**QA**

**1 what is bash operator**

**This operator allow you run any bash command directly in your workflow.**

**Imagine needing to copy a file - you can use a Bash Operator with the cp command.**

**Executes a bash command**

In BashOperator, a component of Apache Airflow, is a type of operator that executes a bash command or a series of bash commands. BashOperator allows you to run scripts or commands that you would typically run in a Unix-like shell environment directly from your Airflow DAG (Directed Acyclic Graph).

**how it is work on airflow**

**Defining the Operator**: You define a BashOperator in your Airflow DAG by importing it and then instantiating it with the desired parameters. Typically, you specify the bash command(s) you want to execute within the bash\_command parameter.

**Execution**: When your DAG is triggered and reaches the task associated with the BashOperator, Airflow will execute the specified bash command(s) as a subprocess. This means that Airflow will create a new process and execute the command(s) in a shell environment.

**Logging and Monitoring**: Airflow captures the stdout and stderr streams from the bash command(s) executed by the BashOperator. This allows you to monitor the execution of the command(s) and check for any errors or warnings.

**Error Handling**: Airflow provides mechanisms for error handling with BashOperator. You can specify retries, timeouts, and provide custom error handling logic to handle failures gracefully.

**Dependencies**: Like other operators in Airflow, you can define dependencies between tasks. This allows you to control the order in which tasks are executed within your DAG.

**Example:**

from airflow import DAG

from airflow.operators.bash\_operator import BashOperator

from datetime import datetime

default\_args = {

'owner': 'airflow',

'depends\_on\_past': False,

'start\_date': datetime(2024, 5, 20),

'retries': 1

}

dag = DAG('bash\_example', default\_args=default\_args, schedule\_interval='@daily')

task1 = BashOperator(

task\_id='print\_date',

bash\_command='date',

dag=dag

)

task2 = BashOperator(

task\_id='sleep',

bash\_command='sleep 5',

retries=3,

dag=dag

)

task3 = BashOperator(

task\_id='print\_hello',

bash\_command='echo "hello"',

dag=dag

)

task1 >> task2 >> task3

In this example, we define a DAG with three tasks. The first task (print\_date) simply prints the current date using the date command. The second task (sleep) sleeps for 5 seconds. The third task (print\_hello) echoes "hello". The tasks are chained together so that print\_date executes first, followed by sleep, and then print\_hello.

**2 what is python operator and how it is works on airflow**

**This one lets you call Python functions.**

**Maybe you have a complex data processing step written in Python - this operator can execute that function.**

PythonOperator in Airflow enables execution of custom Python functions within DAGs for seamless workflow automation.

A PythonOperator in Apache Airflow enables you to execute Python functions or callables within your DAGs. It allows for dynamic execution of Python code as part of your workflow

Let's say you have a data pipeline where you need to process some data using Python functions. You want to integrate this process into your Airflow workflow.

**Example:**

import requests

from airflow import DAG

from airflow.operators.python\_operator import PythonOperator

from datetime import datetime

def fetch\_data():

response = requests.get('https://api.example.com/data')

data = response.json()

# Store data in database (this is just a placeholder)

# database.store(data)

print("Data fetched and stored successfully!")

default\_args = {

'owner': 'airflow',

'depends\_on\_past': False,

'start\_date': datetime(2024, 5, 20),

'retries': 1

}

dag = DAG('data\_fetch\_dag', default\_args=default\_args, schedule\_interval='@daily')

task\_fetch\_data = PythonOperator(

task\_id='fetch\_data\_task',

python\_callable=fetch\_data,

dag=dag

)

Explation of above code:

The fetch\_data function fetches data from an API and prints a success message.

The PythonOperator fetch\_data\_task executes the fetch\_data function within the Airflow DAG.

When the DAG is triggered, Airflow will execute the fetch\_data function, fetching data from the API.

Any output produced by the fetch\_data function will be captured in the Airflow logs for monitoring.

**3 what is sftp operator**

**SFTPOperator allows you to transfer files to or from a remote server using SFTP (SSH File Transfer Protocol). This operator is useful for tasks involving file transfer operations between Airflow and remote systems.**

The SFTPOperator in Apache Airflow allows you to perform file transfer operations using the SFTP (SSH File Transfer Protocol) protocol. It's particularly useful for scenarios where you need to securely transfer files between systems.

Imagine you have a data pipeline where you need to regularly transfer files between servers. You want to automate this process within your Airflow workflow. Here's how you would use the SFTPOperator:

**Defining the Operator:**

First, you define an SFTPOperator in your Airflow DAG by importing it and then instantiating it with the required parameters. You specify details such as the SSH connection, the remote and local paths, and the operation (e.g., put, get).

**Using SFTPOperator:**

You specify the SFTP operation you want to perform (e.g., uploading a file to a remote server or downloading a file from a remote server) and provide the necessary details such as the file paths and SSH connection.

**Execution:**

When your DAG is triggered and reaches the task associated with the SFTPOperator, Airflow will execute the specified SFTP operation using the provided parameters.

**Logging and Monitoring:**

Airflow captures logs of the SFTP operation, allowing you to monitor the transfer process and check for any errors or warnings.

**Error Handling:**

You can implement error handling within your DAG to handle exceptions gracefully. Airflow also provides mechanisms for retries, timeouts, and custom error handling to ensure robustness.

**Example:**

from airflow import DAG

from airflow.contrib.hooks.sftp\_hook import SFTPHook

from airflow.operators.sftp\_operator import SFTPOperator

from datetime import datetime

default\_args = {

'owner': 'airflow',

'start\_date': datetime(2024, 5, 20),

}

dag = DAG('sftp\_transfer\_dag', default\_args=default\_args, schedule\_interval='@daily')

local\_file\_path = '/path/to/local/file.csv'

remote\_file\_path = '/path/to/remote/file.csv'

sftp\_hook = SFTPHook(ssh\_conn\_id='my\_ssh\_connection')

task\_transfer\_file = SFTPOperator(

task\_id='transfer\_file\_task',

ssh\_hook=sftp\_hook,

local\_filepath=local\_file\_path,

remote\_filepath=remote\_file\_path,

operation='put',

dag=dag

)

We define a DAG named sftp\_transfer\_dag scheduled to run daily.

We specify the local file path (local\_file\_path) and the remote file path (remote\_file\_path) for the file transfer operation.

We instantiate an SFTPHook to establish an SSH connection to the remote server.

The transfer\_file\_task SFTPOperator uploads the CSV file from the local server to the remote server using the specified paths and SSH connection.

**4 gcs operator**

**Airflow provides several operators for interacting with Google Cloud Storage (GCS), allowing you to perform various operations**

**such as uploading files, downloading files, deleting files, and more.**

The GCS (Google Cloud Storage) Operator in Apache Airflow facilitates interactions with Google Cloud Storage, allowing you to perform various operations such as uploading, downloading, copying, and deleting files stored in Google Cloud Storage buckets.

Suppose you have data stored in Google Cloud Storage, and you need to manage it as part of your data pipeline. The GCS Operator enables you to seamlessly integrate these operations into your Airflow workflows.

**Defining the Operator:**

You define a GCS Operator in your Airflow DAG by importing it and then instantiating it with the required parameters. These parameters typically include the GCS connection ID, the bucket name, and details of the operation you want to perform (e.g., upload, download).

**Using GCS Operator:**

You specify the GCS operation you want to perform (e.g., uploading a file to a bucket, downloading a file from a bucket) and provide the necessary details such as file paths, bucket names, and authentication credentials.

**Execution:**

When your DAG is triggered and reaches the task associated with the GCS Operator, Airflow will execute the specified GCS operation using the provided parameters.

**Logging and Monitoring:**

Airflow captures logs of the GCS operation, allowing you to monitor the process and check for any errors or warnings.

**Error Handling:**

You can implement error handling within your DAG to handle exceptions gracefully. Airflow also provides mechanisms for retries, timeouts, and custom error handling to ensure robustness.

**Example 1: Uploading a File to Google Cloud Storage:**

Suppose you have a local CSV file that you want to upload to Google Cloud Storage.

from airflow import DAG

from airflow.contrib.operators.gcs\_operator import GoogleCloudStorageUploadOperator

from datetime import datetime

default\_args = {

'owner': 'airflow',

'start\_date': datetime(2024, 5, 20),

}

dag = DAG('gcs\_upload\_dag', default\_args=default\_args, schedule\_interval='@daily')

local\_file\_path = '/path/to/local/file.csv'

gcs\_bucket = 'your\_bucket\_name'

gcs\_object = 'file.csv'

task\_upload\_file = GoogleCloudStorageUploadOperator(

task\_id='upload\_file\_task',

bucket=gcs\_bucket,

object=gcs\_object,

filename=local\_file\_path,

dag=dag

)

In this example, the upload\_file\_task uploads the local CSV file to the specified Google Cloud Storage bucket.

**Example 2: Downloading a File from Google Cloud Storage:**

Suppose you want to download a CSV file from Google Cloud Storage to a local directory.

from airflow import DAG

from airflow.contrib.operators.gcs\_operator import GoogleCloudStorageDownloadOperator

from datetime import datetime

default\_args = {

'owner': 'airflow',

'start\_date': datetime(2024, 5, 20),

}

dag = DAG('gcs\_download\_dag', default\_args=default\_args, schedule\_interval='@daily')

gcs\_bucket = 'your\_bucket\_name'

gcs\_object = 'file.csv'

local\_file\_path = '/path/to/local/directory/'

task\_download\_file = GoogleCloudStorageDownloadOperator(

task\_id='download\_file\_task',

bucket=gcs\_bucket,

object=gcs\_object,

filename=local\_file\_path + gcs\_object,

dag=dag

)

In this example, the download\_file\_task downloads the CSV file from the specified Google Cloud Storage bucket to the local directory.

**5 How IAM roles works:**

As a GCP data engineer, IAM (Identity and Access Management) roles are crucial for managing access to various data-related services and resources within Google Cloud Platform. Here's how IAM roles work specifically for a data engineer:

**Granular Access Control**: IAM roles allow you to define granular access controls for different data-related services and resources within GCP. This includes services like BigQuery, Cloud Storage, Pub/Sub, Dataflow, and others that are commonly used by data engineers.

**Least Privilege Principle**: Following the principle of least privilege, you assign IAM roles to users, service accounts, or groups with only the permissions they need to perform their specific data engineering tasks. For example, you might grant read-only access to a dataset in BigQuery for a data analyst or grant write access to a Cloud Storage bucket for a data pipeline.

**Predefined Roles**: GCP provides a set of predefined IAM roles tailored for data-related tasks. These roles include roles like BigQuery Admin, BigQuery Data Viewer, Storage Object Admin, Storage Object Creator, and others. You can assign these roles to users based on their responsibilities and the level of access required.

**Custom Roles**: In addition to predefined roles, GCP allows you to create custom IAM roles with specific sets of permissions tailored to your organization's requirements. As a data engineer, you might create custom roles to enforce stricter access controls or to grant permissions for specific data processing tasks.

**Service Accounts**: Service accounts play a crucial role in data engineering workflows. They are used to authenticate applications and services running on GCP, such as data pipelines built with Dataflow or batch jobs running on Compute Engine. As a data engineer, you manage the IAM roles assigned to service accounts to ensure they have the necessary permissions to access data sources, write output, and interact with other GCP services.

**Auditing and Compliance**: IAM roles also play a significant role in auditing and compliance efforts. By maintaining proper IAM role assignments and regularly reviewing access controls, you ensure that data access is compliant with organizational policies and regulatory requirements.

**6 how the roles are managed in airflow UI**

As an Airflow developer, managing roles within the Airflow UI involves configuring role-based access control (RBAC) to control user permissions and access levels to various Airflow features and resources. Here's how roles are managed in the Airflow UI:

**RBAC Configuration**: Airflow provides RBAC functionality starting from version 1.10. The RBAC feature allows administrators to define roles and permissions for users accessing the Airflow UI.

**Predefined Roles**: Airflow comes with a set of predefined roles that cover common use cases. These roles typically include:

**Admin**: Users with administrative privileges who can perform all actions within Airflow, including managing connections, variables, DAGs, and users.

**User**: Regular users who can view DAGs, task instances, and logs but cannot modify Airflow configurations or perform administrative tasks.

**Custom Roles**: In addition to predefined roles, Airflow allows administrators to create custom roles with specific sets of permissions tailored to their organization's requirements. For example, you might create custom roles for data engineers, data analysts, or QA testers with different levels of access to Airflow features.

**Permission Management**: Within the Airflow UI, administrators can assign permissions to roles, specifying which actions each role is allowed to perform. Permissions may include viewing DAGs, triggering DAG runs, pausing or unpausing DAGs, accessing task logs, and more.

**User Management**: Airflow administrators can manage users and their associated roles directly within the Airflow UI. They can create new user accounts, assign roles to users, and deactivate or delete user accounts as needed.

**Authentication Providers**: Airflow supports various authentication providers, such as LDAP, OAuth, and custom authentication backends. Depending on the chosen authentication method, user accounts may be managed externally, and role assignments may be synchronized with external identity providers.

**Access Control Lists (ACLs):** In addition to role-based access control, Airflow also supports ACLs for finer-grained access control over individual DAGs and tasks. ACLs allow administrators to specify which users or roles have access to specific DAGs and tasks.

**Audit Logging:** Airflow logs user actions and access attempts, providing administrators with visibility into who accessed the Airflow UI and what actions they performed. Audit logs can help identify security incidents, track user activity, and ensure compliance with organizational policies.

**7 why we use dag and how the task executes:**

As an Airflow developer, you use Directed Acyclic Graphs (DAGs) to define and orchestrate your data workflows. DAGs are fundamental to Airflow's architecture and workflow management.

**Why Use DAGs?**

**Workflow Definition**: DAGs provide a structured way to define the sequence of tasks and dependencies in your data pipeline. You can visually represent the flow of data and operations from start to finish, making it easier to understand and manage complex workflows.

**Dependency Management**: DAGs allow you to define dependencies between tasks, ensuring that tasks are executed in the correct order based on their dependencies. This enables parallel execution of independent tasks and ensures that downstream tasks wait for upstream tasks to complete before executing.

**Scheduling and Monitoring**: Airflow uses DAGs to schedule and monitor the execution of tasks. You can specify the schedule for executing tasks (e.g., daily, hourly) and monitor the progress and status of task executions through the Airflow UI or command-line interface.

**Retry and Error Handling**: DAGs provide built-in mechanisms for retrying failed tasks and handling errors. You can configure the number of retries, set retry intervals, and define error handling logic to handle exceptions and failures gracefully.

**Scalability and Parallelism**: DAGs allow you to scale your data workflows horizontally by adding more workers or vertically by parallelizing tasks within a DAG. This enables efficient utilization of resources and faster execution of tasks.

**How Tasks Execute Within DAGs**

**Task Definition**: Within a DAG, you define individual tasks, each representing a unit of work to be performed. Tasks can be Python functions (PythonOperator), Bash commands (BashOperator), SQL queries (BigQueryOperator), or any other operation supported by Airflow's operators.

**Dependency Declaration**: Tasks within a DAG can have dependencies on other tasks. You specify dependencies using the set\_upstream() and set\_downstream() methods or the >> and << operators. Dependencies define the execution order of tasks within the DAG.

**Execution Schedule**: Airflow's Scheduler periodically checks the DAGs for tasks that are ready to be executed based on their schedule (e.g., cron expression). When a task's dependencies are met, the Scheduler triggers the task for execution.

**Task Execution**: When a task is triggered for execution, Airflow's Executor assigns it to an available worker node for execution. The worker executes the task, which may involve running Python code, executing Bash commands, querying databases, or interacting with external systems.

**Task State Management**: As tasks execute, Airflow tracks their state (e.g., running, success, failure) and updates the task status in the metadata database. Task logs and outputs are also captured and stored for monitoring and troubleshooting purposes.

**Task Completion and Cleanup**: After a task completes execution, Airflow updates its status and triggers downstream tasks for execution if their dependencies are met. Airflow also performs any necessary cleanup operations, such as removing temporary files or releasing resources used by the task.

**8 how to pass the run time parameters while running the dag**

As an Airflow developer, you can pass runtime parameters to your DAG runs using Airflow's default\_args and DAGRun.conf feature. Here's how you can do it:

**Define Default Arguments**: In your DAG definition, you can define default\_args, which is a dictionary containing default parameters for your DAG. These parameters will be used for every DAG run unless overridden during runtime.

**Accessing Runtime Parameters**: Within your DAG definition or within your task functions, you can access runtime parameters passed to the DAG run using context['dag\_run'].conf. This allows you to dynamically adjust the behavior of your DAG or tasks based on runtime parameters.

Here's a step-by-step guide on how to pass and access runtime parameters in Airflow:

**Step 1: Define Default Arguments**

from airflow import DAG

from datetime import datetime

default\_args = {

'owner': 'airflow',

'start\_date': datetime(2024, 5, 20),

'retries': 1,

'retry\_delay': timedelta(minutes=5)

}

dag = DAG('my\_dag', default\_args=default\_args, schedule\_interval='@daily')

**Step 2: Access Runtime Parameters**

Within your DAG definition or within your task functions, you can access the runtime parameters using context['dag\_run'].conf`.

Here's an example of how you can access and use the runtime parameters:

from airflow import DAG

from airflow.operators.python\_operator import PythonOperator

from datetime import datetime

def my\_task(\*\*kwargs):

runtime\_parameters = kwargs['dag\_run'].conf

# Access and use runtime parameters here

print("Runtime parameters:", runtime\_parameters)

task1 = PythonOperator(

task\_id='my\_task',

python\_callable=my\_task,

provide\_context=True,

dag=dag

)

**Step 3: Triggering the DAG Run with Runtime Parameters**

When triggering the DAG run, you can pass runtime parameters using the conf parameter. For example:

airflow dags trigger --conf '{"param1": "value1", "param2": "value2"}' my\_dag

In this example, param1 and param2 are the runtime parameters that will be accessible within your DAG or task functions.

**CREATE A DAG:**

**5 steps involved:**

**Step 1:** Import all the necessary modules required in the flow

Example:

from airflow import DAG

from airflow.models import Variable

from operators.updated\_infa\_workflow\_operator import InformaticaWorkflowOperator

from utils.email\_alerts import email\_on\_success, email\_on\_failure

from airflow.operators.dummy import DummyOperator

from datetime import datetime

import yaml

import os

**Step 2:** Define all the default arguments. Airflow allows passing a dictionary of parameters that would be available to all the task in that DAG. This is also playing a major role when you want alerts on individual task failures instead of just DAG failures

Some of the default arguments involves:

Owner: Owner of the dag

depends\_on\_past: task dependency with the previous run

email\_on\_failure: Email alert for any task failure

email: email address

retries: No. of retries automatically whenever task is failed

We can even pass the connection variables inside the dag if the connection is used in complete dag

Example: ‘gcp\_conn\_id’ = ‘gcp\_query\_connection’

default\_args = {

"owner": "airflow",

"depends\_on\_past": False,

"provide\_context": True,

"email\_on\_retry" : False,

"email\_on\_failure" : False

}

**Step 3:** Instantiate a Dag

Defining all the dag level information like dag\_id , schedule interval, parameters …

There parameters are nothing but a replica of metadata table “dag” in the airflow backend database.

Example:

dag = DAG(os.path.basename(\_\_file\_\_).replace(".py",""), #parameterized dag name

start\_date=datetime(2021, 10, 4), ### start date of the dag for first run

schedule\_interval='\*/10 \* \* \* \*', ##defining the schedule interval as cron expression

default\_args = default\_args, ## Reading all the default arguments

catchup=False, ## to catch all the missing instances..

)

**Step 4:** Task definition

Defining the task whether it’s a simple Bach command or specific operator used to trigger the respective flow.

Example:

task1 = BashOperator(task\_id='test\_echo\_3',

bash\_command="echo 'airflow test'",

pool\_slots=1,

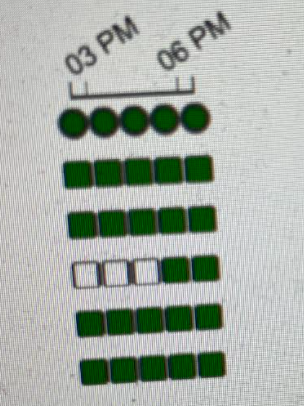
dag=dag)

**Step 5:** Setting up the dependencies between the tasks

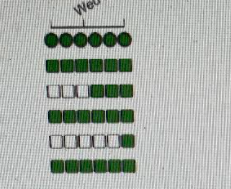
Example: task1 >> task2 >> task3 >> task4 >> task5

**Editing Task & Dag Metadata:**

**Add Task:** Tasks can be added but the new task would have no state in the UI for already executed dag.



**Rename Task:** Task name should not be edited once dag is deployed as the new task name would not be shown in the UI. Renaming the task loses all the pervious metadata.



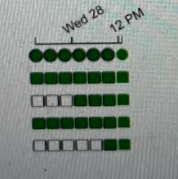
**Change Schedule Interval:** We have to create a new dag whenever we have to change the schedule interval as it may lead in corruption if we try to update the same dag. Airflow Run Id is generated based on the schedule interval of the dag.

**Rename the dag:** This creates a new entry in UI and metadata table in DB and the old dag metadata will remain same.

**Change start time:** This also plays a major role same as schedule interval.

**Change Dependency:** Yes, we can change the dependency any time but we have to be sure that dag is not having any active running instance

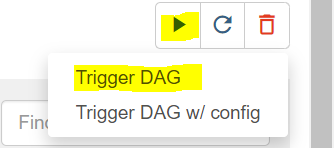
**Remove Task:** Doesn’t impact. Task entries will be removed.



**Execution of DAG:**

We have two options to execute the dag.

**Option 1:** Executing the dag from UI using trigger dag option



**Trigger dag**: Directly triggering dag (for all the Adhoc runs and for testing the dag)

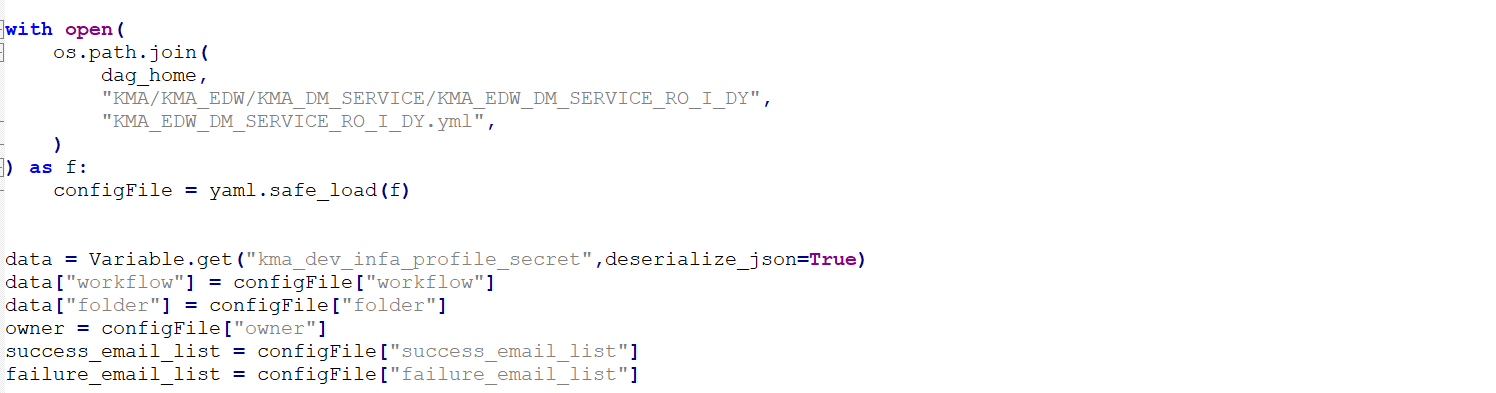
**Trigger dag with config:** Triggering the dag based on config file.

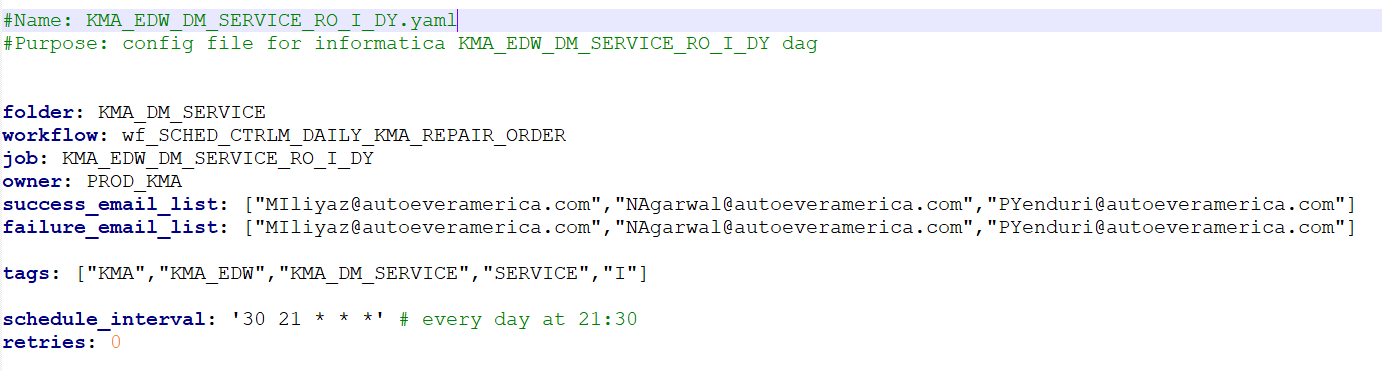
**Example: KMA\_EDW\_DM\_SERVICE\_RO\_I\_DY**

Dag consists of below parameters to read the config file

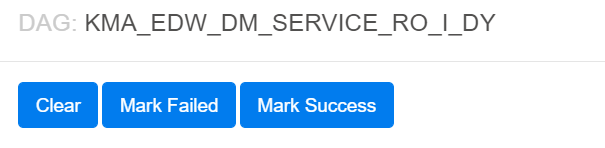
Dag Name: KMA\_EDW\_DM\_SERVICE\_RO\_I\_DY.py

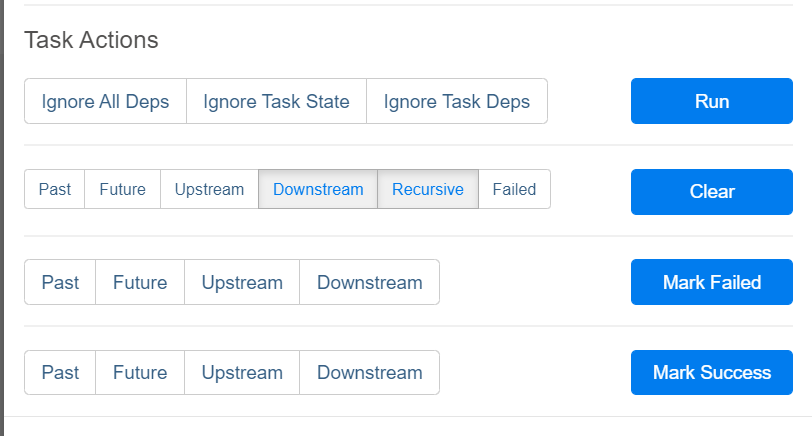
Config (Yaml) file name: KMA\_EDW\_DM\_SERVICE\_RO\_I\_DY.yml





**Different Actions involved in running the dag/tasks:**





**Case 1:** Rerun the failure job from its same position

Click on the failure square box → click on "clear" button → its clears the complete jobs along with the downstream and starts running from the failure position.

Note: Options should be check in before click on "Clear" button are Downstream & Recursive

**Case 2**: Success the failure job and run from next set of jobs .

Click on Failure square box and click on "mark success" . Next job which was "upstream failed status" has to be cleared using "clear button"

Example: task1 → task2 → task3 → task4 . Case when task2 is failed and was mark successes. Click on Task3 and use the "clear" button (Options should be check in before click on "Clear" button are Downstream & Recursive )

Note: If you want to continue the next set of jobs in that case none of the options should be check in.

**Case 3:** Run the job which was in already completed and in success state.

Click on the respective task box and check in "ignore All Deps" "Ignore Task state" "ignore task deps" and then → click on "Run" button to run the specific task.

**Case 4**: if the entire instance has to be triggered again , there are two ways .

a) Using "Clear" button from the first task which makes to run the complete instance

or

b) Using trigger dag option to trigger the complete instance separately .

**Case 5**: Kill/Fail the current running instance . There were multiple ways to achieve

a) Click on the running instance square box : use "Mark Failed" button by check in downstream

b) Click on "Browse" button on top of the header page → Dag Runs → Click on "Search" → Add Filter" → Add required filter as per Dag id or task d ... → Only check (click) the respective task → click on "Action" → Set state to failed → Ok..

here in the option b we have different status we can achieve like success, fail or make the task as running..

**Case 6:** Delete the Dag permanently from Airflow..

a) Delete the dag from the Web UI by click on Delete button on the respective dag right most corner.

b) Delete the same "XXXX.py" file from the Linux/Unix server.

**Case 7:** If the webpage displays message : as Last heart beat received few minutes ago: Indicates that worker node is down and none of the new dags/tasks will not be able to get triggered until the worker is restarted.

**Case 8**: to Fail the complete instances which are about to get trigger with no status: Use "Mark Failed" (check in the downstream). Which marks the jobs in failed status along with downstream jobs.

**Case 9**: "Mark success" can be used for one to multiple jobs (if you want to success all the downstream jobs as well please check in downstream)

**Option 2:** Using CLI based commands for each case defined above ..

Trigger dag: airflow trigger\_dag #directory #RUN\_ID #CONF #EXEC\_DATE dag\_id

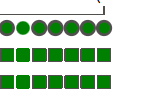
Pause the dag: airflow pause #directory dag\_id

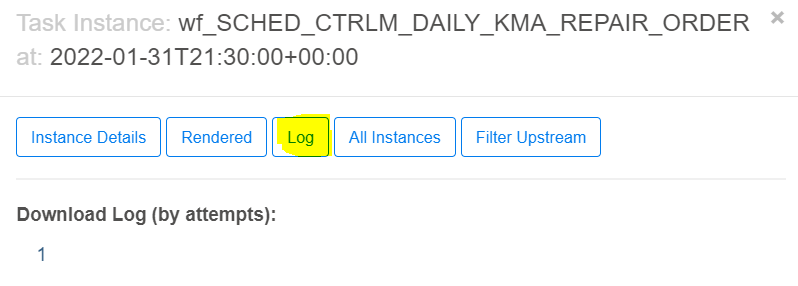
Unpause the dag: airflow unpause #directory dag\_id

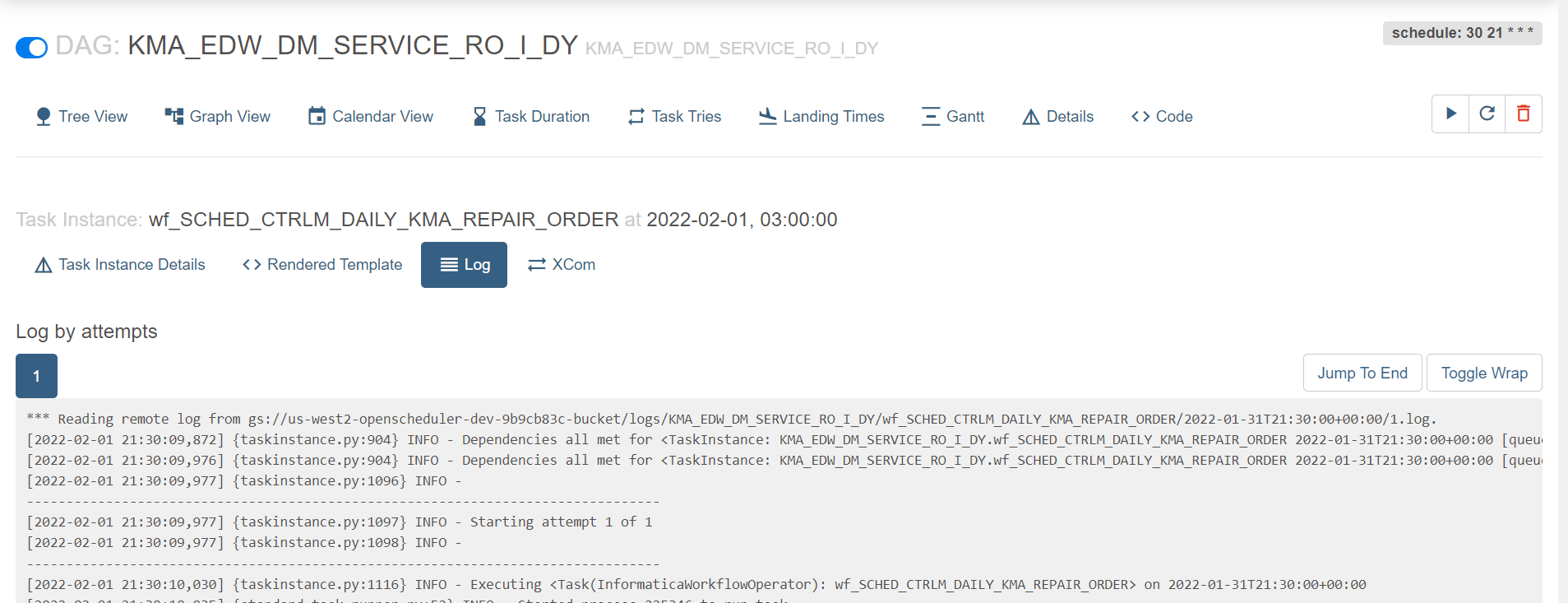
**VALIDATION OF DAG/TASK RUNS:**

We have two ways of validating the logs of the dag/task runs…

**Validating from UI:** Click on the required task instance and it propagates with the window to open the logs







**Validating it from the bucket:**

Sample log name: gs://us-west2-openscheduler-dev-9b9cb83c-bucket/logs/KMA\_EDW\_DM\_SERVICE\_RO\_I\_DY/wf\_SCHED\_CTRLM\_DAILY\_KMA\_REPAIR\_ORDER/2022-01-31T21:30:00+00:00/1.log

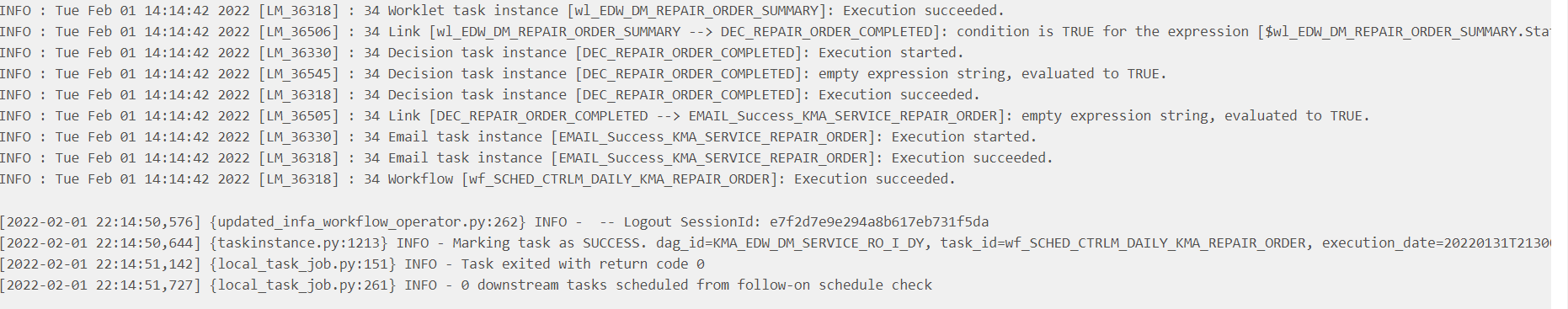
**Log structure:**

Log structure follows: dagname / taskid / run id of dag / 1.log

If we have multiple runs with the same run id due to failure log name continues as follows with 2.log, 3.log etc..

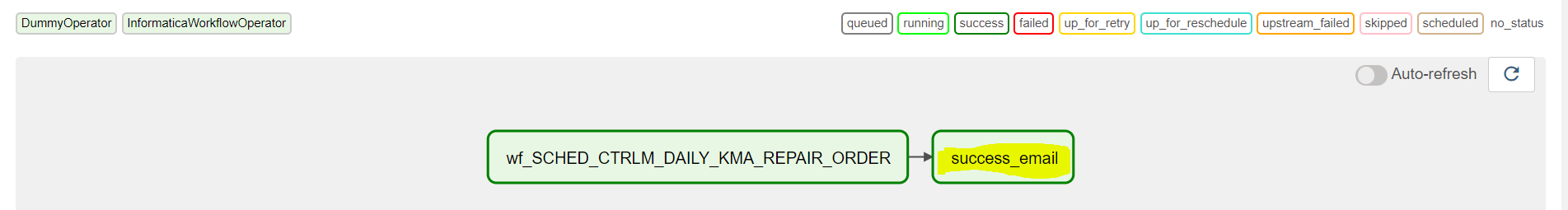
**Validation of Work flow status:**

UI: We can validate the workflow status at the end of each log execution



All the dags are defined by executing a workflow and then success email based on success of the work flow execution.

Receival of Success email states that the workflow is completed successfully.



All the failure workflows has its email alerts generates with the errors.

corn\_Expressions:

DY 00 12 \* \* \* Daily runs at 12 am

WY 00 12 \* \* 0 every sunday runs at 12 am 0sun 1mon 2tue 3wed 4thu 5fri 6sat

MY 00 12 \* 3 \* every 3rd month runs at 12 am

00 12 \* \* 2#2 every thrid week tuesday runs at 12 am

in corn expresiion we cant find corn expression we can simple use CUSTOM\_CALENDER