DAY 3

Dynamic Task Generation

from airflow import DAG

#from airflow.operators.dummy\_operator import DummyOperator

#from airflow.operators.python\_operator import PythonOperator

from datetime import datetime

from airflow.models.variable import Variable

from airflow.operators.empty import EmptyOperator

import requests

# Function to be executed by the PythonOperator

def print\_task(task\_name):

print(f"Executing task: {task\_name}")

# List of task names

#task\_names = ['task\_1', 'task\_2', 'task\_3', 'task\_4', 'task\_5']

#task\_names = Variable.get("task\_list", deserialize\_json=True)

# Define the DAG

dag = DAG(

'dynamic\_task\_example',

description='A simple dynamic DAG',

schedule\_interval=None, # Define the schedule, 'None' for manual trigger

start\_date=datetime(2024, 11, 21),

catchup=False,

)

# Create a start task

start\_task = EmptyOperator(

task\_id='start',

dag=dag,

)

# Dynamically generate tasks based on task\_names

for task\_name in task\_names:

task = PythonOperator(

task\_id=task\_name,

python\_callable=print\_task,

op\_args=[task\_name],

dag=dag,

)

# Set task dependencies: start -> task -> end

start\_task >> task

# Create an end task

end\_task = DummyOperator(

task\_id='end',

dag=dag,

)

# Set dependencies: task -> end

for task\_name in task\_names:

task = dag.get\_task(task\_name)

task >> end\_task

API Interaction

def fetch\_people\_from\_api():

api\_url = "http://api.open-notify.org/astros.json"

response = requests.get(api\_url)

if response.status\_code == 200:

return response.json() # Returns a list of posts

else:

raise ValueError("Failed to fetch data from the API")

people = fetch\_people\_from\_api()

task\_names = []

for i in range(people['number']):

p = people['people'][i]['name'].split(" ")[0]

#title = post['title']

if p not in task\_names:

task\_names.append(p)

Dynamic DAG generation

from \_\_future\_\_ import annotations

import pendulum

from airflow.models.dag import DAG

from airflow.models.variable import Variable

from airflow.operators.empty import EmptyOperator

# --- 1. Get the list of DAG names from the Airflow Variable ---

# We use deserialize\_json=True to automatically parse the JSON list.

try:

dag\_names = Variable.get("dag\_names\_list", deserialize\_json=True)

except KeyError:

# Set a default list if the variable is not found

dag\_names = ["default\_dynamic\_dag"]

# --- 2. Define a function to create a DAG ---

# This is a "DAG factory" function.

def create\_dag(dag\_id: str):

"""

Factory function to create a DAG object with a simple task.

Args:

dag\_id: The unique identifier for the DAG.

Returns:

A new DAG object.

"""

with DAG(

dag\_id=dag\_id,

start\_date=pendulum.datetime(2025, 1, 1, tz="UTC"),

catchup=False,

schedule=None,

tags=["dynamic-generation", "example"],

) as dag:

# Define a simple task for this DAG

EmptyOperator(task\_id="start\_task")

return dag

# --- 3. Loop through the names and generate DAGs ---

for name in dag\_names:

dag\_id = f"{name}"

globals()[dag\_id] = create\_dag(dag\_id)

Xcom Generation

from \_\_future\_\_ import annotations

import pendulum

from airflow.decorators import task

from airflow.models.dag import DAG

from airflow.operators.bash import BashOperator

from airflow.operators.python import PythonOperator

def \_producer\_explicit\_push(\*\*kwargs):

"""

Pushes multiple values to XCom with custom keys.

This provides more control than just returning a single value.

"""

ti = kwargs["ti"]

user\_data = {"name": "Alice", "id": "user\_1234", "role": "admin"}

transaction\_ids = [9001, 9002, 9005, 9010]

# Push a dictionary with a specific key

ti.xcom\_push(key="user\_profile", value=user\_data)

# Push a list with another key

ti.xcom\_push(key="recent\_transactions", value=transaction\_ids)

print(f"Pushed user profile: {user\_data}")

print(f"Pushed transaction IDs: {transaction\_ids}")

def \_consumer\_pull\_multiple(\*\*kwargs):

"""

Pulls the multiple, specific values pushed by the producer task.

"""

ti = kwargs["ti"]

# Pull the dictionary by its custom key

user = ti.xcom\_pull(task\_ids="producer\_explicit\_push", key="user\_profile")

# Pull the list by its custom key

transactions = ti.xcom\_pull(task\_ids="producer\_explicit\_push", key="recent\_transactions")

if not user or not transactions:

raise ValueError("Failed to pull required values from XCom!")

print(f"Pulled user name: {user['name']}")

print(f"Number of transactions pulled: {len(transactions)}")

print(f"Processing transactions for user ID {user['id']}...")

with DAG(

dag\_id="xcom\_dag",

start\_date=pendulum.datetime(2023, 1, 1, tz="UTC"),

catchup=False,

schedule=None,

tags=["example", "xcom", "advanced"],

doc\_md=\_\_doc\_\_,

) as dag:

# Traditional PythonOperator for explicit push/pull

producer\_task = PythonOperator(

task\_id="producer\_explicit\_push",

python\_callable=\_producer\_explicit\_push,

)

consumer\_task = PythonOperator(

task\_id="consumer\_pull\_multiple",

python\_callable=\_consumer\_pull\_multiple,

)

# BashOperator using Jinja templating to pull an XCom value

# Note the special Jinja syntax to pull the 'user\_profile' dictionary

# and access the 'name' key within it.

bash\_consumer = BashOperator(

task\_id="bash\_consumer\_jinja",

bash\_command='echo "User name pulled from XCom via Jinja: {{ ti.xcom\_pull(task\_ids=\'producer\_explicit\_push\', key=\'user\_profile\')[\'name\'] }}"',

)

# Set dependencies for the traditional tasks

producer\_task >> [consumer\_task, bash\_consumer]

Nested Task Group:

from \_\_future\_\_ import annotations

import pendulum

from airflow.models.dag import DAG

from airflow.operators.bash import BashOperator

from airflow.utils.task\_group import TaskGroup

with DAG(

dag\_id="task\_group\_example\_dag",

start\_date=pendulum.datetime(2023, 1, 1, tz="UTC"),

catchup=False,

schedule=None,

tags=["example", "taskgroup"],

doc\_md=\_\_doc\_\_,

) as dag:

# A task defined at the top level of the DAG

start = BashOperator(task\_id="start", bash\_command="echo 'Starting the pipeline...'")

# Define the main TaskGroup

# The group\_id is what's displayed in the Graph View

with TaskGroup(group\_id="data\_processing\_group", tooltip="All data processing steps") as data\_processing\_group:

extract = BashOperator(

task\_id="extract\_data",

bash\_command="echo 'Extracting data...'"

)

validate = BashOperator(

task\_id="validate\_data",

bash\_command="echo 'Validating data...'"

)

# Define a nested TaskGroup for transformation tasks

with TaskGroup(group\_id="transformation\_group", tooltip="Data transformation tasks") as transformation\_group:

# These tasks will appear inside the 'transformation\_group' which is inside 'data\_processing\_group'

transform\_a = BashOperator(

task\_id="transform\_a",

bash\_command="echo 'Running transformation A'"

)

transform\_b = BashOperator(

task\_id="transform\_b",

bash\_command="echo 'Running transformation B'"

)

transform\_c = BashOperator(

task\_id="transform\_c",

bash\_command="echo 'Running transformation C'"

)

# Set dependencies within the nested group

[transform\_a, transform\_b] >> transform\_c

load = BashOperator(

task\_id="load\_data",

bash\_command="echo 'Loading data...'"

)

# Set dependencies within the main TaskGroup.

# Note how we refer to the entire nested TaskGroup `transformation\_group` as a single unit.

extract >> validate >> transformation\_group >> load

# A final task at the top level

end = BashOperator(task\_id="end", bash\_command="echo 'Pipeline finished.'")

# Set the final top-level dependencies

# The entire `data\_processing\_group` is treated as one entity in this dependency chain.

start >> data\_processing\_group >> end

DAY 4:

Visit https://github.com/ to create a github account using personal email id

● Download and install git from https://git-scm.com/downloads/win

○ Verify running command on bash : git

echo "# AstroCICD" >> README.md

git init

git add README.md

git commit -m "first commit"

git branch -M main

git remote add origin https://github.com/<git-username>/<git-repo>.git

git push -u origin main

Custom operators code  
  
# plugins/operators/greeting\_operator.py

from airflow.models.baseoperator import BaseOperator

from airflow.utils.decorators import apply\_defaults

class GreetingOperator(BaseOperator):

"""

A simple custom operator that logs a greeting message.

:param name: The name to greet.

:type name: str

"""

@apply\_defaults

def \_\_init\_\_(self, name: str, \*args, \*\*kwargs) -> None:

super().\_\_init\_\_(\*args, \*\*kwargs)

self.name = name

def execute(self, context):

"""

The main logic of the operator. It logs the greeting message.

"""

self.log.info(f"Hello, {self.name}! 👋")

self.log.info("This greeting comes from a custom operator.")

return f"Successfully greeted {self.name}."

# dags/use\_custom\_operator\_dag.py

from airflow.decorators import dag

from datetime import datetime

# Import the custom operator from the plugins directory

from operators.greeting\_operator import GreetingOperator

@dag(

dag\_id='custom\_operator\_dag\_example',

start\_date=datetime(2025, 10, 16),

schedule=None,

catchup=False,

tags=['example', 'custom\_operator'],

)

def use\_custom\_operator\_dag():

"""

A simple DAG to demonstrate the use of a custom operator.

"""

# Instantiate the custom operator as a task

greet\_airflow\_task = GreetingOperator(

task\_id='greet\_airflow',

name='Airflow' # Pass the custom parameter here

)

# You can use it multiple times with different parameters

greet\_world\_task = GreetingOperator(

task\_id='greet\_the\_world',

name='World'

)

greet\_airflow\_task >> greet\_world\_task

# Instantiate the DAG

use\_custom\_operator\_dag()

Day 5 AWS connection

import pandas as pd

from airflow import DAG

from airflow.providers.amazon.aws.hooks.s3 import S3Hook

from airflow.providers.amazon.aws.sensors.s3 import S3KeySensor

from airflow.operators.python import PythonOperator

import pendulum

import io

import logging

# Default arguments for the DAG

default\_args = {

'owner': 'airflow',

'start\_date': pendulum.datetime(2025, 1, 1, tz="UTC")

,

'retries': 1,

}

# Define the DAG

dag = DAG(

's3\_read\_and\_print\_csv',

default\_args=default\_args,

description='DAG to read CSV from S3 and print the values',

schedule=None, # Set to None for manual execution

)

# AWS S3 connection details

BUCKET\_NAME = 'airflowbuckets25'

OBJECT\_KEY = 'awsdata/user.csv'

#LOCAL\_FILE\_PATH = '/usr/local/airflow/include/'

# Function to read the CSV from S3 and print its values

def read\_and\_print\_csv\_from\_s3():

# Use S3Hook to interact with S3

s3\_hook = S3Hook(aws\_conn\_id='aws\_s3') # Use your AWS connection ID

# Fetch the file from S3 as a byte stream

file\_obj = s3\_hook.get\_key(key=OBJECT\_KEY, bucket\_name=BUCKET\_NAME)

if file\_obj:

# Read the content into a pandas DataFrame

csv\_content = file\_obj.get()["Body"].read()

df = pd.read\_csv(io.BytesIO(csv\_content)) # Read the CSV from byte stream

# Print the DataFrame content

logging.info(df)

else:

logging.info(f"File {OBJECT\_KEY} not found in bucket {BUCKET\_NAME}")

wait\_for\_file = S3KeySensor(

task\_id='wait\_for\_s3\_file',

bucket\_name=BUCKET\_NAME,

bucket\_key=OBJECT\_KEY,

aws\_conn\_id='aws\_s3', # Use your AWS connection ID

poke\_interval=60, # How often to check the S3 bucket (in seconds)

timeout=600, # How long to wait before giving up (in seconds)

mode='poke', # This mode will keep checking until the file is found

dag=dag,

)

# Task to read and print CSV content

read\_csv\_task = PythonOperator(

task\_id='read\_and\_print\_csv',

python\_callable=read\_and\_print\_csv\_from\_s3,

dag=dag,

)

wait\_for\_file >> read\_csv\_task

[s3://airflowbuckets25/awsdata/](https://us-east-1.console.aws.amazon.com/s3/buckets/airflowbuckets25?region=us-east-1&bucketType=general&prefix=awsdata/)user.csv

apache-airflow-providers-amazon

pandas

AWS Access Key : AKIA4E74WJIUTPMAQ26N

AWS Secret key : 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

from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator