



Manage Data Pipelines with Apache Airflow

Use Apache Airflow to build and monitor better data pipelines.

Apache

 Todd Birchard

It seems like almost every data-heavy Python shop is using Airflow in some way these days. It shouldn't take much time in Airflow's interface to figure out why: Airflow is the missing piece data engineers need to standardize the creation of ETL pipelines. The best part of Airflow, of course, is that it's one of the rare projects donated to the Apache foundation which is written in Python. Hooray!

If you happen to be a data engineer who isn't using Airflow (or equivalent) yet, you're in for a treat. It won't take much time using Airflow before you wonder how you managed to get along without it.

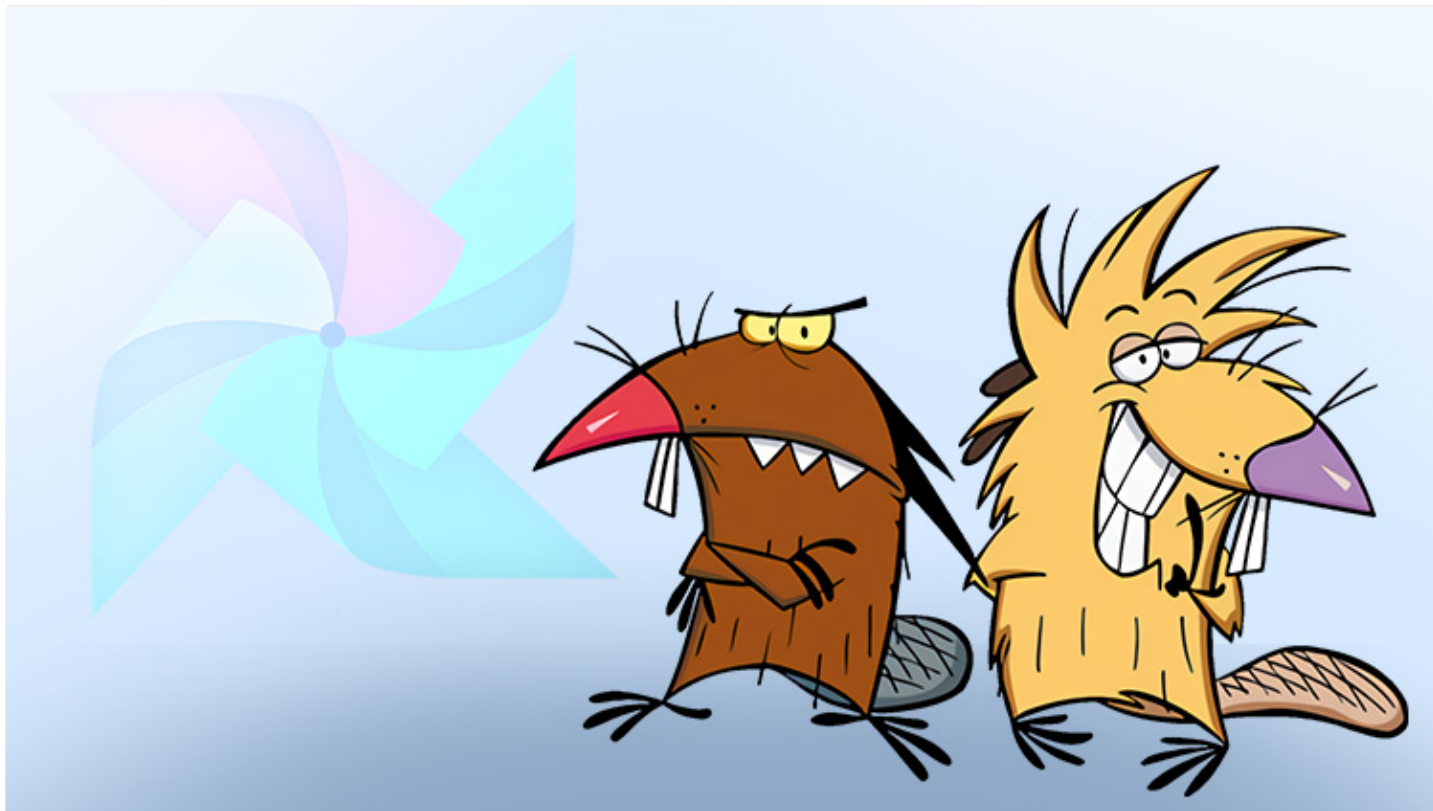
What's the Point of Airflow?

Airflow provides countless benefits to those in the pipeline business. It's not too crazy to group these benefits into two main categories: *code quality* and *visibility*.

Airflow provides us with a better way to build data pipelines by serving as a sort of 'framework' for creating pipelines. In the same way a web framework might help developers by abstracting common patterns, Airflow does the same by providing data engineers with tools to trivialize certain repetitive aspects of pipeline creation. Airflow comes with numerous powerful integrations that serve almost any need when it comes to outputting data. By leveraging these tools, engineers begin to see their pipelines abiding by a well-understood format, making code readable to others.

The more obvious benefits of Airflow are centered around its powerful GUI. Wrangling multiple pipelines which are prone to failure might be the least glorious aspect of any data engineer's job. By creating our pipelines within Airflow, we gain immediate visibility across *all* our pipelines to quickly spot areas of failure. Even more impressive is that the code we write is *visually represented* in Airflow's GUI. Not only can we check the heartbeat of our pipelines, but we can also view graphical representations of the very code we write.

To get started with Airflow, we should stop throwing the word "pipeline" around. Instead, get used to saying **DAG**.



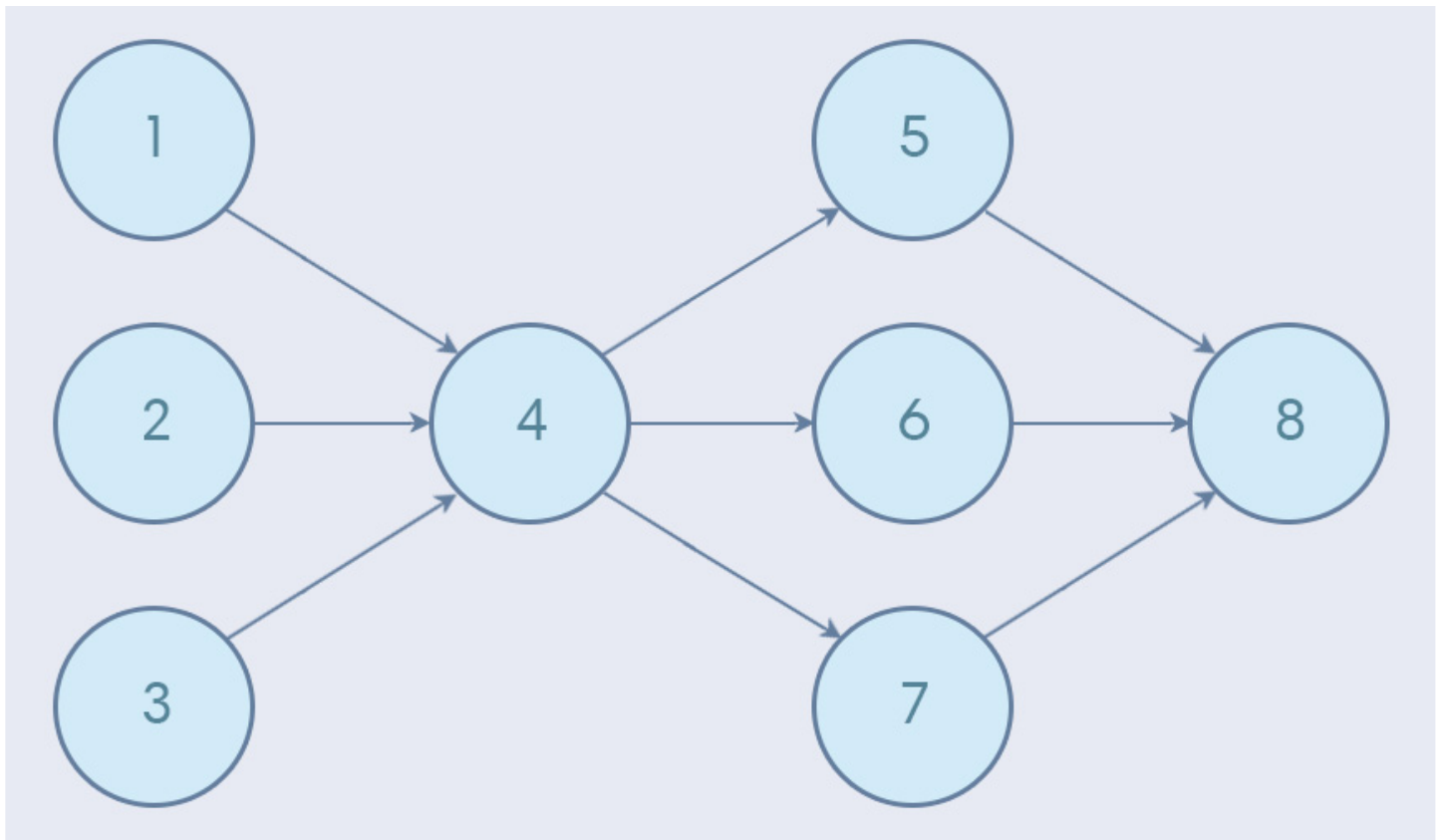
The OG Dag.

What is a DAG?

Airflow refers to what we've been calling "pipelines" as **DAGs** (directed acyclic graphs). In computer science, a *directed acyclic graph* simply means a workflow which only flows in a single direction. Each "step" in the workflow (an *edge*) is reached via the previous step in the workflow until we reach the beginning. The connection of edges is called a *vertex*.

If this remains unclear, consider how nodes in a tree data structure relate to one another. Every node has a "parent" node, which of course means that a child node cannot be its parents' parent. That's it - there's no need for fancy language here.

Edges in a DAG can have numerous "child" edges. Interestingly, a "child" edge can *also* have multiple parents (this is where our tree analogy fails us). Here's an example:



An example DAG structure.

In the above example, the DAG begins with edges 1, 2 and 3 kicking things off. At various points in the pipeline, information is consolidated or broken out. Eventually, the DAG ends with edge 8.

We'll dig deeper into DAGs, but first, let's install Airflow.

Installing Airflow

Installing Apache's data services is typically an awful experience. In most cases, things start out by installing some highly specific version of Java after getting harassed to create an Oracle account (please kill me). Once that's done, you usually need to install and configure three or four different Apache services with obnoxious animal-themed names.

Setting up Airflow is refreshingly easy. To get started with a barebones Airflow setup, all we need is to install the **apache-airflow** Python library:

```
$ pip3 install apache-airflow
```

Installing Airflow on its own is fine for testing the waters, but in order to build something somewhat meaningful, we'll need to install one of Airflow's many "extra features". Each Airflow *feature* we install enables a built-in integration between Airflow and a service, most commonly a database. Airflow installs an SQLite *feature* by default.

Airflow needs a database to create tables necessary for running Airflow. Chances are we don't be using a local SQLite database when we use Airflow in production, so I've opted to use a Postgres database:

```
$ pip3 install apache-airflow[postgres]
$ pip3 install psycopg2-binary
```

Airflow leverages the familiar SQLAlchemy library to handle database connections. As a result, the act of setting database connection strings should all be familiar.

Airflow has features for much more than just databases. Some features which can be installed with airflow include Redis, Slack, HDFS, RabbitMQ, and a whole lot more. To see everything available, check out the list: <https://airflow.apache.org/installation.html>

Basic Airflow Configuration

Before we do anything, we need to set an important environment variable called `AIRFLOW_HOME`. When we initiate Airflow, it's going to look for a folder which matches the name set as the value of this variable. Then, it's going to unpack a bunch of core files needed to run Airflow into said folder:

```
AIRFLOW_HOME=./airflow
```

Next we need to unpack the Airflow configuration files into **/airflow**. This is as simple as running the following command:

```
$ airflow initdb
```

A bunch of new files should magically appear in your **/airflow** directory, like this:

```
./
├── /airflow
│   ├── airflow.cfg
│   ├── airflow.db
│   ├── /logs
│   │   └── /scheduler
│   │       ├── 2019-06-01
│   │       └── latest → airflow/logs/scheduler/2019-06-01
│   └── unittests.cfg
└── requirements.txt
```

Running `initdb` for the first time defaults to creating an Airflow instance pointing to a local SQLite database. Once we run it the first time, we can now change the variables found in our new `./airflow/airflow.cfg` file to point to a Postgres database (or database of your choosing).

Database Configuration

To hook Airflow up to a live database, we need to modify a couple of settings in `airflow.cfg`. Look for the `sql_alchemy_conn` variable and paste an SQLAlchemy connection string for your database of choice. If you're using Postgres, don't forget to set `sql_alchemy_schema` as well:

```
sql_alchemy_conn = postgresql+psycopg2://[username]:[password]@[host]:[port]/[database]
sql_alchemy_schema = public
```

With those changes made, initialize your database again:

```
$ airflow initdb
```

This time around, Airflow should initialize your provided database with a bunch of tables necessary for running the application. I decided to take a peek in my database to see for myself:

Search for item: *name\$...

Items

Favorites

History

▼ Functions

▼ Tables

alembic_version

chart

connection

dag

dag_pickle

dag_run

import_error

job

known_event

known_event_type

kube_resource_version

kube_worker_uuid

log

sla_miss

slot_pool

task_fail

task_instance

task_reschedule

test

users

variable

xcom

id	conn_id	conn_type	host	schema
1	airflow_db	mysql	mysql	airflow
2	beeline_default	beeline	localhost	default
3	bigquery_default	google_cloud_platform	NULL	default
4	local_mysql	mysql	localhost	airflow
5	presto_default	presto	localhost	hive
6	google_cloud_default	google_cloud_platform	NULL	default
7	hive_cli_default	hive_cli	NULL	default
8	hiveserver2_default	hiveserver2	localhost	default
9	metastore_default	hive_metastore	localhost	NULL
10	mongo_default	mongo	mongo	NULL
11	mysql_default	mysql	mysql	airflow
12	postgres_default	postgres	postgres	airflow
13	sqlite_default	sqlite	/tmp/sqlite_default.db	NULL
14	http_default	http	https://www.google.com/	NULL
15	mssql_default	mssql	localhost	NULL
16	vertica_default	vertica	localhost	NULL
17	wasb_default	wasb	NULL	NULL
18	webhdfs_default	hdfs	localhost	NULL
19	ssh_default	ssh	localhost	NULL
20	sftp_default	sftp	localhost	NULL
21	fs_default	fs	NULL	NULL
22	aws_default	aws	NULL	NULL
23	spark_default	spark	yarn	NULL
24	druid_broker_default	druid	druid-broker	NULL
25	druid_ingest_default	druid	druid-overlord	NULL
26	redis_default	redis	redis	NULL
27	sqoop_default	sqoop	rmdbs	NULL
28	emr_default	emr	NULL	NULL
29	databricks_default	databricks	localhost	NULL
30	qubole_default	qubole	localhost	NULL
31	segment_default	segment	NULL	NULL
32	azure_data_lake_default	azure_data_lake	NULL	NULL
33	azure_cosmos_default	azure_cosmos	NULL	NULL

Tables that Airflow creates.

Launch Airflow

With everything configured, run the following to kick up Airflow at port 8080:

```
$ airflow webserver -p 8080
```

The output should look something like this:

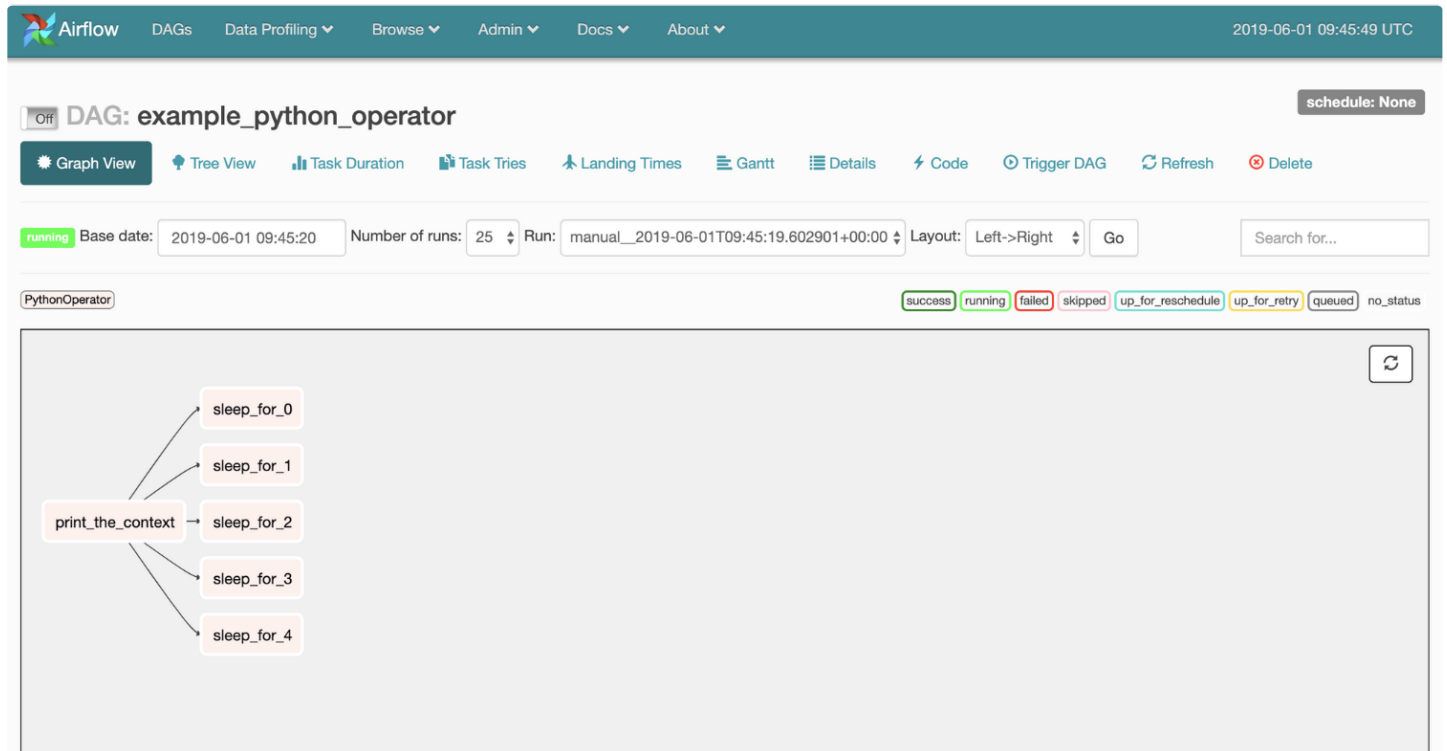
```

_____|__|_()_____ _/_ _ /_____ _
_____/| | _/_ _ _/_/_ _/_ _ \_ | /| //
____ _ _ | / _ / _ _/_/_ _///_/_ | /| /
_/_/_ |/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/
[2019-06-01 04:38:27,785] {__init__.py:305} INFO - Filling up the DagBag from airflow/dags
Running the Gunicorn Server with:
Workers: 4 sync
Host: 0.0.0.0:8080
Timeout: 120
```

Logfiles: - -

=====

Now let's see what's going on at **localhost:8080**:



Default Airflow instance running at localhost:8080

Sweet! Airflow is kind enough to create a bunch of example DAGs for us to poke around in. These examples are a pretty good starting point for becoming acquainted.

Before we get too crazy, let's break down the elements of the screen above:

- ◇ **DAG:** Name of a DAG job.
- ◇ **Schedule:** The reoccurring CRON schedule for running the current DAG.
- ◇ **Owner:** Name of the user within your Airflow instance who owns the job.
- ◇ **Recent Tasks:** Visual status (pass/fail/running) of the last 10 times this job ran.
- ◇ **Last run:** Time the DAG was run last.
- ◇ **DAG Runs:** Total number of times a DAG has been executed.

Feel free to click around and break some things while you have the chance.

Anatomy of an Airflow DAG

Any pipeline is essentially just a chain of tasks, and DAGs are no different. Within DAGs, our "tasks" are defined by **operators**.

Let's take a step back: DAGs are our workflows, and tasks within a DAG are the *actions* taken by said workflow. An "operator" refers to the *type* of action a task belongs to, such as a database action or a script action. It isn't crazy to imagine we might have multiple database-related actions in a single DAG: in this case, we'd use the same operator to define multiple tasks (`PostgresOperator` , assuming we're dealing with Postgres).

Here are some different types of operators:

- ◇ `BashOperator` : Execute a bash command.
- ◇ `PythonOperator` : Call a Python function.
- ◇ `EmailOperator` : Send an email.
- ◇ `SimpleHttpOperator` : Generate an HTTP request.
- ◇ `MySqlOperator` , `SqliteOperator` , `PostgresOperator` : Execute a SQL command.
- ◇ `DummyOperator` : Operator that does nothing (basically for testing purposes).

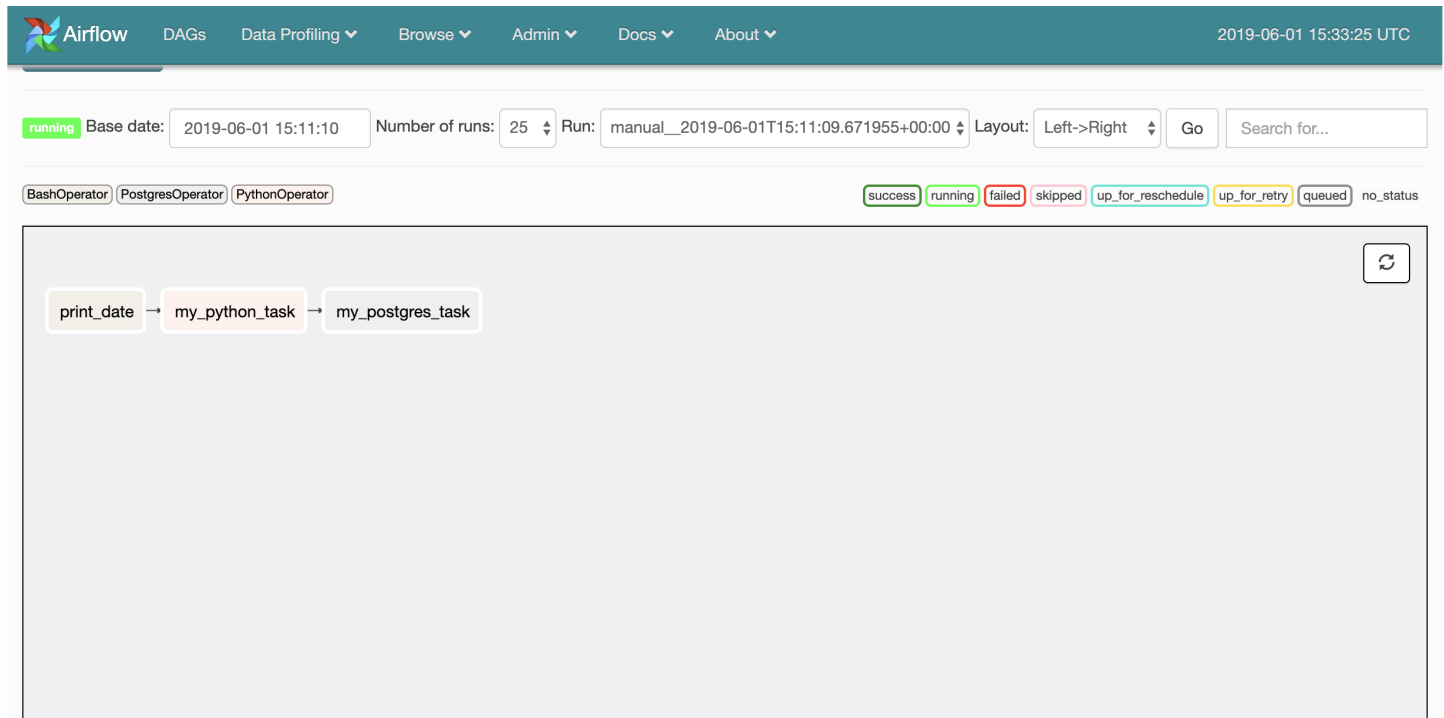
There's also:

- ◇ `DingdingOperator` : Sends a message to the "Dingding" Alibaba message service. I didn't know what this was either.
- ◇ **Google Cloud Operators**: There are a *ton* of operators specific to Google cloud services, such as Bigtable operators, Compute Engine operators, Cloud function operators, etc.
- ◇ **Sensor**: Waits for an event to occur before moving forward (can wait for time to elapse, for a script to finish, etc).

Operators can occur at any point in a DAG. We can fire multiple operators throughout the lifetime of a single DAG at any point.

DAGs By Example

Let's explore some of the example DAGs Airflow has provided us. A good place to start is **`example_python_operator`**:

Graph view of **example_python_operator**

Here I'm checking out the **Graph View** tab of a DAG: this view is the best representation of what's happening from start to finish.

This seems to be a simple DAG: it's just spinning up 5 Python operators which trigger a sleep timer, and nothing else. Run this DAG by clicking the **Trigger DAG** item in our menu bar, and check the output in whichever console you used to launch Airflow:

```
[2019-06-01 06:07:00,897] {__init__.py:305} INFO - Filling up the DagBag from /Users/toddbirchard/.local/sha:
example_dags/example_python_operator.py
127.0.0.1 - - [01/Jun/2019:06:07:01 -0400] "POST /admin/airflow/trigger?dag_id=example_python_operator&origin=1.1" 302 307 "http://0.0.0.0:8080/admin/airflow/code?dag_id=example_python_operator" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3729.169 Safari/537.36"
127.0.0.1 - - [01/Jun/2019:06:07:01 -0400] "GET /admin/airflow/tree?dag_id=example_python_operator HTTP/1.1" 200 1000 "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3729.169 Safari/537.36"
```

This doesn't really tell us anything, but then again... isn't that what we'd expect? After all, there aren't any steps which occur after the "sleep" Python operators, so nothingness might be a success here.

To get a better idea, check out the **code** tab. That's right: we can investigate the source code for any DAG right from our GUI!

```
from __future__ import print_function

import time
from builtins import range
from pprint import pprint

import airflow
```

```

from airflow.models import DAG
from airflow.operators.python_operator import PythonOperator

args = {
    'owner': 'airflow',
    'start_date': airflow.utils.dates.days_ago(2),
}

dag = DAG(
    dag_id='example_python_operator',
    default_args=args,
    schedule_interval=None,
)

# [START howto_operator_python]
def print_context(ds, **kwargs):
    pprint(kwargs)
    print(ds)
    return 'Whatever you return gets printed in the logs'

run_this = PythonOperator(
    task_id='print_the_context',
    provide_context=True,
    python_callable=print_context,
    dag=dag,
)
# [END howto_operator_python]

# [START howto_operator_python_kwargs]
def my_sleeping_function(random_base):
    """This is a function that will run within the DAG execution"""
    time.sleep(random_base)

# Generate 5 sleeping tasks, sleeping from 0.0 to 0.4 seconds respectively
for i in range(5):
    task = PythonOperator(
        task_id='sleep_for_' + str(i),
        python_callable=my_sleeping_function,
        op_kwargs={'random_base': float(i) / 10},
        dag=dag,
    )

    run_this >> task
# [END howto_operator_python_kwargs]

```

Every DAG starts out with some basic configuration variables. `args` contains high-level configuration values:

- ◇ `owner`: The Airflow user the DAG belongs to (again).
- ◇ `start_date`: The time at which the DAG should execute.
- ◇ `email`: An email address for alert notifications when something goes wrong.

- ◇ `email_on_failure` : When **True**, a failed execution will email the specified email address with details of the failed job.
- ◇ `email_on_retry` : When **True**, an email will be sent every time the DAG attempts to retry a failed execution.
- ◇ `retries` : Number of times to retry the DAG in case of a failure.
- ◇ `retry_delay` : Time between retry attempts.
- ◇ `concurrency` : Number of processes to run the DAG.
- ◇ `depends_on_past` : If **True**, this task will depend on the success of the preceding task before executing.

After setting our DAG's configuration, the DAG is instantiated with `dag = DAG()`. We pass a few things into `DAG()` upon creation (like the args we set earlier). This is where we set the DAG's name and schedule time.

In our DAG's graph view, we saw tasks named `print_the_context`, and a bunch of tasks following a convention like `sleep_for_#`. In the source code, we can see exactly where these task names are being defined! Check out the first task:

```
# [START howto_operator_python]
def print_context(ds, **kwargs):
    pprint(kwargs)
    print(ds)
    return 'Whatever you return gets printed in the logs'

run_this = PythonOperator(
    task_id='print_the_context',
    provide_context=True,
    python_callable=print_context,
    dag=dag,
)
```

The first task is being set inside `PythonOperator()` with an id equal to `print_the_context`. It's a function that prints some information and returns a string!

Here's the second group of tasks:

```
def my_sleeping_function(random_base):
    """This is a function that will run within the DAG execution"""
    time.sleep(random_base)

# Generate 5 sleeping tasks, sleeping from 0.0 to 0.4 seconds respectively
for i in range(5):
    task = PythonOperator(
        task_id='sleep_for_' + str(i),
        python_callable=my_sleeping_function,
```

```
op_kwargs={'random_base': float(i) / 10},  
dag=dag,  
)
```

This is super cool: this time, the Python operator is in a for loop. This is an excellent demonstration of how we can create DAGs dynamically!

Think you have what it takes to create your own DAG? I think you do too. Let's set the stage.

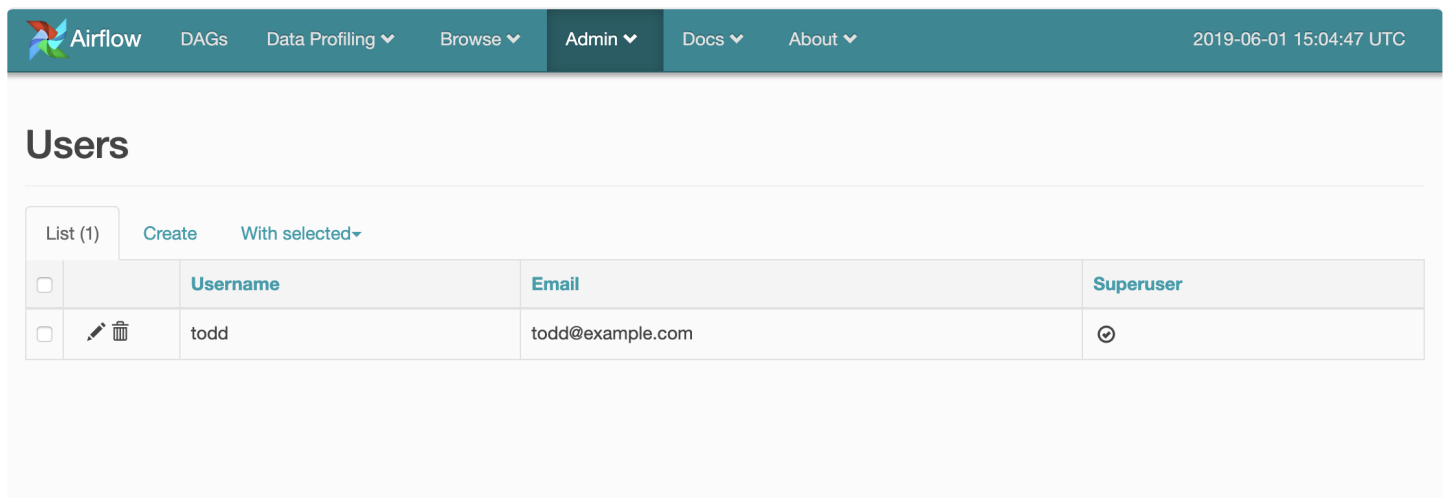
Getting Ready to Create Our Own DAG



There are a couple of housekeeping items we should knock out before moving forward. This won't take long.

Create an Airflow User

As we've already seen, DAGs need to have an "owner". The default DAGs we've seen so far set their user as **airflow**. We can't do this with the new DAGs we create -we need to set a legitimate user. In your Airflow UI, navigate to **Admin > Users**.

Then, create a user that will "own" our new DAG.




		Username	Email	Superuser
<input type="checkbox"/>	 	todd	todd@example.com	<input checked="" type="checkbox"/>

Airflow's user admin page.

Create a New Connection

To simulate a real-world scenario, we should have our DAG insert information into a database. Destinations for DAG output need to be created and managed in the Airflow UI, under **Admin > Connections**.

I've started setting up a Postgres connection below:


Airflow
DAGs
Data Profiling ▾
Browse ▾
Admin ▾
Docs ▾
About ▾
2019-06-01 14:46:46 UTC

Connection [create]

List
Create

Conn Id

Conn Type

Host

Schema

Login

Password

Port

Extra

Save

Save and Add Another

Save and Continue Editing

Cancel

Creating a new connection in Airflow.

Creating our First DAG

Shut down your web server and create a folder within **/airflow** called **/dags**. This will be where we store the source code for DAGs we create moving forward (the location of your DAGs can be changed in `airflow.cfg`, but **/dags** is the default). Create a Python file in **/dags** named something along the lines of **my_first_dag.py** (the name of the file doesn't matter tbh).

I'm going to create a DAG made up of three tasks. These tasks will use the **Bash**, **Python**, and **Postgres** operators:

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator
from airflow.operators.postgres_operator import PostgresOperator
from datetime import datetime, timedelta

default_args = {
    'owner': 'todd',
    'depends_on_past': False,
    'start_date': datetime(2015, 6, 1),
    'email': ['todd@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5)
}
```

```

dag = DAG(dag_id='my_custom_dag',
          default_args=default_args,
          schedule_interval=timedelta(days=1))

# Task 1
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag)

# Task 2
def my_python_function():
    now = datetime.now()
    response = 'This function ran at ' + str(now)
    return response

t2 = PythonOperator(
    task_id='my_python_task',
    python_callable=my_python_function,
    params={'my_param': 'Parameter I passed in'},
    dag=dag)

# Task 3
t3 = PostgresOperator(task_id='my_postgres_task',
                      sql="INSERT INTO test VALUES (3, 69, 'this is a test!');",
                      postgres_conn_id='my_postgres_instance',
                      autocommit=True,
                      database="airflow2",
                      dag=dag)

# Pipeline Structure
t2.set_upstream(t1)
t3.set_upstream(t2)

```

I've purposely set a bunch more arguments in `default_args` to demonstrate what this would look like. Note the value of the *owner* argument.

Check out the tasks we created:

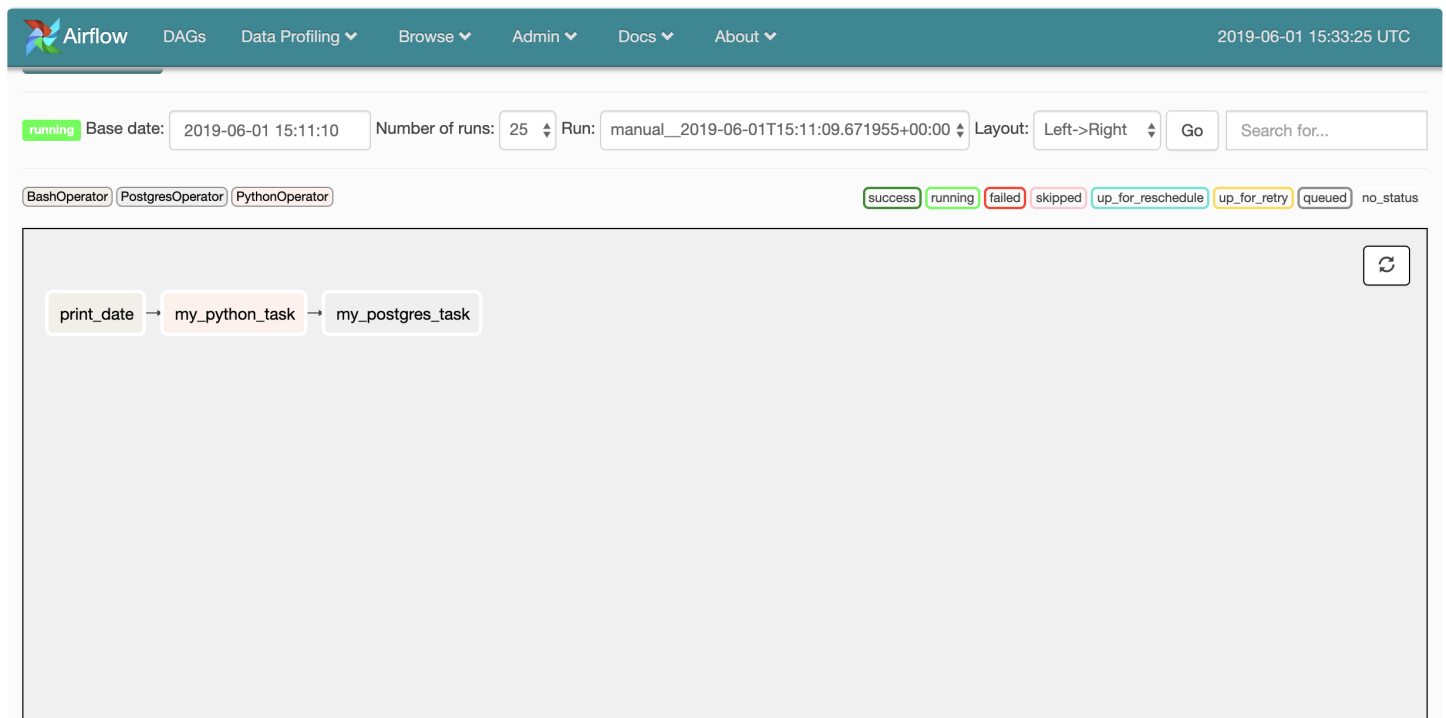
- ◇ **Task 1** is a simple bash function to print the date.
- ◇ **Task 2** returns the current time via a Python function.
- ◇ **Task 3** inserts a bunch of values into a Postgres Database (inserts 3 values: `3, 69, 'this is a test!'`).

The last part of our script is *muy importante*: this is where we set our pipeline structure. `set_upstream()` is one way we set the order of operations to occur: by calling `set_upstream()` on each task, it is inferred that t1 will be the first task. Forgetting to set your pipeline structure will result in tasks which don't run! There are a couple of other ways to set this:

- ◇ `set_downstream()` achieves the opposite of `set_upstream()`.

◇ `t1 >> t2 >> t3` is a cleaner way of handling a straightforward DAG like this one.

We should be able to see the structure of this DAG in the UI's graph view now:



Visualizing our pipeline in Airflow's UI.

Testing Our DAG

If you're like me, your DAG won't run the first time. This can be very frustrating. Luckily, there's an easy way to test tasks in our new DAG via the Airflow CLI. Simply enter the below:

```
airflow test [your_dag_id] [your_task_name_to_test] [today's_date]
```

This is what I entered to test the Postgres task:

```
airflow test my_custom_dag my_python_task 06/01/2019
```

After some adjustments, I was able to receive a success:

```
[2019-06-01 11:36:49,702] {__init__.py:1354} INFO - Starting attempt 1 of 2
[2019-06-01 11:36:49,702] {__init__.py:1355} INFO -
-----
[2019-06-01 11:36:49,702] {__init__.py:1374} INFO - Executing <Task(PythonOperator): my_python_task> on 2019-
[2019-06-01 11:36:50,059] {python_operator.py:104} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_ID=my_custom_dag
```



```
AIRFLOW_CTX_TASK_ID=my_python_task
```

```
AIRFLOW_CTX_EXECUTION_DATE=2019-06-01T00:00:00+00:00
```

```
[2019-06-01 11:36:50,060] {python_operator.py:113} INFO - Done. Returned value was: This function ran at 2019-
```

Surely enough, the record was created in my database!

Apache

Python

Data Engineering

ETL



Todd
Birchard's'
avatar

Todd Birchard

🏠 New York City

🌐 <http://toddbirchard.com>

🐦 @ToddRBirchard

Engineer with an ongoing identity crisis. Breaks everything before learning best practices. Completely normal and emotionally stable.

Becoming Familiar with Apache Kafka and Message Queues

Newsletter

Are you into data to the point where it's almost embarrassing? Toss us your email and we'll promise to only give you the good stuff.

SUBSCRIBE



Community of hackers obsessed with data science, data engineering, and analysis. Openly pushing a pro-robot agenda.



©2019 Hackers and Slackers, All Rights Reserved.

LINKS

[About Us](#)

[Series](#)

[Projects](#)

[Resources](#)

[Join The Club](#)

[Donate](#)

AUTHORS

[Todd Birchard](#)

[Matthew Alhonte](#)

[Max Mileaf](#)

[David Moore](#)

[Ryan Rosado](#)

[David Aquino](#)

[Graham Beckley](#)

TAGS

[Python](#)

[Data Engineering](#)

[Software Development](#)

[Data Science](#)

[DevOps](#)

[Architecture](#)

[Pandas](#)

[SQL](#)

[Data Analysis](#)