DAG Structure and Operators

Estimated time needed: 15 minutes

Introduction

Apache Airflow is a Python framework that helps create workflows using multiple technologies using both CLI and a user-friendly WebUI. An Apache Airflow Directed Acyclic Graph (DAG) is a Python program where you define the tasks and the pipeline with the order in which the tasks will be executed.

Objectives

After completing this reading, you'll be able to:

- Explain the structure of Directed Acyclic Graphs
- Categorize the operators that you can use with the DAGs
- Identify DAG arguments
- Describe how to create tasks for a DAG
- Explain how to define the dependencies for the tasks

Airflow operator for task definition

Airflow offers a wide range of operators, including many that are built into the core or are provided by pre-installed providers. Some popular core operators include:

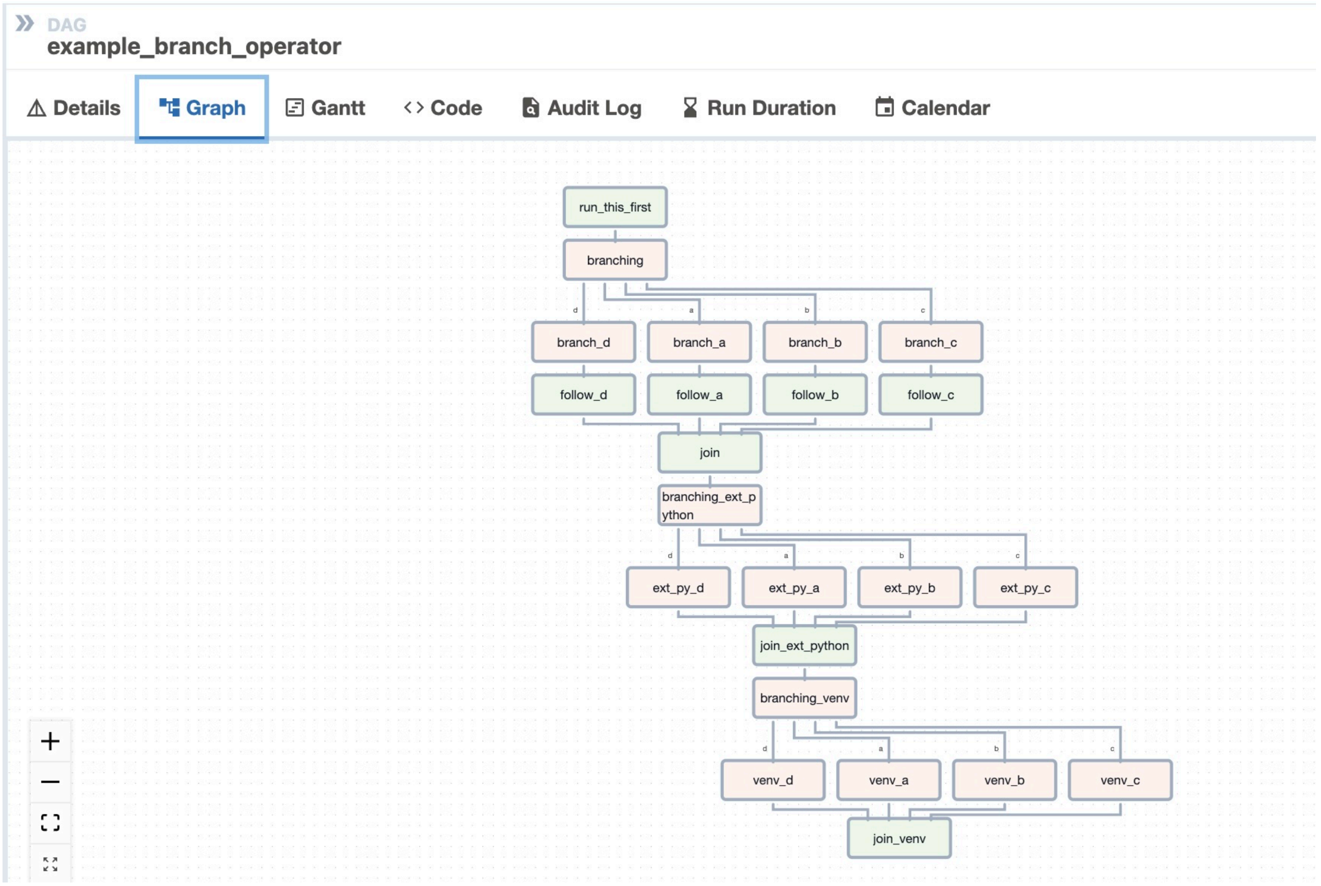
- BashOperator - executes a bash command

- PythonOperator - calls an arbitrary Python function

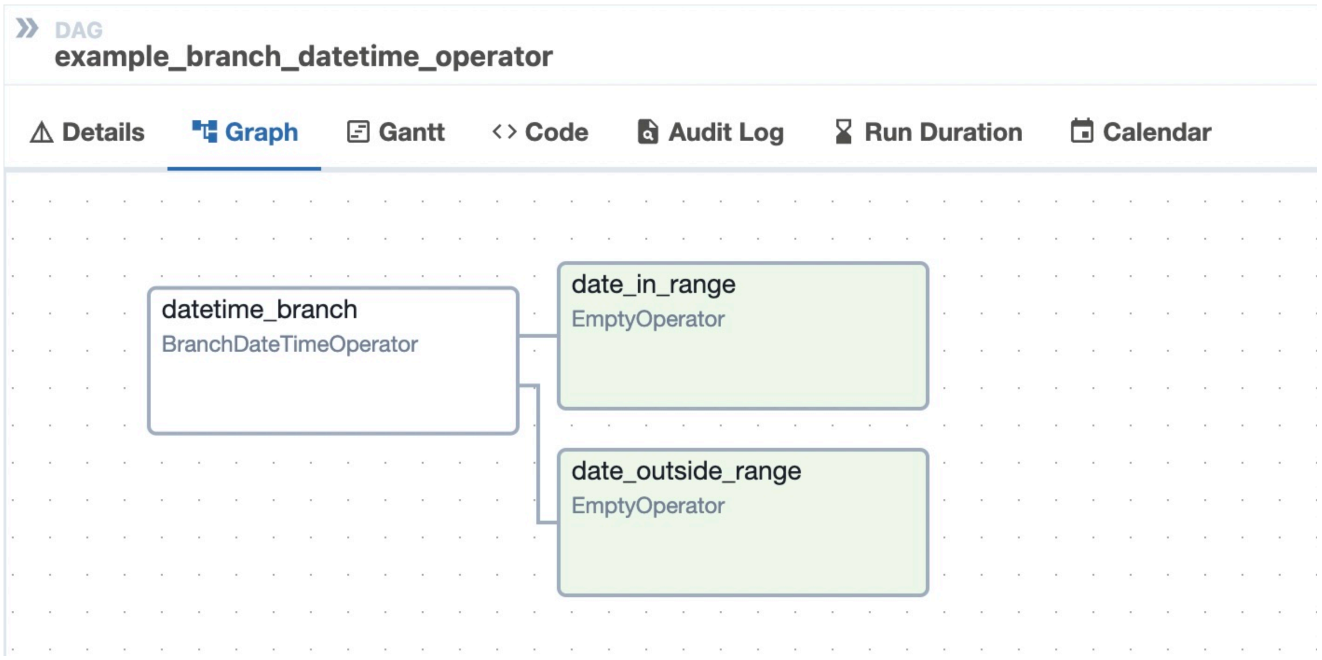
- EmailOperator - sends an email

The other core operators available include:

- BaseBranchOperator - A base class for creating operators with branching functionality



- BranchDateTimeOperator



- EmptyOperator - Operator that does nothing
- GenericTransfer - Moves data from on database connection to another.
- LatestOnlyOperator - Skip tasks that are not running during the most recent schedule interval.
- TriggerDagRunOperator - Triggers a DAG run for a specified dag_id.

Besides these, there are also many community provided operators. Some of the popular and useful ones are:

- HttpOperator
- MySqlOperator
- PostgresOperator
- MsSqlOperator
- OracleOperator
- JdbcOperator
- DockerOperator
- HiveOperator
- S3FileTransformOperator
- PrestoToMySqlOperator
- SlackAPIOperator

In addition to operators, you also have sensors and decorators that allow you to combine bash and Python. You can find more information regarding the same in this link.

Anatomy of a DAG

A DAG consists of these logical blocks.

- Imports
- DAG Arguments
- DAG Definition
- Task Definitions
- Task Pipeline

`imports` block example

```
# import the libraries
from datetime import timedelta
# The DAG object; we'll need this to instantiate a DAG
from airflow.models import DAG

# Operators; you need this to write tasks!
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python import PythonOperator
from airflow.operators.email import EmailOperator
```

Copied!

`DAG Arguments` block example

```
#defining DAG arguments

# You can override them on a per-task basis during operator initialization
default_args = {
    'owner': 'Your name',
    'start_date': days_ago(0),
    'email': ['youemail@somemail.com'],
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}
```

Copied!

DAG arguments are like the initial settings for the DAG.

The above settings mention:

- The owner name
- When this DAG should run from: days_ago(0) means today
- The email address where the alerts are sent to
- The number of retries in case of failure
- The time delay between retries

The other options that you can include are:

- 'queue': The name of the queue the task should be a part of
- 'pool': The pool that this task should use
- 'email_on_failure': Whether an email should be sent to the owner on failure
- 'email_on_retry': Whether an email should be sent to the owner on retry
- 'priority_weight': Priority weight of this task against other tasks.
- 'end_date': End date for the task
- 'wait_for_downstream': Boolean value indicating whether it should wait for downtime
- 'sla': Time by which the task should have succeeded. This can be a timedelta object
- 'execution_timeout': Time limit for running the task. This can be a timedelta object
- 'on_failure_callback': Some function, or list of functions to call on failure
- 'on_success_callback': Some function, or list of functions to call on success
- 'on_retry_callback': Another function, or list of functions to call on retry
- 'sla_miss_callback': Yet another function, or list of functions when 'sla' is missed
- 'on_skipped_callback': Some function to call when the task is skipped
- 'trigger_rule': Defines the rule by which the generated task gets triggered

`DAG definition` block example

```
# define the DAG
dag = DAG(
    dag_id='unique_id_for_DAG',
    default_args=default_args,
    description='A simple description of what the DAG does',
    schedule_interval=timedelta(days=1),
)
```

Copied!

Here you are creating a variable named dag by instantiating the DAG class with the following parameters:

`unique_id_for_DAG` is the ID of the DAG. This is what you see on the web console. This is what you can use to trigger the DAG using a TriggerDagRunOperator.

You are passing the dictionary `default_args`, in which all the defaults are defined.

`description` helps us in understanding what this DAG does.

`schedule_interval` tells us how frequently this DAG runs. In this case every day. (`days=1`).

`task definitions` block example

The tasks can be defined using any of the operators that have been imported.

```
# define the tasks

# define a task with BashOperator
task1 = BashOperator(
    task_id='unique_task_id',
    bash_command='<some bashcommand>',
    dag=dag,
)

# define a task with PythonOperator
task2 = PythonOperator(
    task_id='bash_task',
    python_callable=<the python function to be called>,
    dag=dag,
)

# define a task with EmailOperator
task3 = EmailOperator(
    task_id='mail_task',
    to='recipient@example.com',
    subject='Airflow Email Operator example',
    html_content='<p>This is a test email sent from Airflow.</p>',
    dag=dag,
)
```

Copied!

A task is defined using:

- A task_id which is a string that helps in identifying the task
- The dag this task belongs to
- The actual task to be performed
  - The bash command it represents in case of BashOperator
  - The Python callable function in case of a PythonOperator
  - Details of the sender, subject of the mail and the mail text as HTML in case of EmailOperator

`task pipeline` block example

```
# task pipeline
task1 >> task2 >> task3
```

Copied!

You can also use upstream and downstream to define the pipeline. For example:

```
task1.set_downstream(task2)
task3.set_upstream(task2)
```

Copied!

Task pipeline helps us to organize the order of tasks. In the example, the task `task1` must run first, followed by `task2`, followed by the task `task3`.

Author(s)

Lavanya T S