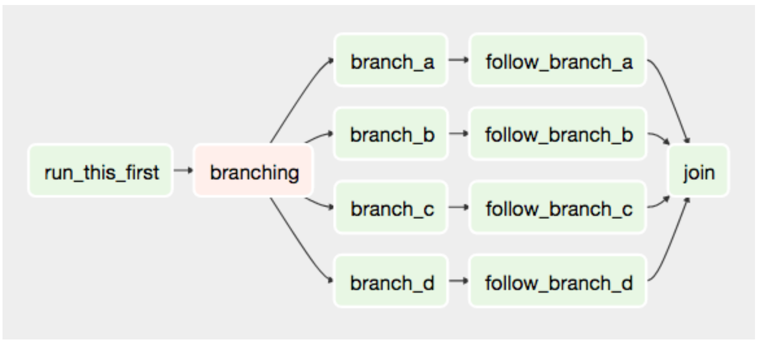
Get started developing workflows with Apache Airflow

Mar 19th, 2017

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`

When?	DAG	Task	Info about other tasks
Base class	`DAG`	`BaseOperator`	

Airflow documentation provides more information about these and other concepts.

Prerequisites

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.

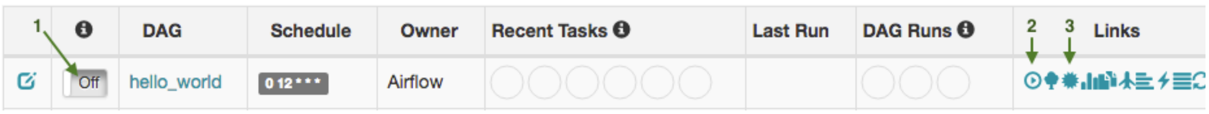
Running your DAG

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`.

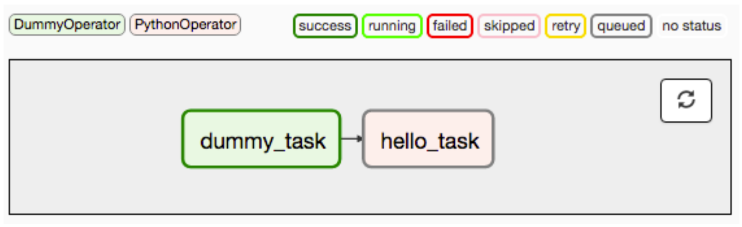
When you reload the Airflow UI in your browser, you should see your `hello_world` DAG listed in Airflow UI.



	ℹ	DAG	Schedule	Owner	Recent Tasks ℹ	Last Run	DAG Runs ℹ	Links
	Off	hello_world	0 12 * * *	Airflow	○○○○○○		○○○	

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] {models.py:1342} INFO - Ex
[2017-03-19 13:49:58,818] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 13:49:58,818] {python_operator.py:81} IN
```

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

```
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

```
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:

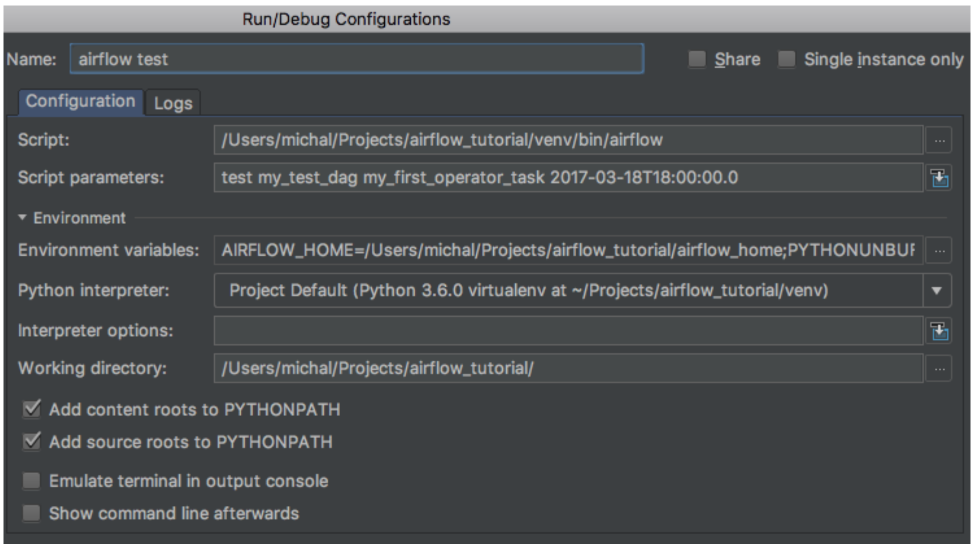
```
(venv) $ airflow test my_test_dag my_first_operator_task 2017-03-18T18:00:00.0
```

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/python3.6/site-packages/a
 'dag': <DAG: my_test_dag>,
 'dag_run': None,
 '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/site-packages/airfl
 '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 [running]>,
 '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.



Name: airflow test

Script: /Users/michal/Projects/airflow_tutorial/venv/bin/airflow

Script parameters: test my_test_dag my_first_operator_task 2017-03-18T18:00:00.0

Environment variables: AIRFLOW_HOME=/Users/michal/Projects/airflow_tutorial/airflow_home;PYTHONUNBUF

Python interpreter: Project Default (Python 3.6.0 virtualenv at ~/Projects/airflow_tutorial/venv)

Interpreter options:

Working directory: /Users/michal/Projects/airflow_tutorial/

☑ Add content roots to PYTHONPATH

☑ Add source roots to PYTHONPATH

☐ Emulate terminal in output console

☐ Show command line afterwards

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

```python
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_minute)
            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

```python
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

```python
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] {models.py:1342} INFO - Ex
[2017-03-19 14:13:28,743] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 14:13:28,743] {my_operators.py:34} INFO
[2017-03-19 14:13:58,747] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 14:13:58,747] {my_operators.py:34} INFO
[2017-03-19 14:14:28,750] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 14:14:28,750] {my_operators.py:34} INFO
[2017-03-19 14:14:58,752] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 14:14:58,752] {my_operators.py:34} INFO
[2017-03-19 14:15:28,756] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 14:15:28,756] {my_operators.py:37} INFO
[2017-03-19 14:15:28,757] {base_task_runner.py:95} INFO - Subtask: [2017-03-19 14:15:28,756] {sensors.py:83} INFO - Suc
```

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

Gratipay Tweet G+ Like Share One person likes this. Sign Up to see what your friends like.

« EuroPython 2016 Presentation EuroPython 2017 Presentation »

Comments

21 Comments Michał Karzyński's Blog 1 Login

Recommend 6 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Dirk • 7 days ago

Awesome post indeed! Is there any way you can do another post on deployment of Airflow on Google Cloud Platform & CloudSQL. Resources are a bit scarce and I actually like how you go into details with this topic :) Thanks !

Reply • Share ›

ic • 14 days ago
is it possible to get the count of happened retries in a python method?

thanks!
• Reply • Share ›

ic ➜ ic • 13 days ago
try_number through the **kwargs ;)
• Reply • Share ›

Rafael Gomes • a month ago
@postrational, thanks again for your sharing this.
I am sorry to bother you again, but I have been stuck in a problem for a while now.
I am working in a project that my first sensor task need to check if marker file exist and my last sensor task need to check if a marker file was modified and trigger the dag again.
Since these files are dummy files I cannot do a checksum or something like that, so I am checking the modification time.

My problem is that my xcom_pull are always returning None, even though I can see on the UI that the xcom are there.

I have created a ssh sensor that use xcom to push the filename_YYYYMMDD as key and modification time as value. Also, I have tried to different ways to use xcom_pull:
1) Using the task id of the current task instance;
2) Passing the task id as an argument;

Here is the code of the sensor's poke method:

def poke(self, context):
logging.info('Poking for %s', self.path)
try:
task_id = context['task'].task_id

see more

• Reply • Share ›

Jin Kang • a month ago
thanks a lot! really helpful stuff!
• Reply • Share ›

Rafael Gomes • a month ago
@postrational , great post about Airflow.
I am trying to do "Alternatively, you can also use an airflow test based run configuration to set breakpoints in IDEs such as PyCharm." but I could not succeed. Could you please refer some tutorial or material that could help me with that?
• Reply • Share ›

postrational Mod ➜ Rafael Gomes • a month ago
I added a link and a screenshot to that section of the article. Let me know if that helps.
• Reply • Share ›

Rafael Gomes ➜ postrational • a month ago
Thank you very much. It helped a lot, you are doing a great job with this material. I hope to read more of your articles about airflow.
• Reply • Share ›

Anjana A • a month ago
Best article on Airflow! I have been struggling for 2 days with scheduling and settingup an Airflow DAG and this aticle helped me succeed. Thanks!
• Reply • Share ›

Maxime Beauchemin • 5 months ago
@sdotsen DAG typically run on a schedule (though they can be on-demand) as well. You sensor task, say may be waiting for `customer_source_2018_01_01.tar.gz` where somehow `2018_01_01` is parameterized and related to the current run. The sensor pokes at an specified interval and exist with success when the file has landed. Then the assumption it that you'd have a dowstream task that depend on the

specified interval and exist with success when the file has landed. Then the assumption it that you'd have a downstream task that depend on the sensor that will process that file and say load it in a database.

There may be multiple DAG runs (different schedules) running at the same time. Also the assumption is that the scheduler constantly triggers the tasks who's dependencies have been met.

Reply • Share ›

sdotsen • 5 months ago

I'm trying to wrap my head around "sensors" and I'm missing something very basic it seems. So let's say I'm waiting for a file to show up in a directory. I have code to sense for the file and it will execute a command. It then exits and doesn't run again. Maybe I'm using "sensors" the wrong way, but I thought it just runs forever.

If the job exits once it meets its criteria, how does the dag run again? Do i also need to set up the dag to run every X minute?

Reply • Share ›

Christopher Carlson • 5 months 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?

Reply • Share ›

Manu Zhang • 6 months 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 • 2 months ago
>
> `airflow upgradedb` should do the trick
>
> Reply • Share ›

> **Maxime Beauchemin** → Manu Zhang • 5 months 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 • 6 months 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 • 6 months ago
>
> Thanks for pointing this out, I fixed it.
>
> Reply • Share ›

Alexander • 6 months ago

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

Reply • Share ›

Boris Tyukin • 7 months ago

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

Reply • Share ›

David • 7 months ago

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

Reply • Share ›

Hugh McBride • 7 months ago

Nice article Michał , thx

Reply • Share ›

ALSO ON MICHAŁ KARZYŃSKI'S BLOG

Packaging Django applications as Docker container images

31 comments • 3 years ago

Smittles — Is it possible to debug Django with pdb from the host machine?

Setting up a continuous integration server for Django with Jenkins

10 comments • 4 years ago

Carlos Killpack — I wonder why you would assume people are using SVN for version control. I realize there are still valid use cases for the technology, but in general I …

Serving multiple Django applications with Nginx and Gunicorn

25 comments • 4 years ago

powerpbx — Nice write up. Very hard to find all this information in one place like this. Also a lot of these best practices are fairly recent so there is a lot of …

Webmin Administrator's Cookbook

1 comment • 4 years ago

Nick — Help with Webmin is really sketchy so I don't know how I would have configured my server without the snippets of your book I've seen online.

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.

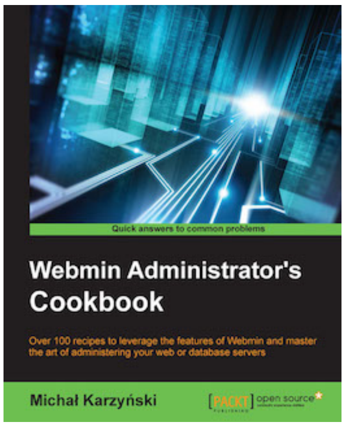
My favorite languages are currently JavaScript and Python and I'm good with Django, Angular, ExtJS and other MVC frameworks.

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

Recent Posts

- EuroPython 2017 Presentation
- Get started developing workflows with Apache Airflow
- EuroPython 2016 Presentation
- Building beautiful REST APIs using Flask, Swagger UI and Flask-RESTPlus
- Packaging Django applications into Docker container images

Grab my book


Quick answers to common problems
Webmin Administrator's Cookbook
Over 100 recipes to leverage the features of Webmin and master the art of administering your web or database servers
Michał Karzyński
[PACKT] open source

GitHub Repos

- blog

  Source code repository for my blog.

- rest_api_demo

  Boilerplate code for a RESTful API based on Flask-RESTPlus

- airflow_tutorial
- 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

Tweets by @postrational

Michał Karzyński Retweeted

Randy Olson
@randal_olson

The #Python Graph Gallery: Useful for discovering and learning how to code #dataviz in Python.python-graph-gallery.com

DISTRIBUTION

Embed View on Twitter