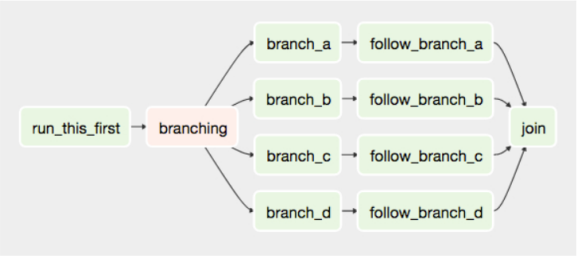
MAR 19TH, 2017

Get started developing workflows with Apache Airflow

Apache Airflow is an open-source tool for orchestrating complex computational workflows and data processing pipelines. If you find yourself running cron task which execute ever longer scripts, or keeping a calendar of big data processing batch jobs then Airflow can probably help you. This article provides an introductory tutorial for people who want to get started writing pipelines with Airflow.

An Airflow workflow is designed as a directed acyclic graph (DAG). That means, that when authoring a workflow, you should think how it could be divided into tasks which can be executed independently. You can then merge these tasks into a logical whole by combining them into a graph.



An example Airflow pipeline DAG

The shape of the graph decides the overall logic of your workflow. An Airflow DAG can include multiple branches and you can decide which of them to follow and which to skip at the time of workflow execution.

This creates a very resilient design, because each task can be retried multiple times if an error occurs. Airflow can even be stopped entirely and running workflows will resume by restarting the last unfinished task.

When designing Airflow operators, it's important to keep in mind that they may be executed more than once. Each task should be idempotent, i.e. have the ability to be applied multiple times without producing unintended consequences.

Airflow nomenclature

Here is a brief overview of some terms used when designing Airflow workflows:

- Airflow **DAGs** are composed of **Tasks**.
- Each Task is created by instantiating an **Operator** class. A configured instance of an Operator becomes a Task, as in: `my_task = MyOperator(...)`.
- When a DAG is started, Airflow creates a **DAG Run** entry in its database.
- When a Task is executed in the context of a particular DAG Run, then a **Task Instance** is created.
- `AIRFLOW_HOME` is the directory where you store your DAG definition files and Airflow plugins.

When?	DAG	Task	Info about other tasks
During definition	DAG	Task	`get_flat_relatives`
During a run	DAG Run	Task Instance	`xcom_pull`
Base class	DAG	`BaseOperator`	

Airflow documentation provides more information about these and other concepts.

Prerequisites

About me

Hi, my name is Michał and I'm a code geek. I work as a developer, project manager and systems architect. I also write.

My favorite languages are currently JavaScript and Python and I'm good with Django, Angular, ExtJS and other MVC frameworks.

If you'd like to chat or hire me for your next project, feel free to contact me.

Recent Posts

Top 20 EuroPython 2019 talks

Python project maturity checklist

EuroPython 2017 Presentation

Get started developing workflows with Apache Airflow

EuroPython 2016 Presentation

GitHub Repos

gym-demo
Explore OpenAI Gym environments

blog
Source code repository for my blog.

ngraph-ci-test

airflow_tutorial

travis_test

rest_api_demo
Boilerplate code for a RESTful API based on Flask-RESTPlus

har2grinder
Creates test files for The Grinder based on HTTP Archive (HAR) files generated by Chrome DevTools.

hello_django
A simple Django project for demonstration purposes

django-xmin
ExtJS Admin for Django

@postrational on GitHub

Airflow is written in Python, so I will assume you have it installed on your machine. I'm using Python 3 (because it's 2017, come on people!), but Airflow is supported on Python 2 as well. I will also assume that you have virtualenv installed.

```
$ python3 --version
Python 3.6.0
$ virtualenv --version
15.1.0
```

Install Airflow

Let's create a workspace directory for this tutorial, and inside it a Python 3 virtualenv directory:

```
$ cd /path/to/my/airflow/workspace
$ virtualenv -p `which python3` venv
$ source venv/bin/activate
(venv) $
```

Now let's install Airflow 1.8:

```
(venv) $ pip install airflow==1.8.0
```

Now we'll need to create the `AIRFLOW_HOME` directory where your DAG definition files and Airflow plugins will be stored. Once the directory is created, set the `AIRFLOW_HOME` environment variable:

```
(venv) $ cd /path/to/my/airflow/workspace
(venv) $ mkdir airflow_home
(venv) $ export AIRFLOW_HOME=`pwd`/airflow_home
```

You should now be able to run Airflow commands. Let's try by issuing the following:

```
(venv) $ airflow version
  ____________       _____________
 ____    |__( )_________  __/__  /________      __
____  /| |_  /__  ___/_  /_ __  /_  __ \_ | /| / /
___  ___ |  / _  /   _  __/ _  / / /_/ /_ |/ |/ /
 _/_/  |_/_/  /_/    /_/    /_/  \____/____/|__/
   v1.8.0rc5+apache.incubating
```

If the `airflow version` command worked, then Airflow also created its default configuration file `airflow.cfg` in `AIRFLOW_HOME`:

```
airflow_home
├── airflow.cfg
└── unittests.cfg
```

Default configuration values stored in `airflow.cfg` will be fine for this tutorial, but in case you want to tweak any Airflow settings, this is the file to change. Take a look at the docs for more information about configuring Airflow.

Initialize the Airflow DB

Next step is to issue the following command, which will create and initialize the Airflow SQLite database:

```
(venv) $ airflow initdb
```

The database will be create in `airflow.db` by default.

```
airflow_home
├── airflow.cfg
├── airflow.db        <- Airflow SQLite DB
└── unittests.cfg
```

Using SQLite is an adequate solution for local testing and development, but it does not support concurrent access. In a production environment you will most certainly want to use a more robust database solution such as Postgres or MySQL.

Start the Airflow web server

Airflow's UI is provided in the form of a Flask web application. You can start it by issuing the command:

```
(venv) $ airflow webserver
```

You can now visit the Airflow UI by navigating your browser to port `8080` on the host where Airflow was started, for example: http://localhost:8080/admin/

Airflow comes with a number of example DAGs. Note that these examples may not work until you have at least one DAG definition file in your own `dags_folder`. You can hide the example DAGs by changing the `load_examples` setting in `airflow.cfg`.

Your first Airflow DAG

OK, if everything is ready, let's start writing some code. We'll start by creating a Hello World workflow, which does nothing other then sending "Hello world!" to the log.

Create your `dags_folder`, that is the directory where your DAG definition files will be stored in `AIRFLOW_HOME/dags`. Inside that directory create a file named `hello_world.py`.

```
airflow_home
├── airflow.cfg
├── airflow.db
├── dags                <- Your DAGs directory
|   └── hello_world.py  <- Your DAG definition file
└── unittests.cfg
```

Add the following code to `dags/hello_world.py`:

airflow_home/dags/hello_world.py

```
from datetime import datetime
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import PythonOperator

def print_hello():
    return 'Hello world!'

dag = DAG('hello_world', description='Simple tutorial DAG',
          schedule_interval='0 12 * * *',
          start_date=datetime(2017, 3, 20), catchup=False)

dummy_operator = DummyOperator(task_id='dummy_task', retries=3, dag=dag)

hello_operator = PythonOperator(task_id='hello_task', python_callable=print_hello, dag=dag)

dummy_operator >> hello_operator
```

This file creates a simple DAG with just two operators, the `DummyOperator`, which does nothing and a `PythonOperator` which calls the `print_hello` function when its task is executed.

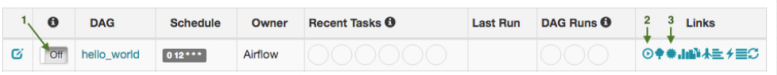
Running your DAG

In order to run your DAG, open a second terminal and start the Airflow scheduler by issuing the following commands:

```
$ cd /path/to/my/airflow/workspace
$ export AIRFLOW_HOME=`pwd`/airflow_home
$ source venv/bin/activate
(venv) $ airflow scheduler
```

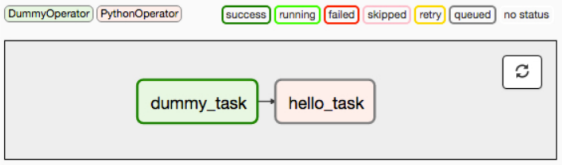
The scheduler will send tasks for execution. The default Airflow settings rely on an executor named `SequentialExecutor`, which is started automatically by the scheduler. In production you would probably want to use a more robust executor, such as the `CeleryExecutor`.

When you reload the Airflow UI in your browser, you should see your `hello_world` DAG listed in Airflow UI.



Hello World DAG in Airflow UI

In order to start a DAG Run, first turn the workflow on (arrow **1**), then click the **Trigger Dag** button (arrow **2**) and finally, click on the **Graph View** (arrow **3**) to see the progress of the run.



Hello World DAG Run - Graph View

You can reload the graph view until both tasks reach the status **Success**. When they are done, you can click on the `hello_task` and then click **View Log**. If everything worked as expected, the log should show a number of lines and among them something like this:

```
[2017-03-19 13:49:58,789] {base_task_runner.py:95} INFO - Subtask: --------------------------------
[2017-03-19 13:49:58,789] {base_task_runner.py:95} INFO - Subtask: Starting attempt 1 of 1
[2017-03-19 13:49:58,789] {base_task_runner.py:95} INFO - Subtask: --------------------------------
[2017-03-19 13:49:58,790] {base_task_runner.py:95} INFO - Subtask:
[2017-03-19 13:49:58,800] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 13:49:58,800] {model
[2017-03-19 13:49:58,818] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 13:49:58,818] {pytho
```

The code you should have at this stage is available in this commit on GitHub.

Your first Airflow Operator

Let's start writing our own Airflow operators. An Operator is an atomic block of workflow logic, which performs a single action. Operators are written as Python classes (subclasses of `BaseOperator`), where the `__init__` function can be used to configure settings for the task and a method named `execute` is called when the task instance is executed.

Any value that the `execute` method returns is saved as an Xcom message under the key `return_value`. We'll cover this topic later.

The `execute` method may also raise the `AirflowSkipException` from `airflow.exceptions`. In such a case the task instance would transition to the Skipped status.

If another exception is raised, the task will be retried until the maximum number of `retries` is reached.

Remember that since the `execute` method can retry many times, it should be idempotent.

We'll create your first operator in an Airflow plugin file named `plugins/my_operators.py`. First create the `airflow_home/plugins` directory, then add the `my_operators.py` file with the following content:

airflow_home/plugins/my_operators.py

```python
import logging

from airflow.models import BaseOperator
from airflow.plugins_manager import AirflowPlugin
from airflow.utils.decorators import apply_defaults

log = logging.getLogger(__name__)

class MyFirstOperator(BaseOperator):

    @apply_defaults
    def __init__(self, my_operator_param, *args, **kwargs):
        self.operator_param = my_operator_param
        super(MyFirstOperator, self).__init__(*args, **kwargs)

    def execute(self, context):
        log.info("Hello World!")
        log.info('operator_param: %s', self.operator_param)

class MyFirstPlugin(AirflowPlugin):
    name = "my_first_plugin"
    operators = [MyFirstOperator]
```

In this file we are defining a new operator named `MyFirstOperator`. Its `execute` method is very simple, all it does is log "Hello World!" and the value of its own single parameter. The parameter is set in the `__init__` function.

We are also defining an Airflow plugin named `MyFirstPlugin`. By defining a plugin in a file stored in the `airflow_home/plugins` directory, we're providing Airflow the ability to pick up our plugin and all the operators it defines. We'll be able to import these operators later using the line `from airflow.operators import MyFirstOperator`.

In the docs, you can read more about Airflow plugins.

Make sure your `PYTHONPATH` is set to include directories where your custom modules are stored.

Now, we'll need to create a new DAG to test our operator. Create a `dags/test_operators.py` file and fill it with the following content:

airflow_home/dags/test_operators.py

```python
from datetime import datetime
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators import MyFirstOperator

```

```
dag = DAG('my_test_dag', description='Another tutorial DAG',
          schedule_interval='0 12 * * *',
          start_date=datetime(2017, 3, 20), catchup=False)

dummy_task = DummyOperator(task_id='dummy_task', dag=dag)

operator_task = MyFirstOperator(my_operator_param='This is a test.',
                                task_id='my_first_operator_task', dag=dag)

dummy_task >> operator_task
```

Here we just created a simple DAG named `my_test_dag` with a `DummyOperator` task and another task using our new `MyFirstOperator`. Notice how we pass the configuration value for `my_operator_param` here during DAG definition.

At this stage your source tree will look like this:

```
airflow_home
├── airflow.cfg
├── airflow.db
├── dags
│   └── hello_world.py
│   └── test_operators.py  <- Second DAG definition file
├── plugins
│   └── my_operators.py    <- Your plugin file
└── unittests.cfg
```

All the code you should have at this stage is available in this commit on GitHub.

To test your new operator, you should stop (CTRL-C) and restart your Airflow web server and scheduler. Afterwards, go back to the Airflow UI, turn on the `my_test_dag` DAG and trigger a run. Take a look at the logs for `my_first_operator_task`.

Debugging an Airflow operator

Debugging would quickly get tedious if you had to trigger a DAG run and wait for all upstream tasks to finish before you could retry your new operator. Thankfully Airflow has the `airflow test` command, which you can use to manually start a single operator in the context of a specific DAG run.

The command takes 3 arguments: the name of the dag, the name of a task and a date associated with a particular DAG Run.

```
(venv) $ airflow test my_test_dag my_first_operator_task 2017-03-18T18:00:00.0
```

You can use this command to restart you task as many times as needed, while tweaking your operator code.

If you want to test a task from a particular DAG run, you can find the needed date value in the logs of a failing task instance.

Debugging an Airflow operator with IPython

There is a cool trick you can use to debug your operator code. If you install IPython in your venv:

```
(venv) $ pip install ipython
```

You can then place IPython's `embed()` command in your code, for example in the `execute` method of an operator, like so:

airflow_home/plugins/my_operators.py

```
def execute(self, context):
    log.info("Hello World!")

    from IPython import embed; embed()

    log.info('operator_param: %s', self.operator_param)
```

Now when you run the `airflow test` command again:

```
(venv) $ airflow test my_test_dag my_first_operator_task 2017-03-18T18:00:00.0
```

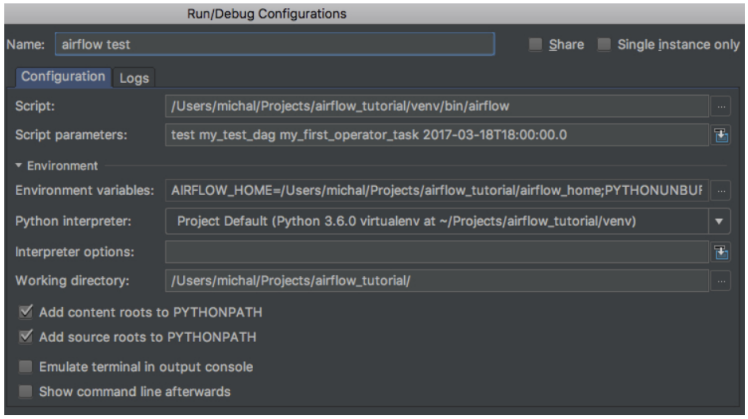
the task will run, but execution will stop and you will be dropped into an IPython shell, from which you can explore the place in the code where you placed `embed()`:

```
In [1]: context
Out[1]:
{'END_DATE': '2017-03-18',
 'conf': <module 'airflow.configuration' from '/path/to/my/airflow/workspace/venv/lib/python3.
 'dag': <DAG: my_test_dag>,
```

```
'ds': '2017-03-18',
'ds_nodash': '20170318',
'end_date': '2017-03-18',
'execution_date': datetime.datetime(2017, 3, 18, 18, 0),
'latest_date': '2017-03-18',
'macros': <module 'airflow.macros' from '/path/to/my/airflow/workspace/venv/lib/python3.6/sit
'next_execution_date': datetime.datetime(2017, 3, 19, 12, 0),
'params': {},
'prev_execution_date': datetime.datetime(2017, 3, 18, 12, 0),
'run_id': None,
'tables': None,
'task': <Task(MyFirstOperator): my_first_operator_task>,
'task_instance': <TaskInstance: my_test_dag.my_first_operator_task 2017-03-18 18:00:00 [runni
'task_instance_key_str': 'my_test_dag__my_first_operator_task__20170318',
'test_mode': True,
'ti': <TaskInstance: my_test_dag.my_first_operator_task 2017-03-18 18:00:00 [running]>,
'tomorrow_ds': '2017-03-19',
'tomorrow_ds_nodash': '20170319',
'ts': '2017-03-18T18:00:00',
'ts_nodash': '20170318T180000',
'var': {'json': None, 'value': None},
'yesterday_ds': '2017-03-17',
'yesterday_ds_nodash': '20170317'}

In [2]: self.operator_param
Out[2]: 'This is a test.'
```

You could of course also drop into Python's interactive debugger `pdb` (`import pdb; pdb.set_trace()`) or the IPython enhanced version `ipdb` (`import ipdb; ipdb.set_trace()`). Alternatively, you can also use an `airflow test` based run configuration to set breakpoints in IDEs such as PyCharm.



A PyCharm debug configuration

Code is in this commit on GitHub.

Your first Airflow Sensor

An Airflow Sensor is a special type of Operator, typically used to monitor a long running task on another system.

To create a Sensor, we define a subclass of `BaseSensorOperator` and override its `poke` function. The `poke` function will be called over and over every `poke_interval` seconds until one of the following happens:

- `poke` returns `True` – if it returns `False` it will be called again.
- `poke` raises an `AirflowSkipException` from `airflow.exceptions` – the Sensor task instance's status will be set to Skipped.
- `poke` raises another exception, in which case it will be retried until the maximum number of `retries` is reached.

There are many predefined sensors, which can be found in Airflow's codebase:

To add a new Sensor to your `my_operators.py` file, add the following code:

airflow_home/plugins/my_operators.py

```
from datetime import datetime
from airflow.operators.sensors import BaseSensorOperator

class MyFirstSensor(BaseSensorOperator):

    @apply_defaults
    def __init__(self, *args, **kwargs):
        super(MyFirstSensor, self).__init__(*args, **kwargs)

    def poke(self, context):
        current_minute = datetime.now().minute
        if current_minute % 3 != 0:
            log.info("Current minute (%s) not is divisible by 3, sensor will retry.", current_
```

```
            return False

        log.info("Current minute (%s) is divisible by 3, sensor finishing.", current_minute)
        return True
```

Here we created a very simple sensor, which will wait until the the current minute is a number divisible by 3. When this happens, the sensor's condition will be satisfied and it will exit. This is a contrived example, in a real case you would probably check something more unpredictable than just the time.

Remember to also change the plugin class, to add the new sensor to the `operators` it exports:

airflow_home/plugins/my_operators.py

```
class MyFirstPlugin(AirflowPlugin):
    name = "my_first_plugin"
    operators = [MyFirstOperator, MyFirstSensor]
```

You can now place the operator in your DAG:

airflow_home/dags/test_operators.py

```
from datetime import datetime
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators import MyFirstOperator, MyFirstSensor


dag = DAG('my_test_dag', description='Another tutorial DAG',
          schedule_interval='0 12 * * *',
          start_date=datetime(2017, 3, 20), catchup=False)

dummy_task = DummyOperator(task_id='dummy_task', dag=dag)

sensor_task = MyFirstSensor(task_id='my_sensor_task', poke_interval=30, dag=dag)

operator_task = MyFirstOperator(my_operator_param='This is a test.',
                                task_id='my_first_operator_task', dag=dag)

dummy_task >> sensor_task >> operator_task
```

Restart your webserver and scheduler and try out your new workflow.

If you click **View log** of the `my_sensor_task` task, you should see something similar to this:

```
[2017-03-19 14:13:28,719] {base_task_runner.py:95} INFO - Subtask: --------------------------------
[2017-03-19 14:13:28,719] {base_task_runner.py:95} INFO - Subtask: Starting attempt 1 of 1
[2017-03-19 14:13:28,720] {base_task_runner.py:95} INFO - Subtask: --------------------------------
[2017-03-19 14:13:28,720] {base_task_runner.py:95} INFO - Subtask:
[2017-03-19 14:13:28,728] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 14:13:28,728] {model
[2017-03-19 14:13:28,743] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 14:13:28,743] {my_op
[2017-03-19 14:13:58,747] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 14:13:58,747] {my_op
[2017-03-19 14:14:28,750] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 14:14:28,750] {my_op
[2017-03-19 14:14:58,752] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 14:14:58,752] {my_op
[2017-03-19 14:15:28,756] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 14:15:28,756] {my_op
[2017-03-19 14:15:28,757] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 14:15:28,756] {senso
```

Code is in this commit on GitHub.

Communicating between operators with Xcom

In most workflow scenarios downstream tasks will have to use some information from an upstream task. Since each task instance will run in a different process, perhaps on a different machine, Airflow provides a communication mechanism called Xcom for this purpose.

Each task instance can store some information in Xcom using the `xcom_push` function and another task instance can retrieve this information using `xcom_pull`. The information passed using Xcoms will be pickled and stored in the Airflow database (`xcom` table), so it's better to save only small bits of information, rather then large objects.

Let's enhance our Sensor, so that it saves a value to Xcom. We're using the `xcom_push()` function which takes two arguments – a key under which the value will be saved and the value itself.

airflow_home/plugins/my_operators.py

```
class MyFirstSensor(BaseSensorOperator):
    ---

    def poke(self, context):
        ---
        log.info("Current minute (%s) is divisible by 3, sensor finishing.", current_minute)
        task_instance = context['task_instance']
        task_instance.xcom_push('sensors_minute', current_minute)
        return True
```

Now in our operator, which is downstream from the sensor in our DAG, we can use this value, by retrieving it from Xcom. Here we're using the `xcom_pull()` function providing it with two

arguments – the task ID of the task instance which stored the value and the `key` under which the value was stored.

airflow_home/plugins/my_operators.py

```
class MyFirstOperator(BaseOperator):
    ---

    def execute(self, context):
        log.info("Hello World!")
        log.info('operator_param: %s', self.operator_param)
        task_instance = context['task_instance']
        sensors_minute = task_instance.xcom_pull('my_sensor_task', key='sensors_minute')
        log.info('Valid minute as determined by sensor: %s', sensors_minute)
```

Final version of the code is in this commit on GitHub.

If you trigger a DAG run now and look in the operator's logs, you will see that it was able to display the value created by the upstream sensor.

In the docs, you can read more about Airflow XComs.

I hope you found this brief introduction to Airflow useful. Have fun developing your own workflows and data processing pipelines!

Posted by Michał Karzyński • Mar 19th, 2017 • tech

« EuroPython 2016 Presentation

EuroPython 2017 Presentation »

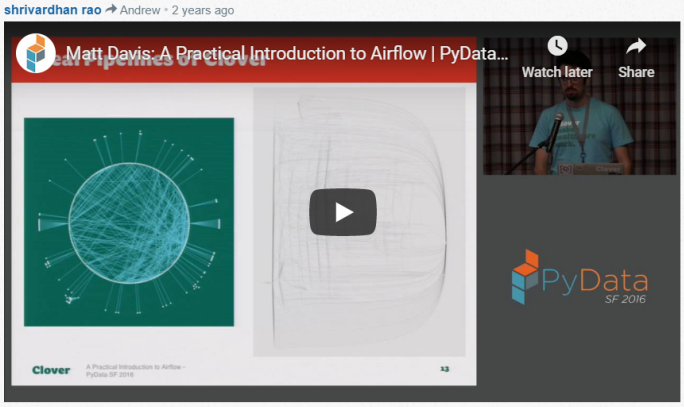
Comments

4 ^ | v 2 • Reply • Share ›

Robert Lugg • a year ago
If you are using Python 3 and get a syntax error with '1L' and snakebite in the error message, then as of 2019-06-4 you should pip uninstall snakebite.
1 ^ | v • Reply • Share ›

Andrew • 2 years ago
This has been very helpful. I'm stuck trying to do some remote debugging with PyCharm and Docker. Have you had any luck? I'm using https://github.com/puckel/d... for my docker compose file. I can get the example operator you provided to run, I just can't get it to pause on break points. Any thoughts? Thanks
1 ^ | v • Reply • Share ›

shrivardhan rao ↗ Andrew • 2 years ago



Check out this video. He explains how you can debug using the sequential executor.
^ | v • Reply • Share ›

Adam Arold • 2 years ago
How do I import this project to PyCharm? My problem is that my `test_operators.py` has compile errors because we try to import `MyFirstOperator` and `MyFirstSensor` from `airflow.operators` but they are in fact in the `plugins` directory. How can I solve this?
1 ^ | v • Reply • Share ›

Geoffrey ↗ Adam Arold • a year ago
The best for PyCharm is to put the pluins folder inside the dags folder.
Then open PyCharm "open project" and use the dags folder as root project dir.
^ | v • Reply • Share ›

Mariusz ↗ Adam Arold • 2 years ago
File | Settings | Project: airflow | Project Structure
remove current content root and create new two:
/home/mariusz/anaconda/envs/airflow/dags
/home/mariusz/anaconda/envs/airflow/plugins
^ | v • Reply • Share ›

Christopher Carlson • 3 years ago
When I try to run the test plugin I get: Broken DAG: [/usr/local/airflow/dags/test_operator.py] cannot import name
MyFirstOperator. Any idea why this is happening?
1 ^ | v 1 • Reply • Share ›

y2k_shubham ↗ Christopher Carlson • 2 years ago • edited
I got this error because I had used space character ' ' in task_id, which isn't supported by Airflow. See this for details: https://stackoverflow.com/q...
17 ^ | v • Reply • Share ›

Ethan Lyon ↗ Christopher Carlson • a year ago
I had to add `from my_operators import MyFirstOperator, MyFirstSensor` to test_operators.py
^ | v • Reply • Share ›

Seetharaman Srinivasan • a year ago
Thank you! It's a useful blog with fully functional examples.
^ | v • Reply • Share ›

vinoth • a year ago
Hi I'm new to apache airflow and I don't have any knowledge in python also.
while running this code

```
virtualenv -p `which python3` venv
```

I am getting this error.

```
The path `which (from --python=`which) does not exist.
```

can any one help me
^ | v • Reply • Share ›

meghana V ↗ vinoth • a year ago
you should mention the python you using instead of which python3.
For python version 2 use virtualenv -p `which python venv
Python 3 virtualenv -p python3 venv

it should work without the error
1 ^ | v • Reply • Share ›

Selvaa S • a year ago

When I try to execute my DAG, I getting error "not able to execute successfully; config file is missing"
Please help.

Reply • Share ›

Greg Walsh • a year ago

Thank you for the great post Michal. I tried setting up the debugging as you mention with PyCharm for a local Airflow instance. I've set up env variables and the method works well for instantiating the DAGs and tasks but when it comes to executing code in custom made plugins the threads do not stop at the breakpoints I set. I don't think it's a caching issue because changes to the execute methods are reflected immediately. Any ideas would be much appreciated!

Reply • Share ›

Dimitri De Franciscis • 2 years ago

FYI, I tried the tutorial using Python 3.7 but it wasn't able to install airflow.
The solution was to install Python 3.6 and create a virtualenv using 3.6

Reply • Share ›

Geoffrey ➚ Dimitri De Franciscis • a year ago

Same for me

Reply • Share ›

Vivian Varun Michael • 2 years ago

Thanks for the really helpfull intro

Reply • Share ›

Chandu Kavar • 2 years ago

Nice Article Michal. It is very helpful. Thanks for writing.

Reply • Share ›

Danielle Felder • 2 years ago

Great article and introduction. Would love to add your review of Airflow to IT Central Station.

Users interested in this solution also read reviews for Control-M. You can find user reviews for this, as well as other comparisons between Airflow and other major workload automation tools here:
https://www.itcentralstatio....

Reply • Share ›

buckets-maestro • 2 years ago

Interesting -- I hadn't seen anyone run Airflow in a virtualenv. When it's on say, a remote server (e.g. an EC2) is there any risk it's not "activated" and thus not able to run it's DAGs? Do you have a fix for that to make sure it's always running? maybe "source activate airflow_env" in the .bashrc file?

Reply • Share ›

20__97 • 2 years ago

[2017-12-16 14:21:30,609] {__init__.py:57} INFO - Using executor SequentialExecutor
[2017-12-16 14:21:30,709] {driver.py:123} INFO - Generating grammar tables from
/usr/lib/python2.7/lib2to3/Grammar.txt
[2017-12-16 14:21:30,741] {driver.py:123} INFO - Generating grammar tables from
/usr/lib/python2.7/lib2to3/PatternGrammar.txt

what does generating grammar tables do ?

Reply • Share ›

John McManus • 2 years ago

Excellent article, just what i was looking for, thank you Michal.

Reply • Share ›

Tripti Singh • 2 years ago

Is there a way for Airflow to make an Http Call to a REST endpoint for invocation of a task, and wait on the completion of the task to go to the next step?

Reply • Share ›

Phil Friedman • 3 years ago • edited

Awesome introduction to Airflow. Any ideas or pointers how to do something like "A PyCharm debug configuration" to debug with Visual Studio Code? I've tried adding an Airflow configuration in launch.json, but have not gotten it to work.

FWIW, this gets you into into the debugger with an airflow test command, but the environment is not quite right:

```
{
"name": "Python: Airflow",
"type": "python",
"request": "launch",
"stopOnEntry": true,
"pythonPath": "${config:python.pythonPath}",
"program": "${workspaceRoot}/.venv/bin/airflow",
"args": [
"test",
"smarty_streets", // DAG
"select_addresses", // task
```

see more

Reply • Share ›

ic • 3 years ago

is it possible to get the count of happened retries in a python method?

thanks!

Reply • Share ›

ic ➚ ic • 3 years ago

try_number through the **kwargs ;)

Reply • Share ›

Manu Zhang • 3 years ago

Nice post but I got "[2017-05-08 08:52:47,324] {jobs.py:534} ERROR - Cannot use more than 1 thread when using sqlite. Setting max_threads to 1" when running "airflow scheduler".

Reply • Share ›

Ryan Stack ➚ Manu Zhang • 3 years ago

`airflow upgradedb` should do the trick

Reply • Share ›

Maxime Beauchemin ➚ Manu Zhang • 3 years ago

Airflow works better on a real database like MySQL or Postgres to manage its state. You can still run some mileage on sqlite out of the box. You can configure Airflow to use the "SequentialExecutor" in your airflow.cfg that won't allow for any parallelism, but works with sqllite and for playing around.

Reply • Share ›

Geoffrey Smith • 3 years ago

'sensor_task_id' should be 'my_sensor_task' in line 8 of the last code snippet (xcom_pull function).

Thank you for the awesome tutorial Michal!

Reply • Share ›

postrational Mod ➚ Geoffrey Smith • 3 years ago

Thanks for pointing this out, I fixed it.

Reply • Share ›

Alexander • 3 years ago

the best article on Airflow that I have found to date.

Reply • Share ›

Boris Tyukin • 3 years ago

great job, Michal! I picked a few tricks - thanks!

Reply • Share ›

David • 3 years ago

Really helpful article. A big thanks it helped me understand better.

Reply • Share ›

Hugh McBride • 3 years ago

Nice article Michal , thx

Reply • Share ›

Suribabu V • 2 years ago

Really helpful. Great article.

1 • Reply • Share ›

Mat • 2 years ago • edited

Hi Michal,

Thank you for the great article! It helped a lot in grasping a lot of these airflow concepts!

I have a question about the use of xcoms in your example, specifically the xcom_pull on 'my_sensor_task' as defined in MyFirstOperator. In a production use case would you want to pass the task id of the sensor task as a parameter to your MyFirstOperator? It doesn't seem intuitive to me to have a static value of a task id within the definition of an operator. Or, are operators designed to be so purpose built that having constants such as these in an operator common practice? I would imagine that you would want to abstract this to make the code more reuse-able.

Thanks!

1 • Reply • Share ›

postrational Mod ➚ Mat • 2 years ago

You're right, hard-coding the ID of the task was just an illustration of how it works. In a production workflow this value would probably be specified in a configuration outside of the operator code and then passed into the operator. Alternatively the operator could search for required information in upstream tasks.

Reply • Share ›

Hu Minghao • 2 years ago • edited

hi Michal,
I created a task to run a runable jar file. But when I start the task, it goes to be error(be red) immidiatly. But the the java process was executing normally on the background. Then after the process ended successfully on the background, the task became green again. Can you give some advise? Thanks.
My java command is "java -Xms512m -Xmx1024m -jar XXX.jar [args...]".

BTW, when use 'airflow test' to run the task, it is ok. So strange!

1 • Reply • Share ›

David • 3 years ago

The Airflow stuff is great, and thanks for that, but you blew my mind with the IPython embed trick...!!!

1 • Reply • Share ›

Manjesh Gowda • 3 years ago

Sad that it cannot be shared on LinkedIn

1 • Reply • Share ›

Manjesh Gowda • 3 years ago

You are a rock star

1 • Reply • Share ›

FKO • 3 years ago

Awesome article, Thanks!

1 • Reply • Share ›

Rui Lopes • 3 years ago

Great article Michal. I've just started implementing data pipelines in Airflow and you article was super helpful.