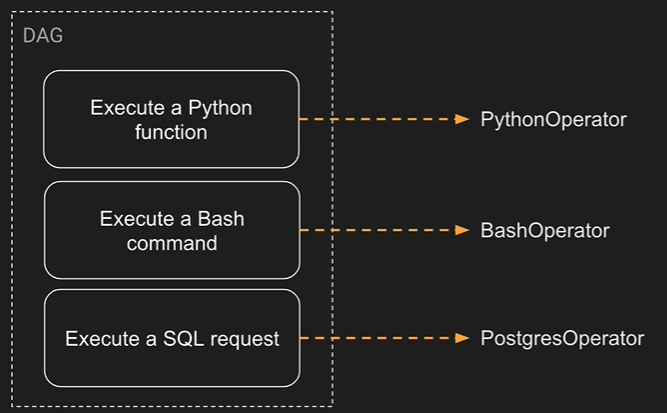
Then you instantiate a dog object with a unique dag id, star\_date to indicate when your dag is going to be scheduled, schedule\_interval to indicate the frequency at which your dag is going to be triggered. And finally, you apply the different arguments.

catchup=False

If you create this DAG on January 15, 2023, with catchup=False, it will not attempt to run any tasks for the period from January 1, 2021, to January 14, 2023, even if there were scheduled runs during that time. It will only execute tasks for dates starting from January 15, 2023, and going forward. Setting catchup to False is often useful when you want to avoid running a large number of backfill or historical tasks when creating or updating a DAG.



Operator is an object encapsulating the logic of the job that you want to execute.



3 Types of Operators

Action

Actual operators are nothing more than operators allowing you to execute something. For example, you want to execute a python function. You use the python operator.

Transfer

Transfer operators allow you to transfer data from a source to a destination. So basically with the transfer operator, you don't have to implement the logic to move data between the source and destination by yourself.

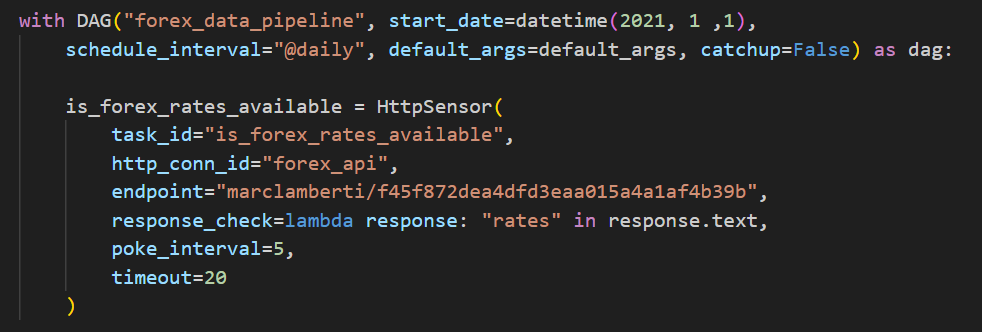
Sensor

The sensor operators allow you to wait for something to happen before moving to the 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 task.

HTTP SENSOR





Using HTTP Sensor to verify whether the url works or not.

is\_forex\_rates\_available = HttpSensor(...) This section defines an HttpSensor task with the following properties:

task\_id="is\_forex\_rates\_available": Each operator must have unique task id in the same dag.

http\_conn\_id="forex\_api": It specifies the Airflow connection to use. The connection named "forex\_api" should be configured in Airflow and contain the necessary HTTP connection details.

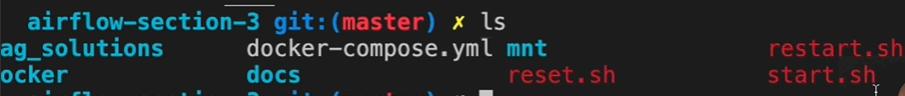
endpoint="marclamberti/f45f872dea4dfd3eaa015a4a1af4b39b": This is the endpoint (URL) that the sensor will poll. In this case, it seems to be a GitHub Gist URL or a similar web resource.

response\_check=lambda response: "rates" in response.text: The response\_check parameter defines a function that checks the HTTP response. In this case, it checks if the string "rates" is present in the response text. The sensor will keep polling the endpoint until this condition is met.

poke\_interval=5: This sets the time interval (in seconds) between each polling attempt by the sensor. Here we are doing it after every 5 seconds.

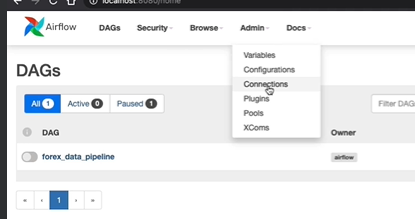
timeout=20: The timeout parameter specifies the maximum time the sensor will wait for the HTTP request to complete (in seconds). Always specify the timeout otherwise the sensor will keep running for 7 days.

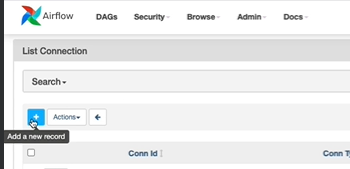
Run Airflow

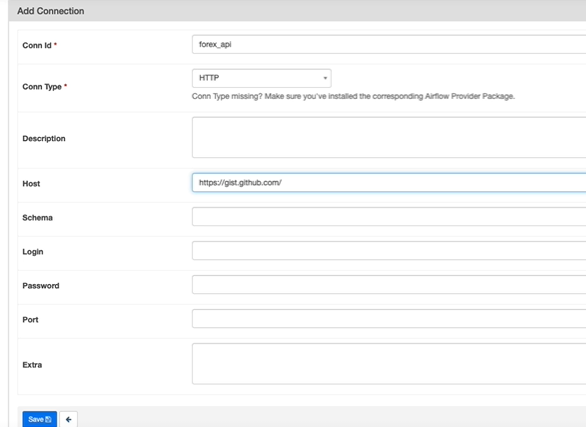




(The above command will take 30 min to run…)





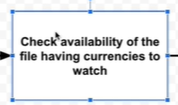




To run our container



The above command tests the task



The file sensor sensor will check every 60 seconds by default if a file or a folder

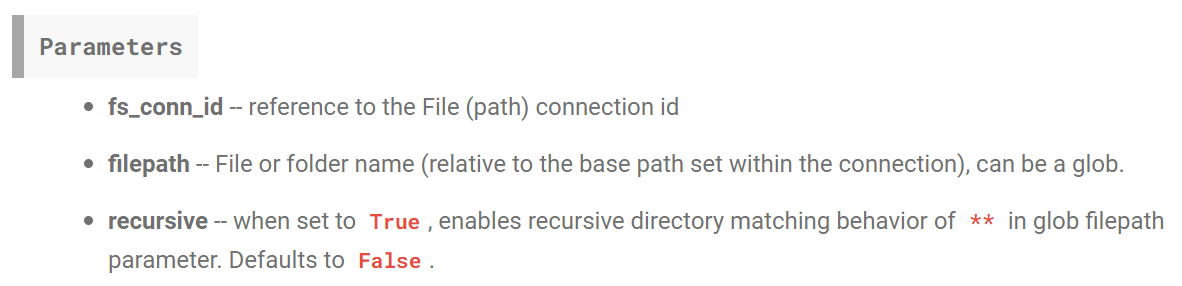
exists at a specific location in your system.

All the sensors-

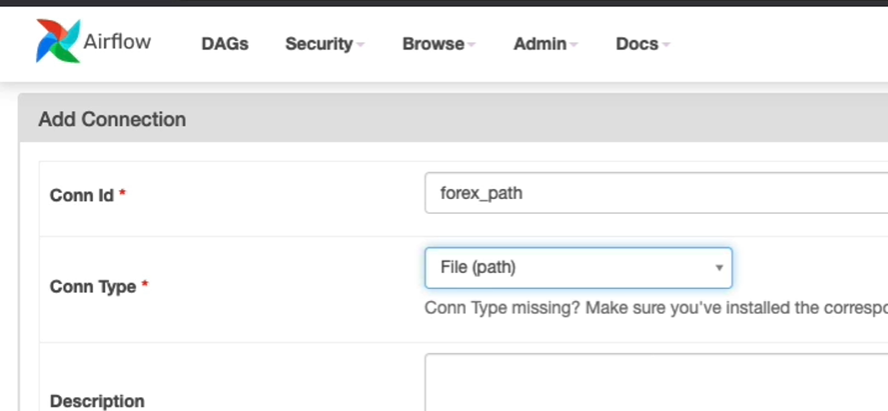
<https://airflow.apache.org/docs/apache-airflow/2.3.0/_api/airflow/sensors/index.html>

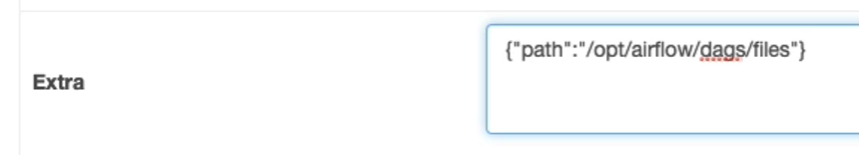
File Sensor-

https://airflow.apache.org/docs/apache-airflow/2.3.0/\_api/airflow/sensors/filesystem/index.html

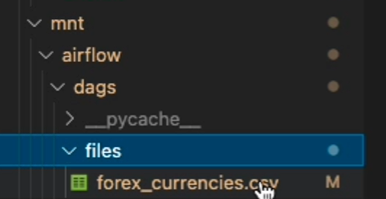


Make a new connection using airflow UI



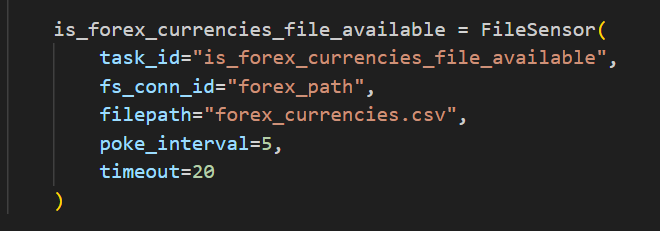


Path of where we are looking for the filw will be. It is JSON



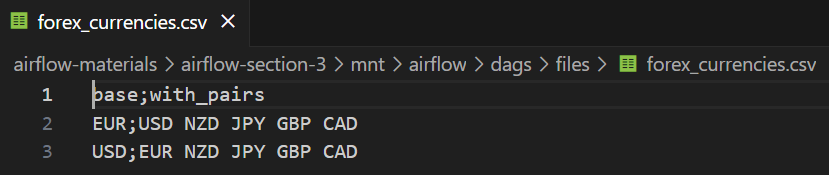
We are checking if this file exists or not



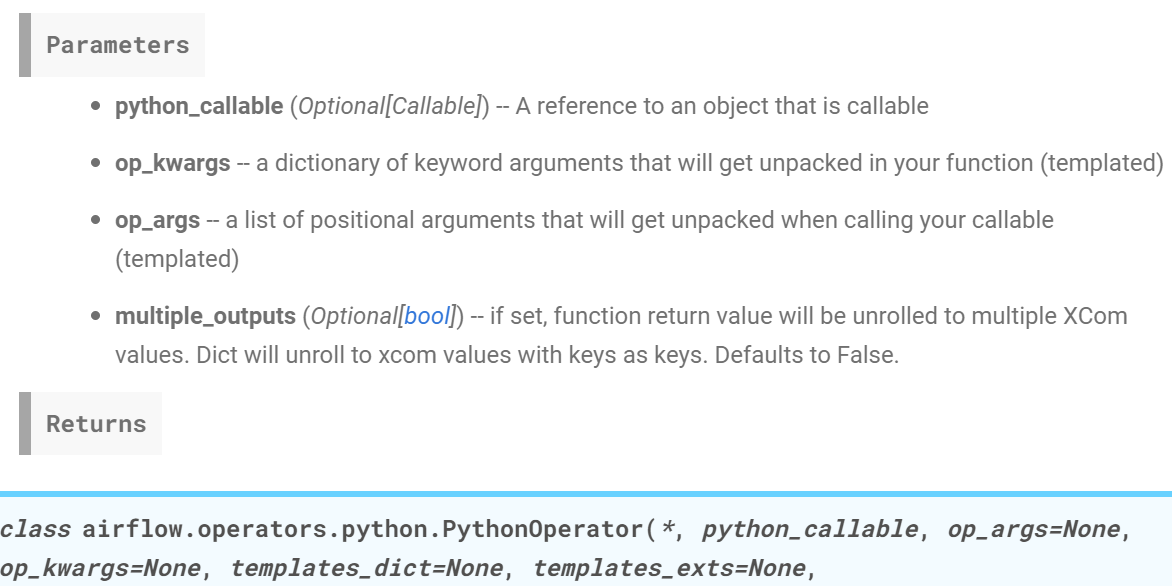




Python Operator

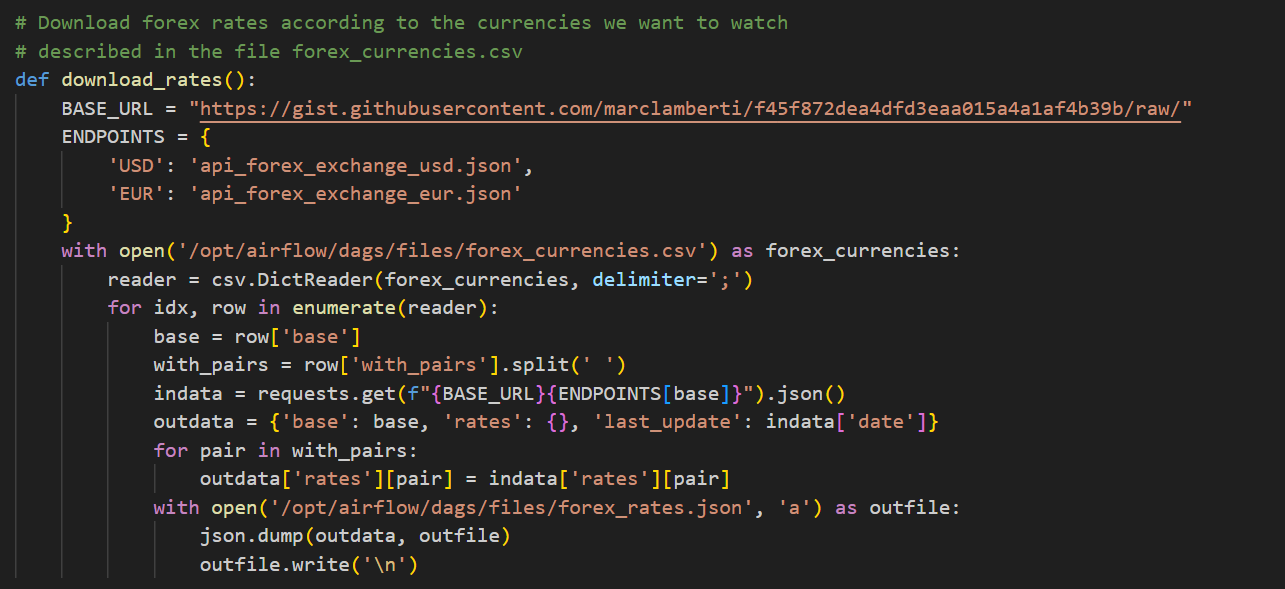


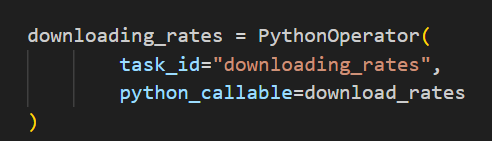
File which we already had.



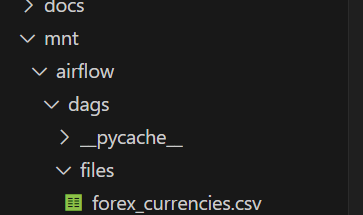
<https://airflow.apache.org/docs/apache-airflow/2.3.0/_api/airflow/operators/python/index.html>



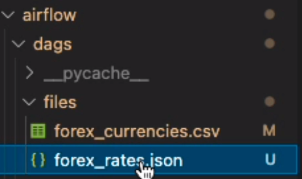






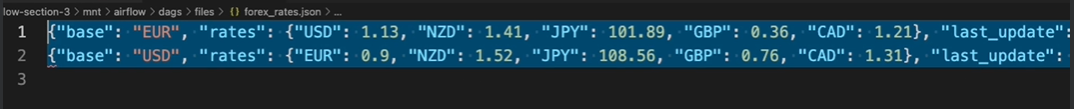


Before



After.

A file has now been created named forex\_rates.json

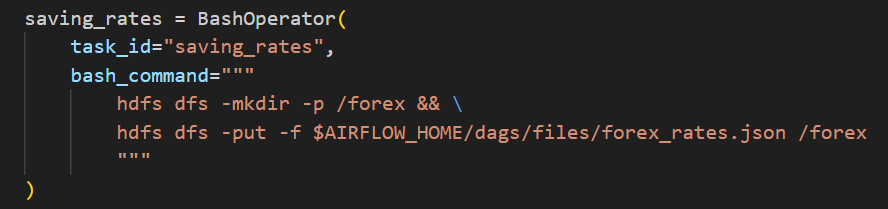


In real world, this file can be terabytes in size. So we would I like to store it in HDFS.

Storing file in HDFS





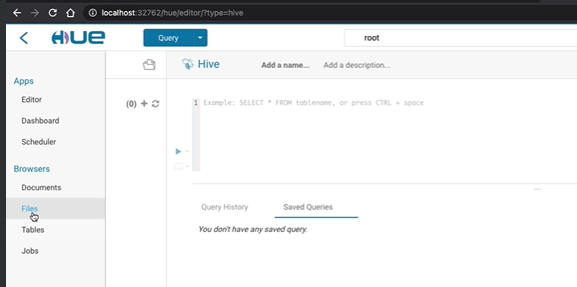


1. hdfs dfs -mkdir -p /forex: This is a command to create a directory named "/forex" in HDFS (Hadoop Distributed File System). The -p option ensures that any necessary parent directories are also created if they don't exist.

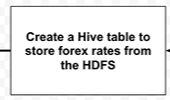
2. hdfs dfs -put -f $AIRFLOW\_HOME/dags/files/forex\_rates.json /forex: This is another HDFS command. It copies a file named "forex\_rates.json" from a local directory specified by the $AIRFLOW\_HOME environment variable to the "/forex" directory in HDFS. The -f option forces the file to be overwritten if it already exists in HDFS.

By using **&&**, the second command will only execute if the first command is successful. If the directory creation (first command) fails, the second command won't execute, ensuring that you don't attempt to copy a file to a non-existent directory.

The backslash **\** at the end of the first line is used for line continuation. It allows you to break a long command into multiple lines for better readability. In this case, it's not strictly necessary, but it's used to format the code neatly. If you removed the **\**, the two commands would be on the same line, which can make the code less readable.



Hue is a nice tool to interact with the HDFS  

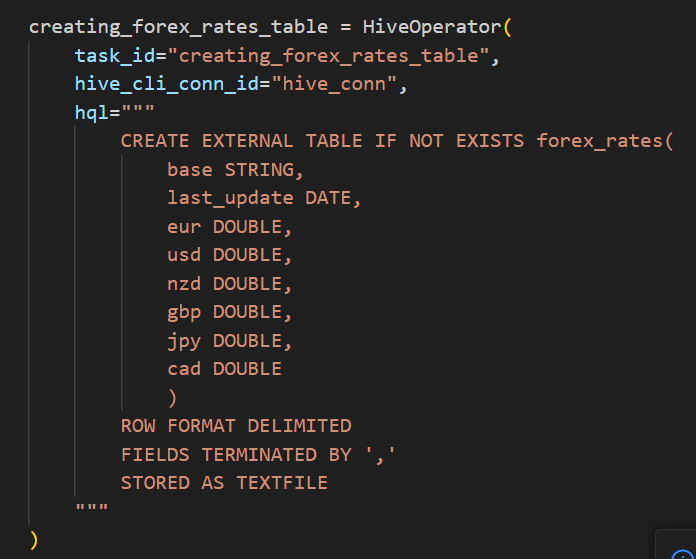



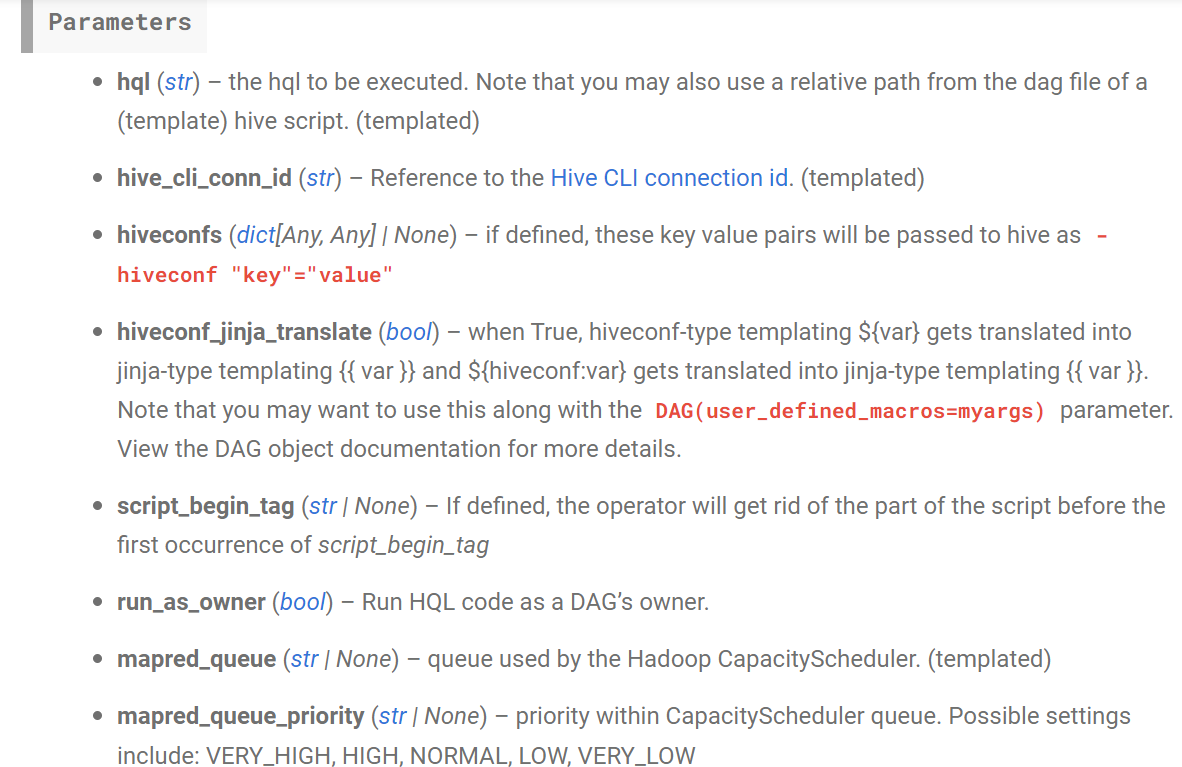
Data is stored as a plain text file. We can use Hive. We are going to create a table on top of our data, on top of your files so that we will be able to query our data by executing sql like queries.

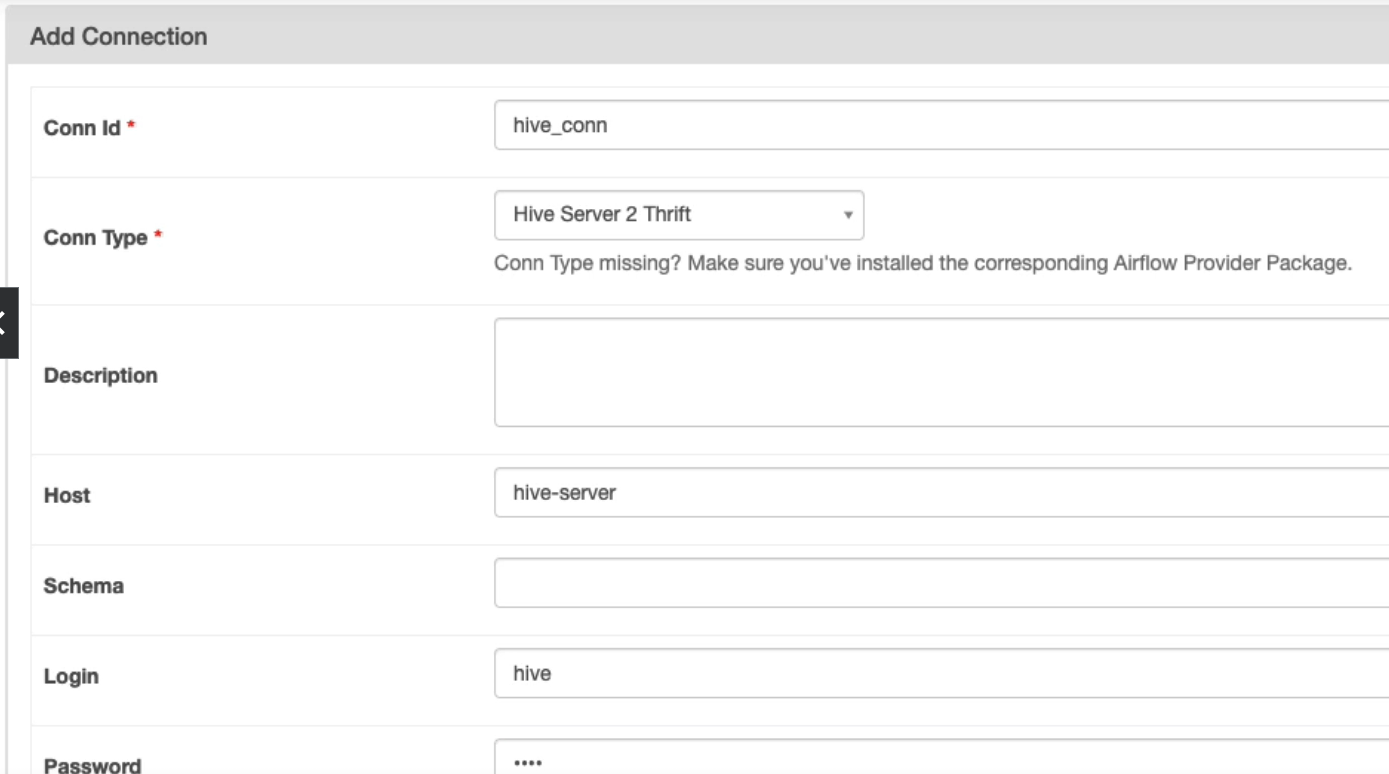
IMPORT HIVE OPERATOR

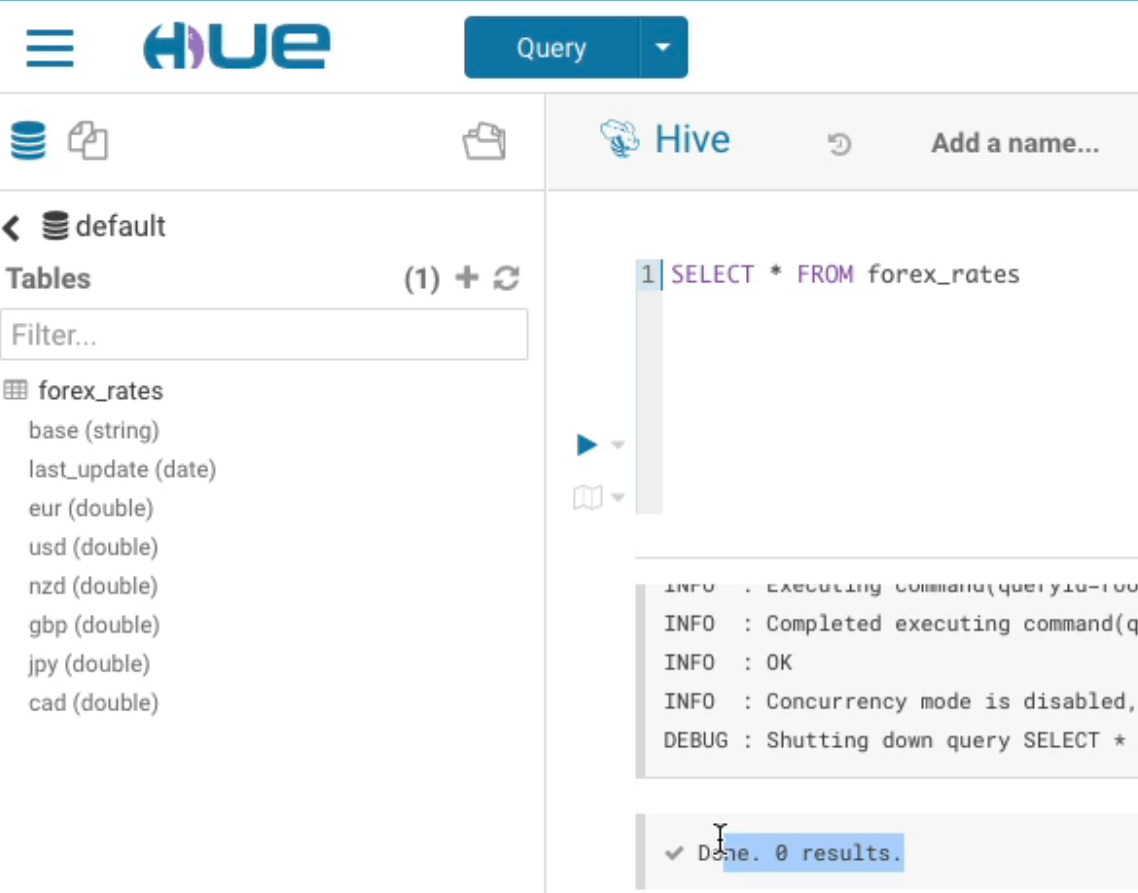
<https://airflow.apache.org/docs/apache-airflow-providers-apache-hive/stable/_api/airflow/providers/apache/hive/operators/hive/index.html>

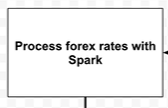






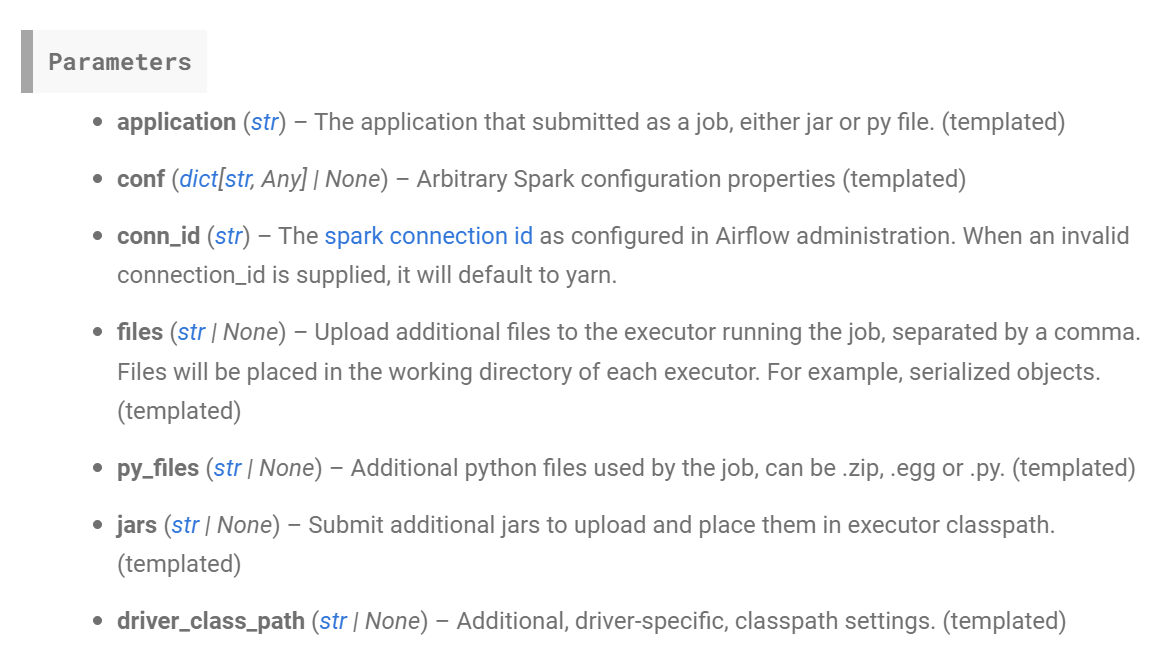


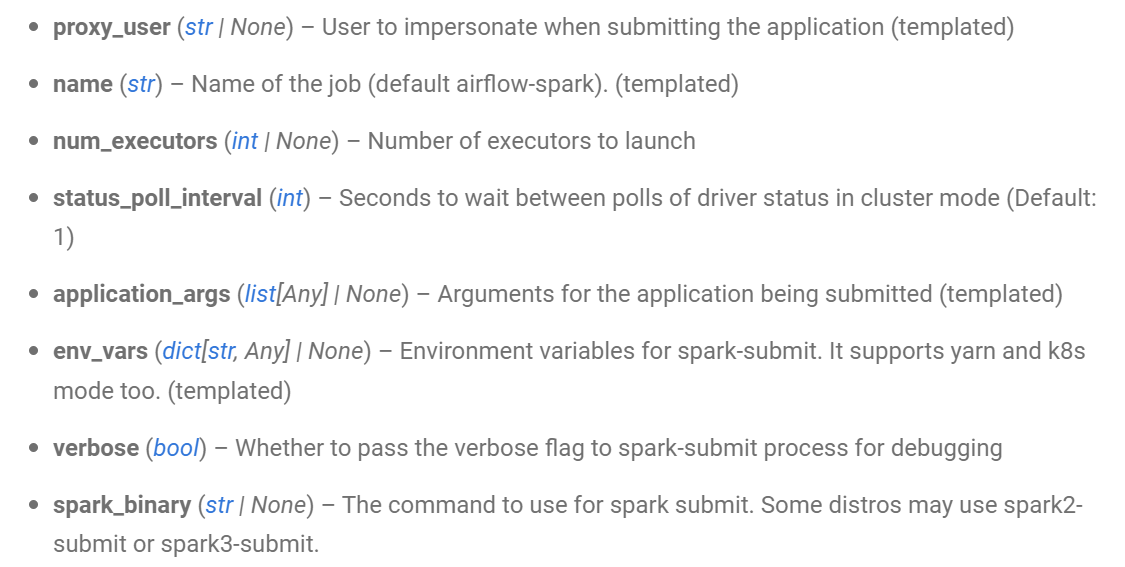




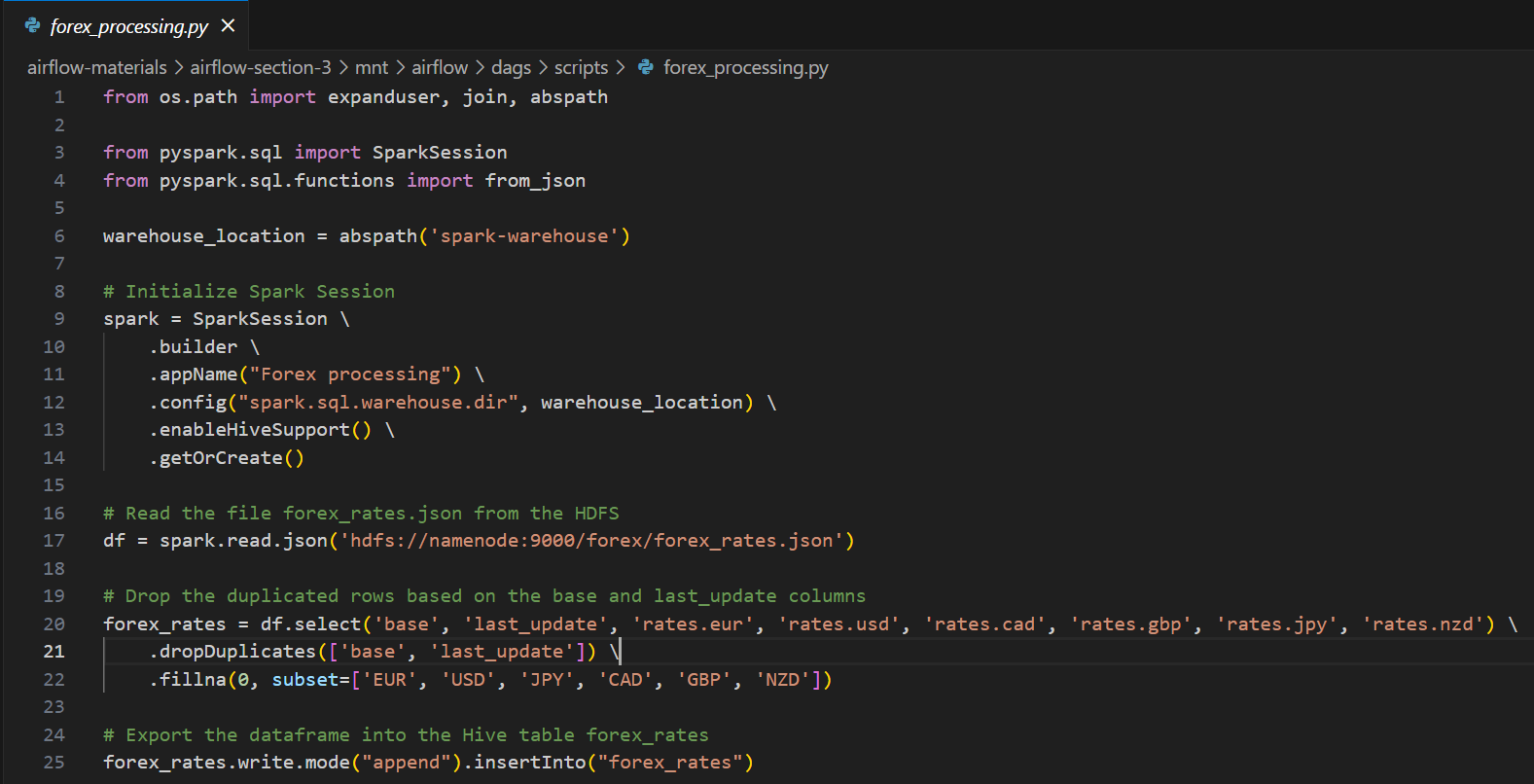
Process in an orchestrating tool so don’t process terabytes of data. Use spark.

<https://airflow.apache.org/docs/apache-airflow-providers-apache-spark/stable/_api/airflow/providers/apache/spark/operators/spark_submit/index.html>

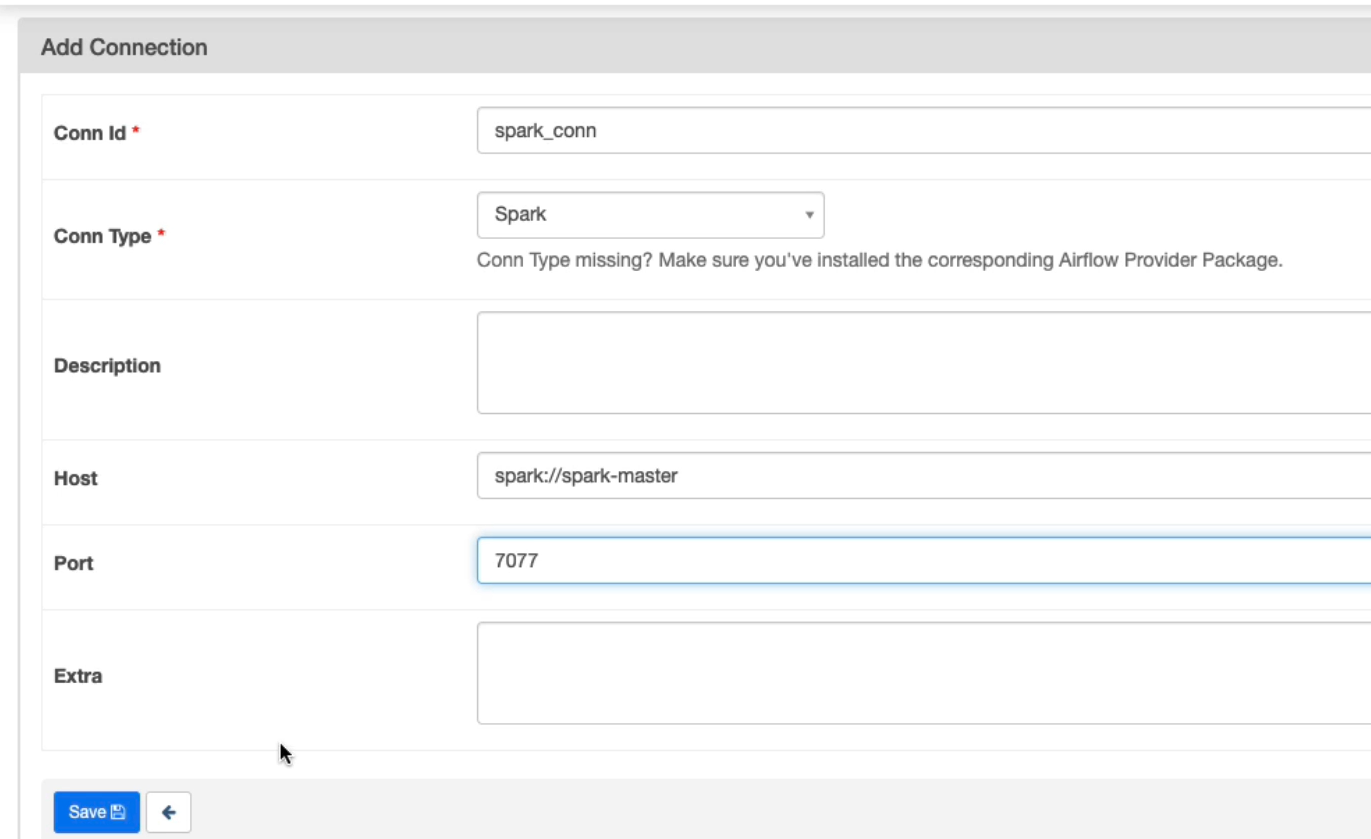


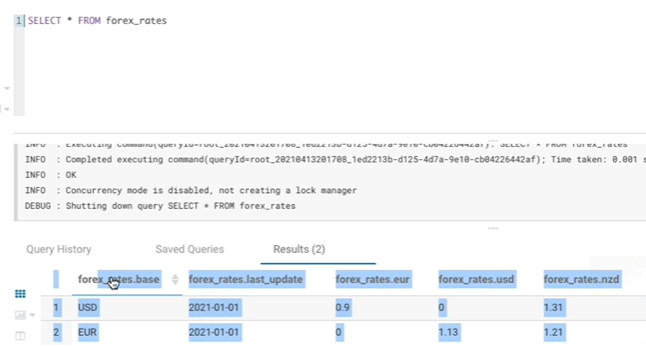


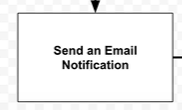


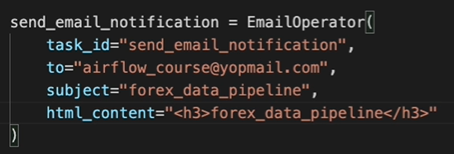


1. Importing Libraries:
   * The script starts by importing necessary libraries.
   * **expanduser**, **join**, and **abspath** are used for file path manipulation.
   * **SparkSession** is imported from the **pyspark.sql** module, which is the entry point for using Spark SQL.
   * **from\_json** is imported from **pyspark.sql.functions** to handle JSON data.
2. Warehouse Location:
   * It defines the warehouse location by using the **abspath** function to get an absolute path to a directory named "spark-warehouse." This directory is used for Spark metadata.
3. Initialize Spark Session:
   * The script initializes a Spark session using **SparkSession.builder**.
   * It sets the application name to "Forex processing" and specifies the warehouse directory using the **config** method.
   * **enableHiveSupport()** is called to enable Hive support in Spark.
   * Finally, it gets or creates a Spark session.
4. Read JSON Data:
   * The script uses the Spark session to read JSON data from an HDFS location (Hadoop Distributed File System) specified as 'hdfs://namenode:9000/forex/forex\_rates.json.'
   * The JSON data is read into a DataFrame named **df**.
5. Data Transformation:
   * The script performs several operations on the DataFrame **df** to clean and reshape the data:
     + It selects specific columns: 'base', 'last\_update', and exchange rates for 'eur', 'usd', 'cad', 'gbp', 'jpy', and 'nzd.'
     + It removes duplicate rows based on the 'base' and 'last\_update' columns using the **dropDuplicates** method.
     + It fills missing values with 0 in the selected columns ('EUR', 'USD', 'JPY', 'CAD', 'GBP', 'NZD') using the **fillna** method.
6. Export to Hive Table:
   * The cleaned and transformed DataFrame, **forex\_rates**, is written to a Hive table named "forex\_rates."
   * It uses the **write** method with mode "append" to insert the data into the Hive table.













This is exactly same.



However instead of set\_upsteam and set\_downstream, we can use this-





Now this line can be extremely long so-

