Airflow Implementation Documentation

Version 1.0

# Airflow Code Demo

# **Basic Idea : The dag is a sequence of tasks that have dependencies between on another. In the pdill dag we have http operators that trigger the jobs and then we have sensors to track the progress of the job. These sensors halt the execution of the dag until the underlying job has completed and only proceeds if the job results in a success**

**from datetime import datetime**

**from airflow import DAG**

**import json**

**from airflow.operators.http\_operator import SimpleHttpOperator**

**from airflow.operators.python\_operator import PythonOperator**

**from airflow.sensors.http\_sensor import HttpSensor**

Firstly, We import the required libraries such as the DAG class and other operators which are required for the development.

We can also import custom libraries from custom packages here.

**Summary - Importing libraries and operators and custom fuctionsq**

**default\_args = {**

**'description':'TDMF data masking',**

**'start\_date': datetime(2020,9,25),**

**'email': ['prashanth.desani@wellsfargo.com'],**

**'retries': 3,**

**'email\_on\_failure': True,**

**'email\_on\_retry': False,**

**}**

**Summary: Default arguments are required and are sent to all tasks. We can set options that are redundant in all tasks here**

The default arguments are the arguments or options that will be passed to all the tasks in the Dag. This will help us avoid redundantly providing these options to each and every task separately. It is also useful if we need to make small changes in these options from time to time. We will have to make it in just one place and not having to make changes in all the tasks.

Description: This parameter holds a one-line description that will show up on the UI beside your dag name (which is defined later in the code).

Start\_date: start date is defined for the scheduler to understand from which day it is supposed to start executing this particular dag. It is helpful when you want to start the dag on a certain day post-deployment or otherwise you can set any past date. However, this is an important option. There is another option catchup (defined later in the DAG object) which goes in conjunction with this start date. The catchup option tells the scheduler if it is supposed to ‘catch-up’ on executing the historical dag runs or not. So if you set a historical date say 5 days ago from the date of deployment. The scheduler will create 5 dag runs and the next required run when the scheduled interval arrives

Email: The email option defines a list of emails that the task is supposed to send the emails to.

Retries: The number of times the task should retry before giving out a failure status

Email\_on\_failure: This option tells the scheduler if it is supposed to email on the failure of a task

Email\_on\_retry: each task also needs retry options. The task then retries for the specified number of retries and this option just tells the scheduler if it is supposed to send an email to the email list or not.

**configuration\_id = ""**

**pdill\_datamasking\_bkp='pdill\_datamasking\_bkp'**

These are initialization variables just created for passing them to the tasks.

**def resolve\_config\_id(\*\*kwargs):**

**task\_instance = kwargs['task\_instance']**

**configuration\_dict = json.loads(task\_instance.xcom\_pull(task\_ids='data\_extraction', key='return\_value'))**

**configuration\_id = configuration\_dict['configurationId']**

**return configuration\_id**

**Summary - This is a Custom Function that helps us resolve the configuration ID. You can define any number of functions and call them through the Python Operator**

The resolve\_config\_id function resolves the configuration\_id for the consequent steps in the dag. It pulls the return value from the starting task and then pushes it as a singleton value in the Xcom to make it easily accessible to the consequent tasks.

Xcom is a common Dictionary available through all the tasks in the dag. This helps the tasks to talk to each other and pass variables and make decisions on the outcome of the previous

Note: the tasks of a Dag are rendered every time a dag run is created. Hence some parameters which are dynamic need to be passed in the ‘Jinja Templating’ syntax. Jinja Template Engine is a web page rendering engine that resolves native python variables to our tasks. You can find them enclosed in ‘{{ … }}’ these braces. It is also important when you want to use variables that are passed on runtime. These runtime variables are attached to the dag\_run object which is only available during runtime.

**def monitor\_job(response):**

**print('###################################monitor process response ##################')**

**print(response.text)**

**responseJson=json.loads(response.text)**

**jobStatus=responseJson["status"]**

**if jobStatus == "FAILED":**

**raise AirflowException("Job FAILED.")**

**else:**

**return jobStatus=="SUCCESS"**

**Summary : This is also a custom function that is invoked by the HttpSensor. It processes the response received and gives either True or False based on condition.**

This function is defined to be called in the response check part of the HttpSensors (defined later). It takes in a response dictionary and then prints out the entire message in the log giving us a better idea of all the things the response contained.

**dag = DAG('pdill\_datamasking', schedule\_interval=None, catchup=False, default\_args=default\_args)**

**Summary : The dag object is created here with the required parameters.**

This line creates the dag object to which all the tasks will be attached and execute.

The first argument is the name of the dag (which will show up on the UI),

next is the scheduled interval which defines the schedule of the dag in the ‘cron’ format

Next is the catchup option to tell the scheduler if historical dag runs are to be created or not

Lastly, we have the default\_arguments dictionary wherein we pass the default arguments that are needed in all the tasks and we do not want to keep passing them in each one separately

**post\_data = {**

**"app": "CNAPP",**

**"project": "SS",**

**"sor": "msp",**

**"datasetName": "CHARMING\_17",**

**"date": "2021-05-30"**

**}**

This is a data dictionary sent to the POST endpoint at the start of the execution. It is hardcoded as of now but will be parameterized

**datamasking\_pipeline\_execution=SimpleHttpOperator(**

**task\_id='datamasking\_pipeline\_execution',**

**http\_conn\_id=pdill\_datamasking\_bkp,**

**endpoint='/datamasking/execution',**

**headers={"Content-Type": "application/json"},**

**data=json.dumps(post\_data),**

**response\_check=lambda response: response.json()['configurationId'] != None,**

**xcom\_push=True,**

**log\_response=True,**

**dag=dag**

**)**

**Summary: This the the SimpleHttpOperator, that helps us send Post and get requests to the api**

Now creating a task in Airlfow means just invoking the right operator object you want with the right options. Here we invoke the SimpleHttpOperator which we imported at the beginning and provide the right options to it.

The task\_id defines the name of the task. It should be unique throughout the DAG.

The next is the http\_conn\_id. This is the connection id that we define in the ‘Connections’ section of the DAG UI. This helps us decouple secret information from the DAG’s codebase.

The next is ‘endpoint’ which is the endpoint we’re trying to hit. Note that it does not have the hostname attached to it.

The headers are the headers to be passed with your request.

Data indicates any data you want to pass with the request. In our case, it is the post\_data dictionary that we have defined earlier.

Moving on we have the response\_check option it checks for the required success criteria and only gives a success output if it is met otherwise the task returns a failed state.

Xcom\_push value indicates if the return\_value is supposed to be pushed to the xcom or not.

Log\_response value indicates if the task is supposed to log the entire response or not.

Lastly, the dag option indicates which dag this task belongs to

**get\_configurationId = PythonOperator(**

**task\_id='get\_configurationId',**

**dag=dag,**

**python\_callable=resolve\_config\_id,**

**provide\_context=True**

**)**

**Summary: this is a python operator that resolves the configuration ID and pushes then to Xcom.**

This is a python operator that helps us execute custom python functions. Here we are using it to invoke resolve\_config\_id function that we have defined.

**datamasking\_extraction = SimpleHttpOperator(**

**task\_id='datamasking\_extraction',**

**http\_conn\_id=pdill\_datamasking\_bkp,**

**endpoint="/datamasking/extraction/{{ task\_instance.xcom\_pull(task\_ids='get\_configurationId', key='return\_value') }}",**

**method='GET',**

**response\_check=lambda response: response.json()['status']=="SUCCESS",**

**log\_response=True,**

**dag=dag**

**)**

**Summary: this is again a Simple Http Operator that fires a get request to the datamasking/extraction endpoint.**

**Note : Response check in Simple Http Operator pertains to that request only. This means the operator only triggers the job and then exits with success when the response\_check condition is satisfied. This is why we use a sensor to track progress**

This is similar to the previous SimpleHttpOperator but with a few differences.

The first one is that we attach a configuration\_id dynamically based on what we receive from the spark UI.

The next is the ‘method’, we specify it as ‘GET’. If we do not specify this value it defaults to ‘POST’.

**datamasking\_extraction\_jobstatus = HttpSensor(**

**task\_id='datamasking\_extraction\_jobstatus',**

**http\_conn\_id=pdill\_datamasking\_bkp,**

**endpoint="/datamasking/jobstatus/{{ task\_instance.xcom\_pull(task\_ids='get\_configurationId', key='return\_value') }}",**

**method='GET',**

**request\_params={},**

**response\_check=lambda body:monitor\_job(body),**

**#response\_check=lambda response: response.json()['status']=="SUCCESS",**

**poke\_interval=5,**

**dag=dag**

**)**

**Summary : the http sensor pokes the tracking endpoint after the set interval of 5 seconds. This interval can be changed based on our requirements.**

This is a HttpSensor. We have imported this at the start and it is used to keep poking any endpoint until we get the required response from the server. So here we have an endpoint that pokes with the interval of 5 seconds and it will keep poking it until we get the ‘status’ key in the response as ‘SUCCESS’

.

.

.

.

.

.

.

**datamasking\_pipeline\_execution >> get\_configurationId >> datamasking\_extraction >> datamasking\_extraction\_jobstatus >> datamasking\_execution >> datamasking\_execution\_jobstatus >> datamasking\_validation >> datamasking\_validation\_jobstatus >> datamasking\_ndm >> datamasking\_ndm\_jobstatus >> datamasking\_ingestion >> datamasking\_ingestion\_jobstatus >> datamasking\_complete**

**Summary : The above is the hierarchy that we have defined. The hierarchy can be any way we want. It can have branches and also conjunctions. In our case, we did not need any of them so we have a pretty simple hierarchy. The ‘>>’ operator defines a dependency between the tasks.**

Where are the logs stored?

The logs are stored in the form of text files on the server where airflow is running. It has a specific folder structure while storing the logs. The logs are stored in the following way.

Logs

Dagname

Taskname

<retrynumber>.txt

Although the same is available in the UI in the logs section of the task.

How can I run the job from the point of failure?

On the UI, you must navigate to the DAG that you want to re-run. Once you’re on go to the graph view and there you can see the tasks highlighted in red as the failed tasks. To rerun the task you need to click on the task and a window pops up with a lot of options. One of them is a clear option. It also has sub-options. They are there for 2 cases. Case 1 is that you just need to rerun that one task. In this case uncheck the ‘Downstream’ and ‘Recursive’ suboptions. The other case which is most likely is that you want to clear/rerun the failed task and also all the downstream tasks. In this case remember to check the ‘Downstream’ and ‘Recursive’ options.

Performance Improvement?

Airflow’s performance largely depends on what mode it is running on and what backend it is using. Airflow has metadata tables that it uses to make execution decisions

Outstanding Questions:

How does the DAG trigger from the UI?

How does one run a DAG multiple times sequentially?

How to control DAG Runs?

How to Promote code from dev to test environment and then finally to production?

How is unit testing done?

## 

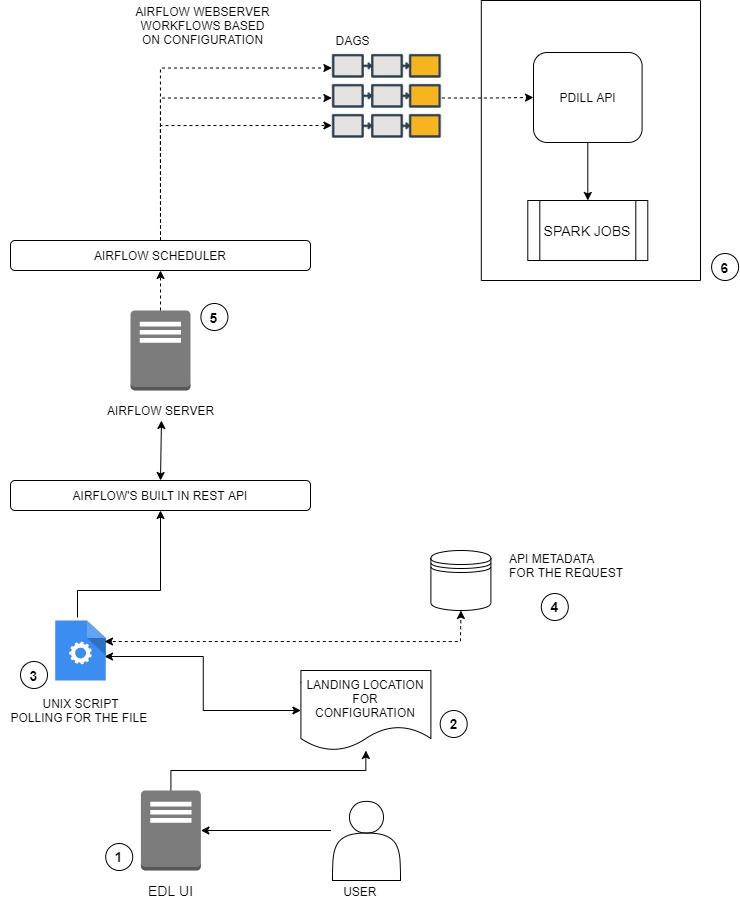
## Design

The screenshot below describes the graphical view of the Airflow Dag for PDILL

**<insert DAG screenshot>**

| **Operator Name** | **Function** | **Parameter Details** |
| --- | --- | --- |
| data\_extraction | This operator hits the endpoint for extraction of data and generates CSV files | ***task\_id:*** Name of the task that is visible on the DAG and for Xcom push and pulls.  ***http\_conn\_id:*** Name of the connection id as defined in the ‘Connections’ section of Airflow UI with auth details  ***endpoint:*** The endpoint that the operator will hit. The endpoint does not include the server name in it only the name of the route is mentioned. Airflow picks up the server name from the Connections sections as mentioned in the Airflow UI  ***headers:*** The headers for the request  ***data:*** The payload for the request in a valid JSON string  ***response\_check:*** This parameter checks the value of a particular parameter in the response to make a decision if the request was completed successfully or failed  ***xcom\_push:*** pushes to values to the Xcom  ***log\_response:*** The response of the request gets logged in the logs section of the UI  ***dag:*** The dag where the operator belongs |
| get\_configurationId | This operator is responsible for resolving the configuration id required to be appended to the ‘GET’ requests in the subsequent tasks in the DAG.  This operator just receives the configurationId from the Xcom pushed by the ***data\_extraction*** operator which is in dictionary format and pushes it to Xcom as a singleton value for it to be easily accessible by the subsequent operators | ***task\_id:*** Name of the task that is visible on the DAG and for Xcom push and pulls.  ***dag:*** The dag where the operator belongs  ***python\_callable:*** The python function to be called for the execution |
| data\_masking | This operator starts the spark job responsible for data masking.  The configurationId is received from the Xcom and appended to the endpoint making it dynamic for any configuration Id | ***task\_id:*** Name of the task that is visible on the DAG and for Xcom push and pulls.  ***http\_conn\_id:*** Name of the connection id as defined in the ‘Connections’ section of Airflow UI with auth details  ***endpoint:*** The endpoint that the operator will hit. The endpoint does not include the server name in it only the name of the route is mentioned. Airflow picks up the server name from the Connections sections as mentioned in the Airflow UI  ***method:*** ‘POST’ or ‘GET’ (or other HTTP verbs)  ***headers:*** The headers for the request  ***data:*** The payload for the request in a valid JSON string  ***response\_check:*** This parameter checks the value of a particular parameter in the response to make a decision if the request was completed successfully or failed  ***xcom\_push:*** pushes to values to the Xcom  ***log\_response:*** The response of the request gets logged in the logs section of the UI  ***dag:*** The dag where the operator belongs |

**Solution Overview**



1. The user using the EDL UI creates a configuration file which is stored on a local storage.
2. The local storage is used to store the EDL configuration files which are based on an on-prem location where there is a designated directory available to fetch files from.
3. The Unix Polling Script is used as a file watcher that watches the above-designated location continuously for new files and triggers the dags on the airflow scheduler for each configuration file present on the local storage.
4. Airflow dags running on the airflow webserver are triggered using the Airflow REST API.
5. Airflow dags that are triggered send a POST request to PDILL API that processes the data using underlying jobs.

Questions:

1. How many servers will airflow need to talk to in order to trigger workflows?
2. What would the configuration file contain?
3. Is the EDL configuration file a one-time use configuration?
4. Will the configurations be unique?
5. Is there a naming convention defined for the configuration file?
6. On what basis can I decide which dag is supposed to be triggered?