DAGs:
The Definitive Guide

Everything you need to know about Airflow DAGs

Powered by Astronomer

Editor's Note

Welcome to the ultimate guide to Apache Airflow DAGs, brought to you by the Astronomer team. Here we've gathered the most popular DAG content from our website—everything you need to know as a data engineer. From a walkthrough on DAG building blocks, design, and best practices, to tips on dynamically generating DAGs, testing, debugging, and more! This ebook was created by practitioners for practitioners.

Enjoy!

P.S. Don't miss the info about our brand new Certification for Apache Airflow DAG Authoring!

Like what you're reading? Share it with others!

Table of Contents

1. DAGs Where to Begin?

What Exactly is a DAG?

> *If you work in the world of data engineering, you're probably familiar with one of two terms: Data Pipeline (as an analyst/marketer/business-focused person) or DAG (as an engineer who favors term accuracy over branding). Though used in different circles, these terms both represent the same mechanism.*

"Data pipeline" describes the general process by which data moves from one system into another. By using the metaphor of plumbing, it helps illuminate an otherwise complex process. But data isn't literally in a single tube starting on one side and coming out of the other.

2. DAG Building Blocks

Operators 101

Operators are the main building blocks of Airflow DAGs. They are classes that encapsulate logic to do a unit of work.
When you create an instance of an operator in a DAG and provide it with its required parameters, it becomes a task. Many tasks can be added to a DAG along with their dependencies. When Airflow executes that task for a given `execution_date`, it becomes a task instance.

To browse and search all of the available Operators in Airflow, visit the **Astronomer Registry**. The following are examples of Operators that are frequently used in Airflow projects.

BashOperator

```
t1 = BashOperator(
        task_id='bash_hello_world',
        dag=dag,
        bash_command='echo "Hello World"'
        )
```

This **BashOperator** simply runs a bash command and echos `"Hello World"`.

Github: **BashOperator Code**

✦ Sensors 101

Sensors are a special kind of Operator. When they run, they check to see if a certain criterion is met before they let downstream tasks execute. This is a great way to have portions of your DAG wait on some external check or process to complete.

To browse and search all of the available Sensors in Airflow, visit the Astronomer Registry. Take the following sensor as an example:

S3 Key Sensor

```
s1 = S3KeySensor(
        task_id='s3_key_sensor',
        bucket_key='{{ ds_nodash }}/my_file.csv',
        bucket_name='my_s3_bucket',
        aws_conn_id='my_aws_connection',
    )
```

The **S3KeySensor** checks for the existence of a specified key in S3 every few seconds until it finds it or times out. If it finds the key, it will be marked as a success and allow downstream tasks to run. If it times out, it will fail and prevent downstream tasks from running.

S3KeySensor Code

Sensor Params

There are sensors for many use cases, such as ones that check a database for a certain row, wait for a certain time of day, or sleep for a certain amount of time. All sensors inherit from the **BaseSensorOperator** and have 4 parameters you can set on any sensor.

DAG Validation Testing

DAG validation tests are designed to ensure that your DAG objects are defined correctly, acyclic, and free from import errors.

These are things that you would likely catch if you were starting with the local development of your DAGs. But in cases where you may not have access to a local Airflow environment or want an extra layer of security, these tests can ensure that simple coding errors don't get deployed and slow down your development.

DAG validation tests apply to all DAGs in your Airflow environment, so you only need to create one test suite.
To test whether your DAG can be loaded, meaning there aren't any syntax errors, you can run the Python file:

```
python your-dag-file.py
```

Or to test for import errors specifically (which might be syntax related but could also be due to incorrect package import paths, etc.), you can use something like the following:

```
import pytest
from airflow.models import DagBag

def test_no_import_errors():
  dag_bag = DagBag()
  assert len(dag_bag.import_errors) == 0, "No Import Failures"
```

Customizing Email Notifications

By default, email notifications will be sent in a standard format as defined in the `email_alert()` and `get_email_subject_content()` methods of the `TaskInstance` class. The default email content is defined like this:

```
default_subject = 'Airflow alert: {{ti}}'
# For reporting purposes, we report based on 1-indexed,
# not 0-indexed lists (i.e. Try 1 instead of
# Try 0 for the first attempt).
default_html_content = (
    'Try {{try_number}} out of {{max_tries + 1}}<br>'
    'Exception:<br>{{exception_html}}<br>'
    'Log: <a href="{{ti.log_url}}">Link</a><br>'
    'Host: {{ti.hostname}}<br>'
    'Mark success: <a href="{{ti.mark_success_url}}">Link</a><br>'
)
```

To see the full method, check out the source code **here**.

You can overwrite this default with your custom content by setting the `subject_template` and/or `html_content_template` variables in your `airflow.cfg` with the path to your jinja template files for subject and content respectively.