Airflow Documentation

Apache Airflow

Jul 18, 2018

Contents

Important: Disclaimer: Apache Airflow is an effort undergoing incubation at The Apache Software Foundation (ASF), sponsored by the Apache Incubator. Incubation is required of all newly accepted projects until a further review indicates that the infrastructure, communications, and decision making process have stabilized in a manner consistent with other successful ASF projects. While incubation status is not necessarily a reflection of the completeness or stability of the code, it does indicate that the project has yet to be fully endorsed by the ASF.

Airflow is a platform to programmatically author, schedule and monitor workflows.

Use airflow to author workflows as directed acyclic graphs (DAGs) of tasks. The airflow scheduler executes your tasks on an array of workers while following the specified dependencies. Rich command line utilities make performing complex surgeries on DAGs a snap. The rich user interface makes it easy to visualize pipelines running in production, monitor progress, and troubleshoot issues when needed.

When workflows are defined as code, they become more maintainable, versionable, testable, and collaborative.

CHAPTER 1

Principles

- **Dynamic**: Airflow pipelines are configuration as code (Python), allowing for dynamic pipeline generation. This allows for writing code that instantiates pipelines dynamically.
- **Extensible**: Easily define your own operators, executors and extend the library so that it fits the level of abstraction that suits your environment.
- **Elegant**: Airflow pipelines are lean and explicit. Parameterizing your scripts is built into the core of Airflow using the powerful **Jinja** templating engine.
- **Scalable**: Airflow has a modular architecture and uses a message queue to orchestrate an arbitrary number of workers. Airflow is ready to scale to infinity.

CHAPTER 2

Beyond the Horizon

Airflow **is not** a data streaming solution. Tasks do not move data from one to the other (though tasks can exchange metadata!). Airflow is not in the Spark Streaming or Storm space, it is more comparable to Oozie or Azkaban.

Workflows are expected to be mostly static or slowly changing. You can think of the structure of the tasks in your workflow as slightly more dynamic than a database structure would be. Airflow workflows are expected to look similar from a run to the next, this allows for clarity around unit of work and continuity.

CHAPTER 3

Content

3.1 Project

3.1.1 History

Airflow was started in October 2014 by Maxime Beauchemin at Airbnb. It was open source from the very first commit and officially brought under the Airbnb Github and announced in June 2015.

The project joined the Apache Software Foundation's incubation program in March 2016.

3.1.2 Committers

- @mistercrunch (Maxime "Max" Beauchemin)
- @r39132 (Siddharth "Sid" Anand)
- @criccomini (Chris Riccomini)
- @bolkedebruin (Bolke de Bruin)
- @artwr (Arthur Wiedmer)
- @jlowin (Jeremiah Lowin)
- @patrickleotardif (Patrick Leo Tardif)
- @aoen (Dan Davydov)
- @syvineckruyk (Steven Yvinec-Kruyk)
- @msumit (Sumit Maheshwari)
- @alexvanboxel (Alex Van Boxel)
- @saguziel (Alex Guziel)
- @joygao (Joy Gao)

- @fokko (Fokko Driesprong)
- @ash (Ash Berlin-Taylor)
- @kaxilnaik (Kaxil Naik)

For the full list of contributors, take a look at Airflow's Github Contributor page:

3.1.3 Resources & links

- Airflow's official documentation
- Mailing list (send emails to `dev-subscribe@airflow.incubator.apache.org` and/or `commits-subscribe@airflow.incubator.apache.org` to subscribe to each)
- Issues on Apache's Jira
- Gitter (chat) Channel
- More resources and links to Airflow related content on the Wiki

3.1.4 Roadmap

Please refer to the Roadmap on the wiki

3.2 License

```
Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.
```

(continues on next page)

(continued from previous page)

```
   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
```

(continues on next page)

(continued from previous page)

```
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
```

(continues on next page)

(continued from previous page)

```
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright 2015 Apache Software Foundation

Licensed to the Apache Software Foundation (ASF) under one
or more contributor license agreements.  See the NOTICE file
```

(continues on next page)

(continued from previous page)

```
distributed with this work for additional information
regarding copyright ownership.  The ASF licenses this file
to you under the Apache License, Version 2.0 (the
"License"); you may not use this file except in compliance
with the License.  You may obtain a copy of the License at

  http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing,
software distributed under the License is distributed on an
"AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
KIND, either express or implied.  See the License for the
specific language governing permissions and limitations
under the License.
Status API Training Shop Blog About
© 2016 GitHub, Inc. Terms Privacy Security Contact Help
```

3.3 Quick Start

The installation is quick and straightforward.

```
# airflow needs a home, ~/airflow is the default,
# but you can lay foundation somewhere else if you prefer
# (optional)
export AIRFLOW_HOME=~/airflow

# install from pypi using pip
pip install apache-airflow

# initialize the database
airflow initdb

# start the web server, default port is 8080
airflow webserver -p 8080

# start the scheduler
airflow scheduler

# visit localhost:8080 in the browser and enable the example dag in the home page
```

Upon running these commands, Airflow will create the `$AIRFLOW_HOME` folder and lay an "airflow.cfg" file with defaults that get you going fast. You can inspect the file either in `$AIRFLOW_HOME/airflow.cfg`, or through the UI in the `Admin->Configuration` menu. The PID file for the webserver will be stored in `$AIRFLOW_HOME/airflow-webserver.pid` or in `/run/airflow/webserver.pid` if started by systemd.

Out of the box, Airflow uses a sqlite database, which you should outgrow fairly quickly since no parallelization is possible using this database backend. It works in conjunction with the `SequentialExecutor` which will only run task instances sequentially. While this is very limiting, it allows you to get up and running quickly and take a tour of the UI and the command line utilities.

Here are a few commands that will trigger a few task instances. You should be able to see the status of the jobs change in the `example1` DAG as you run the commands below.

```
# run your first task instance
airflow run example_bash_operator runme_0 2015-01-01
# run a backfill over 2 days
airflow backfill example_bash_operator -s 2015-01-01 -e 2015-01-02
```

3.3.1 What's Next?

From this point, you can head to the *Tutorial* section for further examples or the *How-to Guides* section if you're ready to get your hands dirty.

3.4 Installation

3.4.1 Getting Airflow

The easiest way to install the latest stable version of Airflow is with `pip`:

```
pip install apache-airflow
```

You can also install Airflow with support for extra features like `s3` or `postgres`:

```
pip install "apache-airflow[s3, postgres]"
```

Note: GPL dependency

One of the dependencies of Apache Airflow by default pulls in a GPL library ('unidecode'). In case this is a concern you can force a non GPL library by issuing `export SLUGIFY_USES_TEXT_UNIDECODE=yes` and then proceed with the normal installation. Please note that this needs to be specified at every upgrade. Also note that if *unidecode* is already present on the system the dependency will still be used.

3.4.2 Extra Packages

The `apache-airflow` PyPI basic package only installs what's needed to get started. Subpackages can be installed depending on what will be useful in your environment. For instance, if you don't need connectivity with Postgres, you won't have to go through the trouble of installing the `postgres-devel` yum package, or whatever equivalent applies on the distribution you are using.

Behind the scenes, Airflow does conditional imports of operators that require these extra dependencies.

Here's the list of the subpackages and what they enable:

sub-package	install command	enables
all	`pip install apache-airflow[all]`	All Airflow features known to man
all_dbs	`pip install apache-airflow[all_dbs]`	All databases integrations
async	`pip install apache-airflow[async]`	Async worker classes for gunicorn
devel	`pip install apache-airflow[devel]`	Minimum dev tools requirements
devel_hadoop	`pip install apache-airflow[devel_hadoop]`	Airflow + dependencies on the Hadoop stack
celery	`pip install apache-airflow[celery]`	CeleryExecutor
crypto	`pip install apache-airflow[crypto]`	Encrypt connection passwords in metadata db
druid	`pip install apache-airflow[druid]`	Druid.io related operators & hooks
gcp_api	`pip install apache-airflow[gcp_api]`	Google Cloud Platform hooks and operators (using `google-api-python-client`)
jdbc	`pip install apache-airflow[jdbc]`	JDBC hooks and operators
hdfs	`pip install apache-airflow[hdfs]`	HDFS hooks and operators
hive	`pip install apache-airflow[hive]`	All Hive related operators
kerberos	`pip install apache-airflow[kerberos]`	kerberos integration for kerberized hadoop
ldap	`pip install apache-airflow[ldap]`	ldap authentication for users
mssql	`pip install apache-airflow[mssql]`	Microsoft SQL operators and hook, support as an Airflow backend
mysql	`pip install apache-airflow[mysql]`	MySQL operators and hook, support as an Airflow backend. The version of MySQL server has to be 5.6.4+. The exact version upper bound depends on version of `mysqlclient` package. For example, `mysqlclient` 1.3.12 can only be used with MySQL server 5.6.4 through 5.7.
password	`pip install apache-airflow[password]`	Password Authentication for users
postgres	`pip install apache-airflow[postgres]`	Postgres operators and hook, support as an Airflow backend
qds	`pip install apache-airflow[qds]`	Enable QDS (qubole data services) support
rabbitmq	`pip install apache-airflow[rabbitmq]`	Rabbitmq support as a Celery backend
s3	`pip install apache-airflow[s3]`	`S3KeySensor`, `S3PrefixSensor`
samba	`pip install apache-airflow[samba]`	`Hive2SambaOperator`
slack	`pip install apache-airflow[slack]`	`SlackAPIPostOperator`
vertica	`pip install apache-airflow[vertica]`	Vertica hook support as an Airflow backend
cloudant	`pip install apache-airflow[cloudant]`	Cloudant hook
redis	`pip install apache-airflow[redis]`	Redis hooks and sensors

3.4.3 Initiating Airflow Database

Airflow requires a database to be initiated before you can run tasks. If you're just experimenting and learning Airflow, you can stick with the default SQLite option. If you don't want to use SQLite, then take a look at *Initializing a Database Backend* to setup a different database.

After configuration, you'll need to initialize the database before you can run tasks:

```
airflow initdb
```

3.5 Tutorial

This tutorial walks you through some of the fundamental Airflow concepts, objects, and their usage while writing your first pipeline.

3.5.1 Example Pipeline definition

Here is an example of a basic pipeline definition. Do not worry if this looks complicated, a line by line explanation follows below.

```
"""
Code that goes along with the Airflow tutorial located at:
https://github.com/apache/incubator-airflow/blob/master/airflow/example_dags/tutorial.
↪py
"""
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta


default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 6, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
}

dag = DAG('tutorial', default_args=default_args)

# t1, t2 and t3 are examples of tasks created by instantiating operators
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag)

t2 = BashOperator(
```

(continues on next page)

(continued from previous page)

```
    task_id='sleep',
    bash_command='sleep 5',
    retries=3,
    dag=dag)

templated_command = """
    {% for i in range(5) %}
        echo "{{ ds }}"
        echo "{{ macros.ds_add(ds, 7)}}"
        echo "{{ params.my_param }}"
    {% endfor %}
"""

t3 = BashOperator(
    task_id='templated',
    bash_command=templated_command,
    params={'my_param': 'Parameter I passed in'},
    dag=dag)

t2.set_upstream(t1)
t3.set_upstream(t1)
```

3.5.2 It's a DAG definition file

One thing to wrap your head around (it may not be very intuitive for everyone at first) is that this Airflow Python script is really just a configuration file specifying the DAG's structure as code. The actual tasks defined here will run in a different context from the context of this script. Different tasks run on different workers at different points in time, which means that this script cannot be used to cross communicate between tasks. Note that for this purpose we have a more advanced feature called `XCom`.

People sometimes think of the DAG definition file as a place where they can do some actual data processing - that is not the case at all! The script's purpose is to define a DAG object. It needs to evaluate quickly (seconds, not minutes) since the scheduler will execute it periodically to reflect the changes if any.

3.5.3 Importing Modules

An Airflow pipeline is just a Python script that happens to define an Airflow DAG object. Let's start by importing the libraries we will need.

```
# The DAG object; we'll need this to instantiate a DAG
from airflow import DAG

# Operators; we need this to operate!
from airflow.operators.bash_operator import BashOperator
```

3.5.4 Default Arguments

We're about to create a DAG and some tasks, and we have the choice to explicitly pass a set of arguments to each task's constructor (which would become redundant), or (better!) we can define a dictionary of default parameters that we can use when creating tasks.

```
from datetime import datetime, timedelta

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 6, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
}
```

For more information about the BaseOperator's parameters and what they do, refer to the *airflow.models.BaseOperator* documentation.

Also, note that you could easily define different sets of arguments that would serve different purposes. An example of that would be to have different settings between a production and development environment.

3.5.5 Instantiate a DAG

We'll need a DAG object to nest our tasks into. Here we pass a string that defines the `dag_id`, which serves as a unique identifier for your DAG. We also pass the default argument dictionary that we just defined and define a `schedule_interval` of 1 day for the DAG.

```
dag = DAG(
    'tutorial', default_args=default_args, schedule_interval=timedelta(1))
```

3.5.6 Tasks

Tasks are generated when instantiating operator objects. An object instantiated from an operator is called a constructor. The first argument `task_id` acts as a unique identifier for the task.

```
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag)

t2 = BashOperator(
    task_id='sleep',
    bash_command='sleep 5',
    retries=3,
    dag=dag)
```

Notice how we pass a mix of operator specific arguments (`bash_command`) and an argument common to all operators (`retries`) inherited from BaseOperator to the operator's constructor. This is simpler than passing every argument for every constructor call. Also, notice that in the second task we override the `retries` parameter with `3`.

The precedence rules for a task are as follows:

1. Explicitly passed arguments

2. Values that exist in the `default_args` dictionary
3. The operator's default value, if one exists

A task must include or inherit the arguments `task_id` and `owner`, otherwise Airflow will raise an exception.

3.5.7 Templating with Jinja

Airflow leverages the power of Jinja Templating and provides the pipeline author with a set of built-in parameters and macros. Airflow also provides hooks for the pipeline author to define their own parameters, macros and templates.

This tutorial barely scratches the surface of what you can do with templating in Airflow, but the goal of this section is to let you know this feature exists, get you familiar with double curly brackets, and point to the most common template variable: `{{ ds }}` (today's "date stamp").

```
templated_command = """
    {% for i in range(5) %}
        echo "{{ ds }}"
        echo "{{ macros.ds_add(ds, 7) }}"
        echo "{{ params.my_param }}"
    {% endfor %}
"""

t3 = BashOperator(
    task_id='templated',
    bash_command=templated_command,
    params={'my_param': 'Parameter I passed in'},
    dag=dag)
```

Notice that the `templated_command` contains code logic in `{% %}` blocks, references parameters like `{{ ds }}`, calls a function as in `{{ macros.ds_add(ds, 7) }}`, and references a user-defined parameter in `{{ params.my_param }}`.

The `params` hook in `BaseOperator` allows you to pass a dictionary of parameters and/or objects to your templates. Please take the time to understand how the parameter `my_param` makes it through to the template.

Files can also be passed to the `bash_command` argument, like `bash_command='templated_command.sh'`, where the file location is relative to the directory containing the pipeline file (`tutorial.py` in this case). This may be desirable for many reasons, like separating your script's logic and pipeline code, allowing for proper code highlighting in files composed in different languages, and general flexibility in structuring pipelines. It is also possible to define your `template_searchpath` as pointing to any folder locations in the DAG constructor call.

Using that same DAG constructor call, it is possible to define `user_defined_macros` which allow you to specify your own variables. For example, passing `dict(foo='bar')` to this argument allows you to use `{{ foo }}` in your templates. Moreover, specifying `user_defined_filters` allow you to register you own filters. For example, passing `dict(hello=lambda name: 'Hello %s' % name)` to this argument allows you to use `{{ 'world' | hello }}` in your templates. For more information regarding custom filters have a look at the Jinja Documentation

For more information on the variables and macros that can be referenced in templates, make sure to read through the *Macros* section

3.5.8 Setting up Dependencies

We have two simple tasks that do not depend on each other. Here's a few ways you can define dependencies between them:

```
t2.set_upstream(t1)

# This means that t2 will depend on t1
# running successfully to run
# It is equivalent to
# t1.set_downstream(t2)

t3.set_upstream(t1)

# all of this is equivalent to
# dag.set_dependency('print_date', 'sleep')
# dag.set_dependency('print_date', 'templated')
```

Note that when executing your script, Airflow will raise exceptions when it finds cycles in your DAG or when a dependency is referenced more than once.

3.5.9 Recap

Alright, so we have a pretty basic DAG. At this point your code should look something like this:

```
"""
Code that goes along with the Airflow located at:
http://airflow.readthedocs.org/en/latest/tutorial.html
"""
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta


default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 6, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
}

dag = DAG(
    'tutorial', default_args=default_args, schedule_interval=timedelta(1))

# t1, t2 and t3 are examples of tasks created by instantiating operators
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag)

t2 = BashOperator(
    task_id='sleep',
    bash_command='sleep 5',
```

(continues on next page)

(continued from previous page)

```
    retries=3,
    dag=dag)

templated_command = """
    {% for i in range(5) %}
        echo "{{ ds }}"
        echo "{{ macros.ds_add(ds, 7)}}"
        echo "{{ params.my_param }}"
    {% endfor %}
"""

t3 = BashOperator(
    task_id='templated',
    bash_command=templated_command,
    params={'my_param': 'Parameter I passed in'},
    dag=dag)

t2.set_upstream(t1)
t3.set_upstream(t1)
```

3.5.10 Testing

3.5.10.1 Running the Script

Time to run some tests. First let's make sure that the pipeline parses. Let's assume we're saving the code from the previous step in `tutorial.py` in the DAGs folder referenced in your `airflow.cfg`. The default location for your DAGs is `~/airflow/dags`.

```
python ~/airflow/dags/tutorial.py
```

If the script does not raise an exception it means that you haven't done anything horribly wrong, and that your Airflow environment is somewhat sound.

3.5.10.2 Command Line Metadata Validation

Let's run a few commands to validate this script further.

```
# print the list of active DAGs
airflow list_dags

# prints the list of tasks the "tutorial" dag_id
airflow list_tasks tutorial

# prints the hierarchy of tasks in the tutorial DAG
airflow list_tasks tutorial --tree
```

3.5.10.3 Testing

Let's test by running the actual task instances on a specific date. The date specified in this context is an `execution_date`, which simulates the scheduler running your task or dag at a specific date + time:

```
# command layout: command subcommand dag_id task_id date

# testing print_date
airflow test tutorial print_date 2015-06-01

# testing sleep
airflow test tutorial sleep 2015-06-01
```

Now remember what we did with templating earlier? See how this template gets rendered and executed by running this command:

```
# testing templated
airflow test tutorial templated 2015-06-01
```

This should result in displaying a verbose log of events and ultimately running your bash command and printing the result.

Note that the `airflow test` command runs task instances locally, outputs their log to stdout (on screen), doesn't bother with dependencies, and doesn't communicate state (running, success, failed, ...) to the database. It simply allows testing a single task instance.

3.5.10.4 Backfill

Everything looks like it's running fine so let's run a backfill. `backfill` will respect your dependencies, emit logs into files and talk to the database to record status. If you do have a webserver up, you'll be able to track the progress. `airflow webserver` will start a web server if you are interested in tracking the progress visually as your backfill progresses.

Note that if you use `depends_on_past=True`, individual task instances will depend on the success of the preceding task instance, except for the start_date specified itself, for which this dependency is disregarded.

The date range in this context is a `start_date` and optionally an `end_date`, which are used to populate the run schedule with task instances from this dag.

```
# optional, start a web server in debug mode in the background
# airflow webserver --debug &

# start your backfill on a date range
airflow backfill tutorial -s 2015-06-01 -e 2015-06-07
```

3.5.11 What's Next?

That's it, you've written, tested and backfilled your very first Airflow pipeline. Merging your code into a code repository that has a master scheduler running against it should get it to get triggered and run every day.

Here's a few things you might want to do next:

- Take an in-depth tour of the UI - click all the things!
- Keep reading the docs! Especially the sections on:
  - Command line interface
  - Operators
  - Macros
- Write your first pipeline!

3.6 How-to Guides

Setting up the sandbox in the *Quick Start* section was easy; building a production-grade environment requires a bit more work!

These how-to guides will step you through common tasks in using and configuring an Airflow environment.

3.6.1 Setting Configuration Options

The first time you run Airflow, it will create a file called `airflow.cfg` in your `$AIRFLOW_HOME` directory (`~/airflow` by default). This file contains Airflow's configuration and you can edit it to change any of the settings. You can also set options with environment variables by using this format: `$AIRFLOW__{SECTION}__{KEY}` (note the double underscores).

For example, the metadata database connection string can either be set in `airflow.cfg` like this:

```
[core]
sql_alchemy_conn = my_conn_string
```

or by creating a corresponding environment variable:

```
AIRFLOW__CORE__SQL_ALCHEMY_CONN=my_conn_string
```

You can also derive the connection string at run time by appending `_cmd` to the key like this:

```
[core]
sql_alchemy_conn_cmd = bash_command_to_run
```

-But only three such configuration elements namely sql_alchemy_conn, broker_url and result_backend can be fetched as a command. The idea behind this is to not store passwords on boxes in plain text files. The order of precedence is as follows -

1. environment variable
2. configuration in airflow.cfg
3. command in airflow.cfg
4. default

3.6.2 Initializing a Database Backend

If you want to take a real test drive of Airflow, you should consider setting up a real database backend and switching to the LocalExecutor.

As Airflow was built to interact with its metadata using the great SqlAlchemy library, you should be able to use any database backend supported as a SqlAlchemy backend. We recommend using **MySQL** or **Postgres**.

Note: We rely on more strict ANSI SQL settings for MySQL in order to have sane defaults. Make sure to have specified *explicit_defaults_for_timestamp=1* in your my.cnf under *[mysqld]*

Note: If you decide to use **Postgres**, we recommend using the `psycopg2` driver and specifying it in your SqlAlchemy connection string. Also note that since SqlAlchemy does not expose a way to target a specific schema in

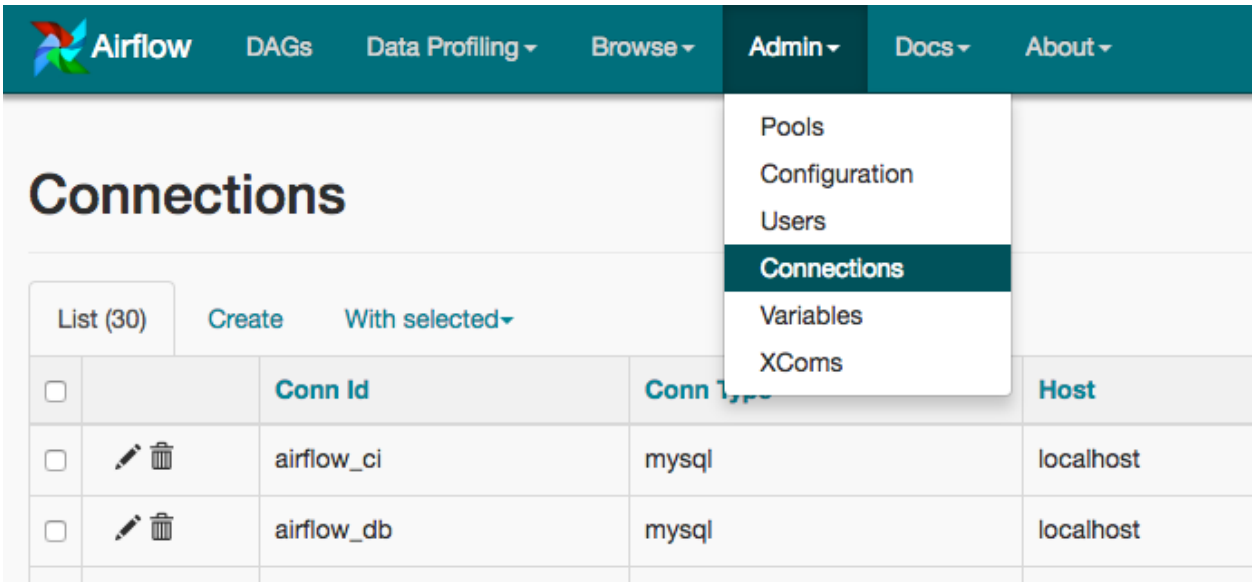
the Postgres connection URI, you may want to set a default schema for your role with a command similar to `ALTER ROLE username SET search_path = airflow, foobar;`

Once you've setup your database to host Airflow, you'll need to alter the SqlAlchemy connection string located in your configuration file `$AIRFLOW_HOME/airflow.cfg`. You should then also change the "executor" setting to use "LocalExecutor", an executor that can parallelize task instances locally.

```
# initialize the database
airflow initdb
```

3.6.3 Managing Connections

Airflow needs to know how to connect to your environment. Information such as hostname, port, login and passwords to other systems and services is handled in the `Admin->Connection` section of the UI. The pipeline code you will author will reference the 'conn_id' of the Connection objects.

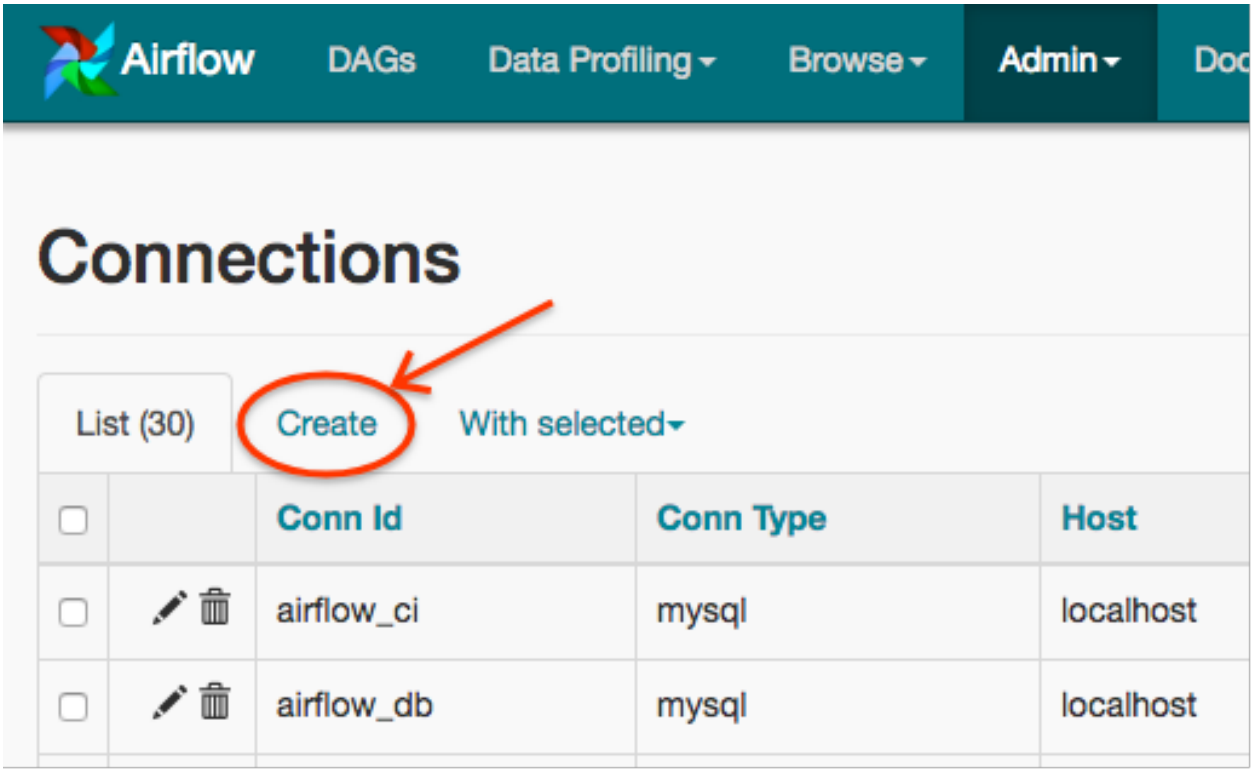


Connections can be created and managed using either the UI or environment variables.

See the *Connenctions Concepts* documentation for more information.

3.6.3.1 Creating a Connection with the UI

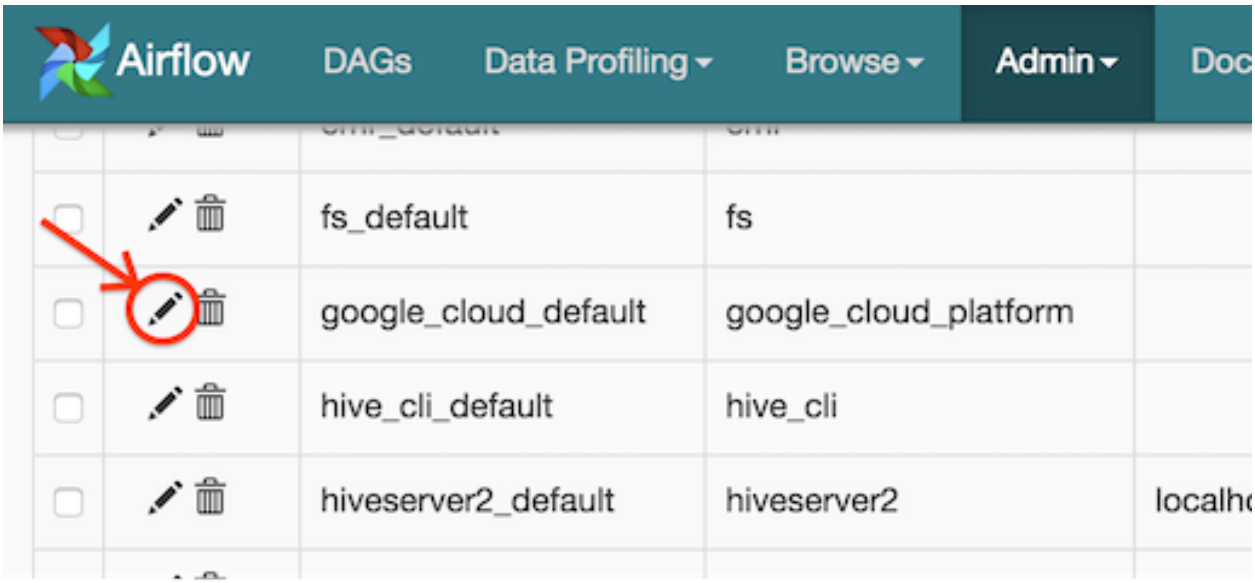
Open the `Admin->Connection` section of the UI. Click the `Create` link to create a new connection.



1. Fill in the `Conn Id` field with the desired connection ID. It is recommended that you use lower-case characters and separate words with underscores.
2. Choose the connection type with the `Conn Type` field.
3. Fill in the remaining fields. See *Connection Types* for a description of the fields belonging to the different connection types.
4. Click the `Save` button to create the connection.

3.6.3.2 Editing a Connection with the UI

Open the `Admin->Connection` section of the UI. Click the pencil icon next to the connection you wish to edit in the connection list.



Modify the connection properties and click the `Save` button to save your changes.

3.6.3.3 Creating a Connection with Environment Variables

Connections in Airflow pipelines can be created using environment variables. The environment variable needs to have a prefix of `AIRFLOW_CONN_` for Airflow with the value in a URI format to use the connection properly.

When referencing the connection in the Airflow pipeline, the `conn_id` should be the name of the variable without the prefix. For example, if the `conn_id` is named `postgres_master` the environment variable should be named `AIRFLOW_CONN_POSTGRES_MASTER` (note that the environment variable must be all uppercase). Airflow assumes the value returned from the environment variable to be in a URI format (e.g. `postgres://user:password@localhost:5432/master` or `s3://accesskey:secretkey@S3`).

3.6.3.4 Connection Types

Google Cloud Platform

The Google Cloud Platform connection type enables the *GCP Integrations*.

Authenticating to GCP

There are two ways to connect to GCP using Airflow.

1. Use Application Default Credentials, such as via the metadata server when running on Google Compute Engine.
2. Use a service account key file (JSON format) on disk.

Default Connection IDs

The following connection IDs are used by default.

`bigquery_default` Used by the *`BigQueryHook`* hook.

`google_cloud_datastore_default` Used by the *`DatastoreHook`* hook.

google_cloud_default Used by the *GoogleCloudBaseHook*, *DataFlowHook*, *DataProcHook*, *MLEngineHook*, and *GoogleCloudStorageHook* hooks.

Configuring the Connection

Project Id (required) The Google Cloud project ID to connect to.

Keyfile Path Path to a service account key file (JSON format) on disk.

Not required if using application default credentials.

Keyfile JSON Contents of a service account key file (JSON format) on disk. It is recommended to *Secure your connections* if using this method to authenticate.

Not required if using application default credentials.

Scopes (comma separated) A list of comma-separated Google Cloud scopes to authenticate with.

Note: Scopes are ignored when using application default credentials. See issue AIRFLOW-2522.

3.6.4 Securing Connections

By default, Airflow will save the passwords for the connection in plain text within the metadata database. The `crypto` package is highly recommended during installation. The `crypto` package does require that your operating system have libffi-dev installed.

If `crypto` package was not installed initially, you can still enable encryption for connections by following steps below:

1. Install crypto package `pip install apache-airflow[crypto]`
2. Generate fernet_key, using this code snippet below. fernet_key must be a base64-encoded 32-byte key.

```
from cryptography.fernet import Fernet
fernet_key= Fernet.generate_key()
print(fernet_key.decode()) # your fernet_key, keep it in secured place!
```

3. Replace `airflow.cfg` fernet_key value with the one from step 2. Alternatively, you can store your fernet_key in OS environment variable. You do not need to change `airflow.cfg` in this case as Airflow will use environment variable over the value in `airflow.cfg`:

```
# Note the double underscores
EXPORT AIRFLOW__CORE__FERNET_KEY = your_fernet_key
```

4. Restart Airflow webserver.
5. For existing connections (the ones that you had defined before installing `airflow[crypto]` and creating a Fernet key), you need to open each connection in the connection admin UI, re-type the password, and save it.

3.6.5 Writing Logs

3.6.5.1 Writing Logs Locally

Users can specify a logs folder in `airflow.cfg` using the `base_log_folder` setting. By default, it is in the `AIRFLOW_HOME` directory.

In addition, users can supply a remote location for storing logs and log backups in cloud storage.

In the Airflow Web UI, local logs take precedance over remote logs. If local logs can not be found or accessed, the remote logs will be displayed. Note that logs are only sent to remote storage once a task completes (including failure). In other words, remote logs for running tasks are unavailable. Logs are stored in the log folder as `{dag_id}/{task_id}/{execution_date}/{try_number}.log`.

3.6.5.2 Writing Logs to Amazon S3

Before you begin

Remote logging uses an existing Airflow connection to read/write logs. If you don't have a connection properly setup, this will fail.

Enabling remote logging

To enable this feature, `airflow.cfg` must be configured as in this example:

```
[core]
# Airflow can store logs remotely in AWS S3. Users must supply a remote
# location URL (starting with either 's3://...') and an Airflow connection
# id that provides access to the storage location.
remote_base_log_folder = s3://my-bucket/path/to/logs
remote_log_conn_id = MyS3Conn
# Use server-side encryption for logs stored in S3
encrypt_s3_logs = False
```

In the above example, Airflow will try to use `S3Hook('MyS3Conn')`.

3.6.5.3 Writing Logs to Azure Blob Storage

Airflow can be configured to read and write task logs in Azure Blob Storage. Follow the steps below to enable Azure Blob Storage logging.

1. Airflow's logging system requires a custom .py file to be located in the `PYTHONPATH`, so that it's importable from Airflow. Start by creating a directory to store the config file. `$AIRFLOW_HOME/config` is recommended.
2. Create empty files called `$AIRFLOW_HOME/config/log_config.py` and `$AIRFLOW_HOME/config/__init__.py`.
3. Copy the contents of `airflow/config_templates/airflow_local_settings.py` into the `log_config.py` file that was just created in the step above.
4. Customize the following portions of the template:

```
# wasb buckets should start with "wasb" just to help Airflow select␣
↪correct handler
REMOTE_BASE_LOG_FOLDER = 'wasb-<whatever you want here>'

# Rename DEFAULT_LOGGING_CONFIG to LOGGING CONFIG
LOGGING_CONFIG = ...
```

5. Make sure a Azure Blob Storage (Wasb) connection hook has been defined in Airflow. The hook should have read and write access to the Azure Blob Storage bucket defined above in `REMOTE_BASE_LOG_FOLDER`.

6. Update `$AIRFLOW_HOME/airflow.cfg` to contain:

```
remote_logging = True
logging_config_class = log_config.LOGGING_CONFIG
remote_log_conn_id = <name of the Azure Blob Storage connection>
```

7. Restart the Airflow webserver and scheduler, and trigger (or wait for) a new task execution.
8. Verify that logs are showing up for newly executed tasks in the bucket you've defined.

3.6.5.4 Writing Logs to Google Cloud Storage

Follow the steps below to enable Google Cloud Storage logging.

1. Airflow's logging system requires a custom .py file to be located in the `PYTHONPATH`, so that it's importable from Airflow. Start by creating a directory to store the config file. `$AIRFLOW_HOME/config` is recommended.
2. Create empty files called `$AIRFLOW_HOME/config/log_config.py` and `$AIRFLOW_HOME/config/__init__.py`.
3. Copy the contents of `airflow/config_templates/airflow_local_settings.py` into the `log_config.py` file that was just created in the step above.
4. Customize the following portions of the template:

```
# Add this variable to the top of the file. Note the trailing slash.
GCS_LOG_FOLDER = 'gs://<bucket where logs should be persisted>/'

# Rename DEFAULT_LOGGING_CONFIG to LOGGING CONFIG
LOGGING_CONFIG = ...

# Add a GCSTaskHandler to the 'handlers' block of the LOGGING_CONFIG␣
↪variable
'gcs.task': {
    'class': 'airflow.utils.log.gcs_task_handler.GCSTaskHandler',
    'formatter': 'airflow.task',
    'base_log_folder': os.path.expanduser(BASE_LOG_FOLDER),
    'gcs_log_folder': GCS_LOG_FOLDER,
    'filename_template': FILENAME_TEMPLATE,
},

# Update the airflow.task and airflow.tas_runner blocks to be 'gcs.task'␣
↪instead of 'file.task'.
'loggers': {
    'airflow.task': {
        'handlers': ['gcs.task'],
        ...
    },
    'airflow.task_runner': {
        'handlers': ['gcs.task'],
        ...
    },
    'airflow': {
        'handlers': ['console'],
        ...
    },
}
```

5. Make sure a Google Cloud Platform connection hook has been defined in Airflow. The hook should have read and write access to the Google Cloud Storage bucket defined above in `GCS_LOG_FOLDER`.

6. Update `$AIRFLOW_HOME/airflow.cfg` to contain:

```
task_log_reader = gcs.task
logging_config_class = log_config.LOGGING_CONFIG
remote_log_conn_id = <name of the Google cloud platform hook>
```

7. Restart the Airflow webserver and scheduler, and trigger (or wait for) a new task execution.

8. Verify that logs are showing up for newly executed tasks in the bucket you've defined.

9. Verify that the Google Cloud Storage viewer is working in the UI. Pull up a newly executed task, and verify that you see something like:

```
*** Reading remote log from gs://<bucket where logs should be persisted>/
↪example_bash_operator/run_this_last/2017-10-03T00:00:00/16.log.
[2017-10-03 21:57:50,056] {cli.py:377} INFO - Running on host chrisr-00532
[2017-10-03 21:57:50,093] {base_task_runner.py:115} INFO - Running: ['bash
↪', '-c', u'airflow run example_bash_operator run_this_last 2017-10-
↪03T00:00:00 --job_id 47 --raw -sd DAGS_FOLDER/example_dags/example_bash_
↪operator.py']
[2017-10-03 21:57:51,264] {base_task_runner.py:98} INFO - Subtask: [2017-
↪10-03 21:57:51,263] {__init__.py:45} INFO - Using executor␣
↪SequentialExecutor
[2017-10-03 21:57:51,306] {base_task_runner.py:98} INFO - Subtask: [2017-
↪10-03 21:57:51,306] {models.py:186} INFO - Filling up the DagBag from /
↪airflow/dags/example_dags/example_bash_operator.py
```

Note the top line that says it's reading from the remote log file.

Please be aware that if you were persisting logs to Google Cloud Storage using the old-style airflow.cfg configuration method, the old logs will no longer be visible in the Airflow UI, though they'll still exist in Google Cloud Storage. This is a backwards incompatbile change. If you are unhappy with it, you can change the `FILENAME_TEMPLATE` to reflect the old-style log filename format.

3.6.6 Scaling Out with Celery

`CeleryExecutor` is one of the ways you can scale out the number of workers. For this to work, you need to setup a Celery backend (**RabbitMQ**, **Redis**, ...) and change your `airflow.cfg` to point the executor parameter to `CeleryExecutor` and provide the related Celery settings.

For more information about setting up a Celery broker, refer to the exhaustive Celery documentation on the topic.

Here are a few imperative requirements for your workers:

- `airflow` needs to be installed, and the CLI needs to be in the path
- Airflow configuration settings should be homogeneous across the cluster
- Operators that are executed on the worker need to have their dependencies met in that context. For example, if you use the `HiveOperator`, the hive CLI needs to be installed on that box, or if you use the `MySqlOperator`, the required Python library needs to be available in the `PYTHONPATH` somehow
- The worker needs to have access to its `DAGS_FOLDER`, and you need to synchronize the filesystems by your own means. A common setup would be to store your DAGS_FOLDER in a Git repository and sync it across machines using Chef, Puppet, Ansible, or whatever you use to configure machines in your environment. If all your boxes have a common mount point, having your pipelines files shared there should work as well

To kick off a worker, you need to setup Airflow and kick off the worker subcommand

```
airflow worker
```

Your worker should start picking up tasks as soon as they get fired in its direction.

Note that you can also run "Celery Flower", a web UI built on top of Celery, to monitor your workers. You can use the shortcut command `airflow flower` to start a Flower web server.

Some caveats:

- Make sure to use a database backed result backend
- Make sure to set a visibility timeout in [celery_broker_transport_options] that exceeds the ETA of your longest running task
- Tasks can and consume resources, make sure your worker as enough resources to run *worker_concurrency* tasks

3.6.7 Scaling Out with Dask

`DaskExecutor` allows you to run Airflow tasks in a Dask Distributed cluster.

Dask clusters can be run on a single machine or on remote networks. For complete details, consult the Distributed documentation.

To create a cluster, first start a Scheduler:

```
# default settings for a local cluster
DASK_HOST=127.0.0.1
DASK_PORT=8786

dask-scheduler --host $DASK_HOST --port $DASK_PORT
```

Next start at least one Worker on any machine that can connect to the host:

```
dask-worker $DASK_HOST:$DASK_PORT
```

Edit your `airflow.cfg` to set your executor to `DaskExecutor` and provide the Dask Scheduler address in the `[dask]` section.

Please note:

- Each Dask worker must be able to import Airflow and any dependencies you require.
- Dask does not support queues. If an Airflow task was created with a queue, a warning will be raised but the task will be submitted to the cluster.

3.6.8 Scaling Out with Mesos (community contributed)

There are two ways you can run airflow as a mesos framework:

1. Running airflow tasks directly on mesos slaves, requiring each mesos slave to have airflow installed and configured.
2. Running airflow tasks inside a docker container that has airflow installed, which is run on a mesos slave.

3.6.8.1 Tasks executed directly on mesos slaves

`MesosExecutor` allows you to schedule airflow tasks on a Mesos cluster. For this to work, you need a running mesos cluster and you must perform the following steps -

1. Install airflow on a mesos slave where web server and scheduler will run, let's refer to this as the "Airflow server".
2. On the Airflow server, install mesos python eggs from mesos downloads.
3. On the Airflow server, use a database (such as mysql) which can be accessed from all mesos slaves and add configuration in `airflow.cfg`.
4. Change your `airflow.cfg` to point executor parameter to *MesosExecutor* and provide related Mesos settings.
5. On all mesos slaves, install airflow. Copy the `airflow.cfg` from Airflow server (so that it uses same sql alchemy connection).
6. On all mesos slaves, run the following for serving logs:

```
airflow serve_logs
```

7. On Airflow server, to start processing/scheduling DAGs on mesos, run:

```
airflow scheduler -p
```

Note: We need -p parameter to pickle the DAGs.

You can now see the airflow framework and corresponding tasks in mesos UI. The logs for airflow tasks can be seen in airflow UI as usual.

For more information about mesos, refer to mesos documentation. For any queries/bugs on *MesosExecutor*, please contact @kapil-malik.

3.6.8.2 Tasks executed in containers on mesos slaves

This gist contains all files and configuration changes necessary to achieve the following:

1. Create a dockerized version of airflow with mesos python eggs installed.

   We recommend taking advantage of docker's multi stage builds in order to achieve this. We have one Dockerfile that defines building a specific version of mesos from source (Dockerfile-mesos), in order to create the python eggs. In the airflow Dockerfile (Dockerfile-airflow) we copy the python eggs from the mesos image.

2. Create a mesos configuration block within the `airflow.cfg`.

   The configuration block remains the same as the default airflow configuration (default_airflow.cfg), but has the addition of an option `docker_image_slave`. This should be set to the name of the image you would like mesos to use when running airflow tasks. Make sure you have the proper configuration of the DNS record for your mesos master and any sort of authorization if any exists.

3. Change your `airflow.cfg` to point the executor parameter to *MesosExecutor* (*executor* = *SequentialExecutor*).

4. Make sure your mesos slave has access to the docker repository you are using for your `docker_image_slave`.

   Instructions are available in the mesos docs.

The rest is up to you and how you want to work with a dockerized airflow configuration.

3.6.9 Running Airflow with systemd

Airflow can integrate with systemd based systems. This makes watching your daemons easy as systemd can take care of restarting a daemon on failure. In the `scripts/systemd` directory you can find unit files that have been tested on Redhat based systems. You can copy those to `/usr/lib/systemd/system`. It is assumed that Airflow will run under `airflow:airflow`. If not (or if you are running on a non Redhat based system) you probably need to adjust the unit files.

Environment configuration is picked up from `/etc/sysconfig/airflow`. An example file is supplied. Make sure to specify the `SCHEDULER_RUNS` variable in this file when you run the scheduler. You can also define here, for example, `AIRFLOW_HOME` or `AIRFLOW_CONFIG`.

3.6.10 Running Airflow with upstart

Airflow can integrate with upstart based systems. Upstart automatically starts all airflow services for which you have a corresponding `*.conf` file in `/etc/init` upon system boot. On failure, upstart automatically restarts the process (until it reaches re-spawn limit set in a `*.conf` file).

You can find sample upstart job files in the `scripts/upstart` directory. These files have been tested on Ubuntu 14.04 LTS. You may have to adjust `start on` and `stop on` stanzas to make it work on other upstart systems. Some of the possible options are listed in `scripts/upstart/README`.

Modify `*.conf` files as needed and copy to `/etc/init` directory. It is assumed that airflow will run under `airflow:airflow`. Change `setuid` and `setgid` in `*.conf` files if you use other user/group

You can use `initctl` to manually start, stop, view status of the airflow process that has been integrated with upstart

```
initctl airflow-webserver status
```

3.6.11 Using the Test Mode Configuration

Airflow has a fixed set of "test mode" configuration options. You can load these at any time by calling `airflow.configuration.load_test_config()` (note this operation is not reversible!). However, some options (like the DAG_FOLDER) are loaded before you have a chance to call load_test_config(). In order to eagerly load the test configuration, set test_mode in airflow.cfg:

```
[tests]
unit_test_mode = True
```

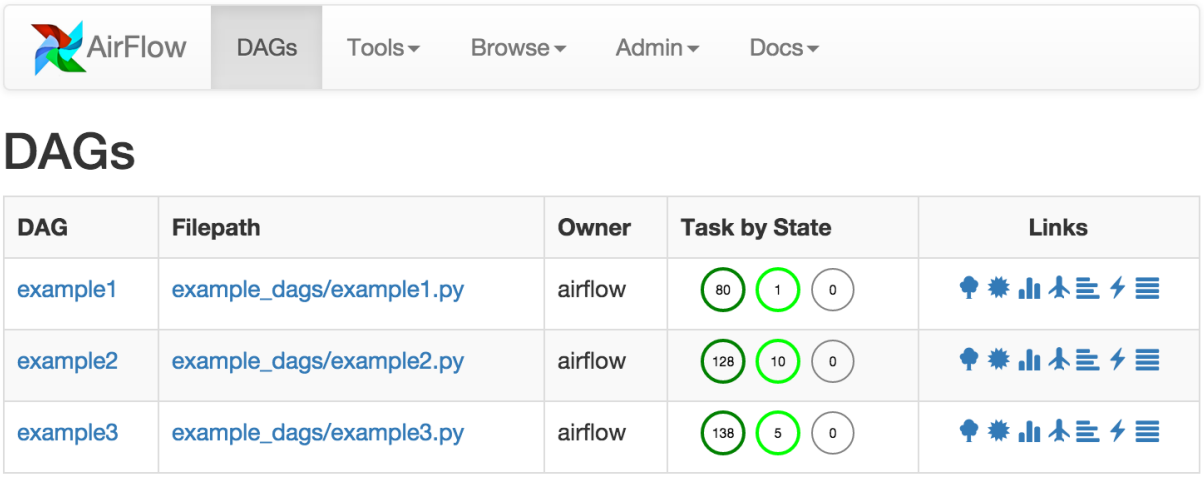
Due to Airflow's automatic environment variable expansion (see *Setting Configuration Options*), you can also set the env var `AIRFLOW__CORE__UNIT_TEST_MODE` to temporarily overwrite airflow.cfg.

3.7 UI / Screenshots

The Airflow UI make it easy to monitor and troubleshoot your data pipelines. Here's a quick overview of some of the features and visualizations you can find in the Airflow UI.

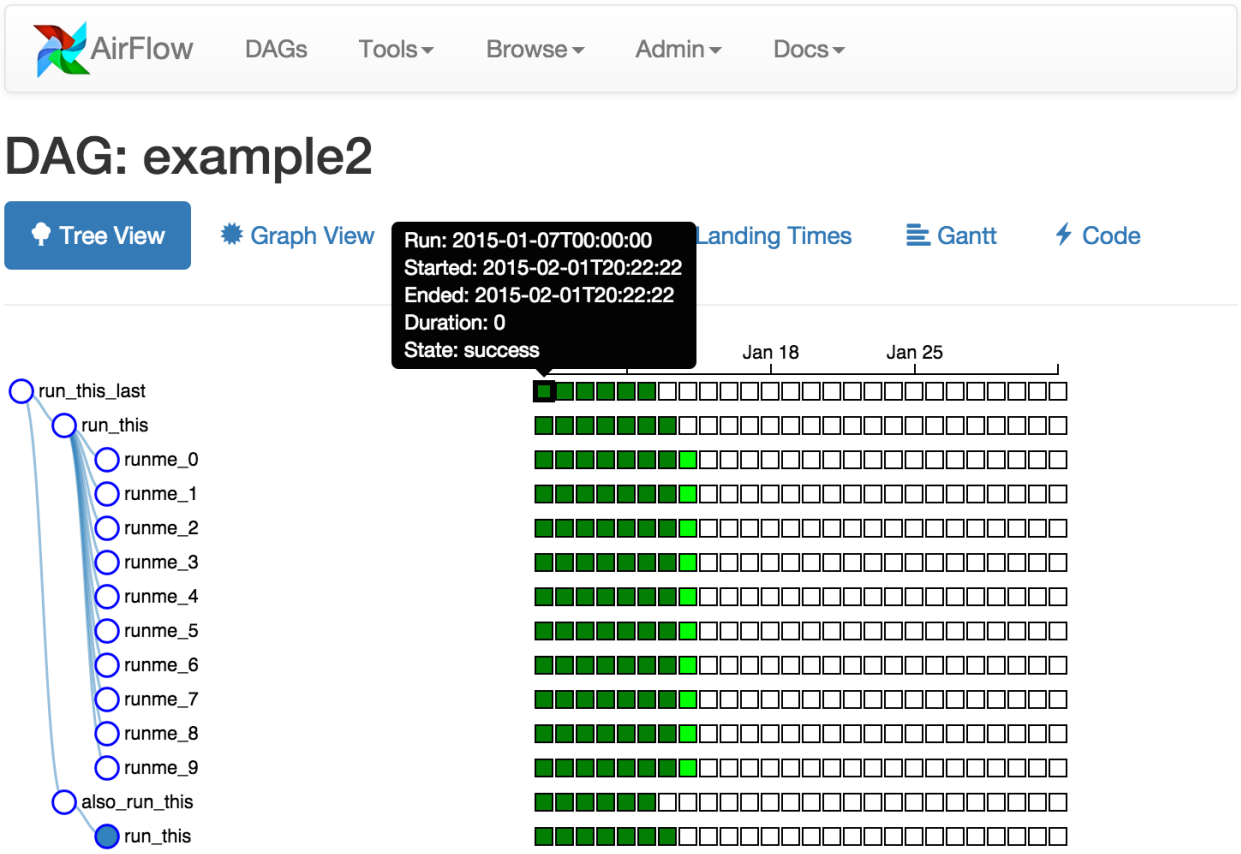
3.7.1 DAGs View

List of the DAGs in your environment, and a set of shortcuts to useful pages. You can see exactly how many tasks succeeded, failed, or are currently running at a glance.



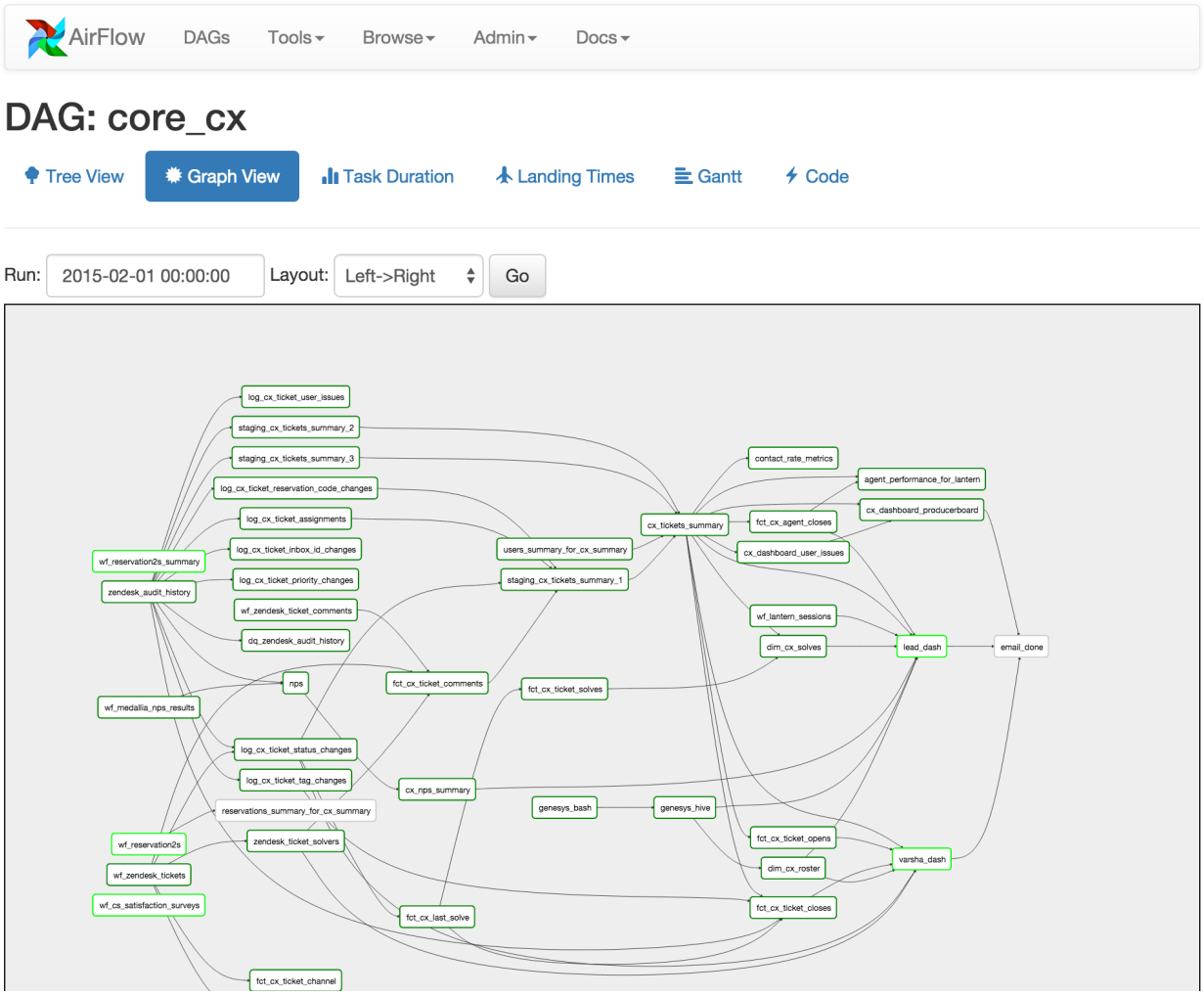
3.7.2 Tree View

A tree representation of the DAG that spans across time. If a pipeline is late, you can quickly see where the different steps are and identify the blocking ones.



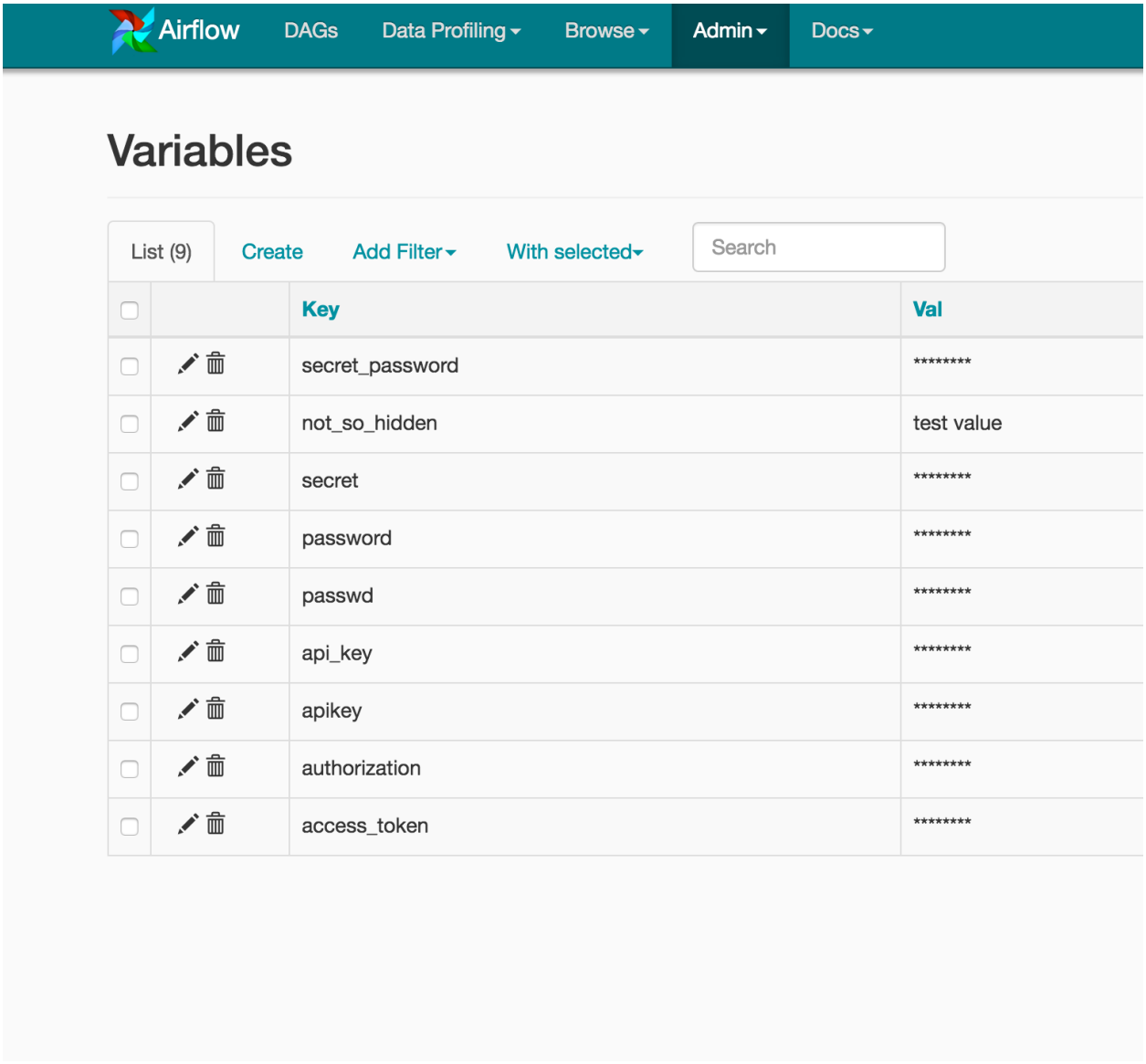
3.7.3 Graph View

The graph view is perhaps the most comprehensive. Visualize your DAG's dependencies and their current status for a specific run.



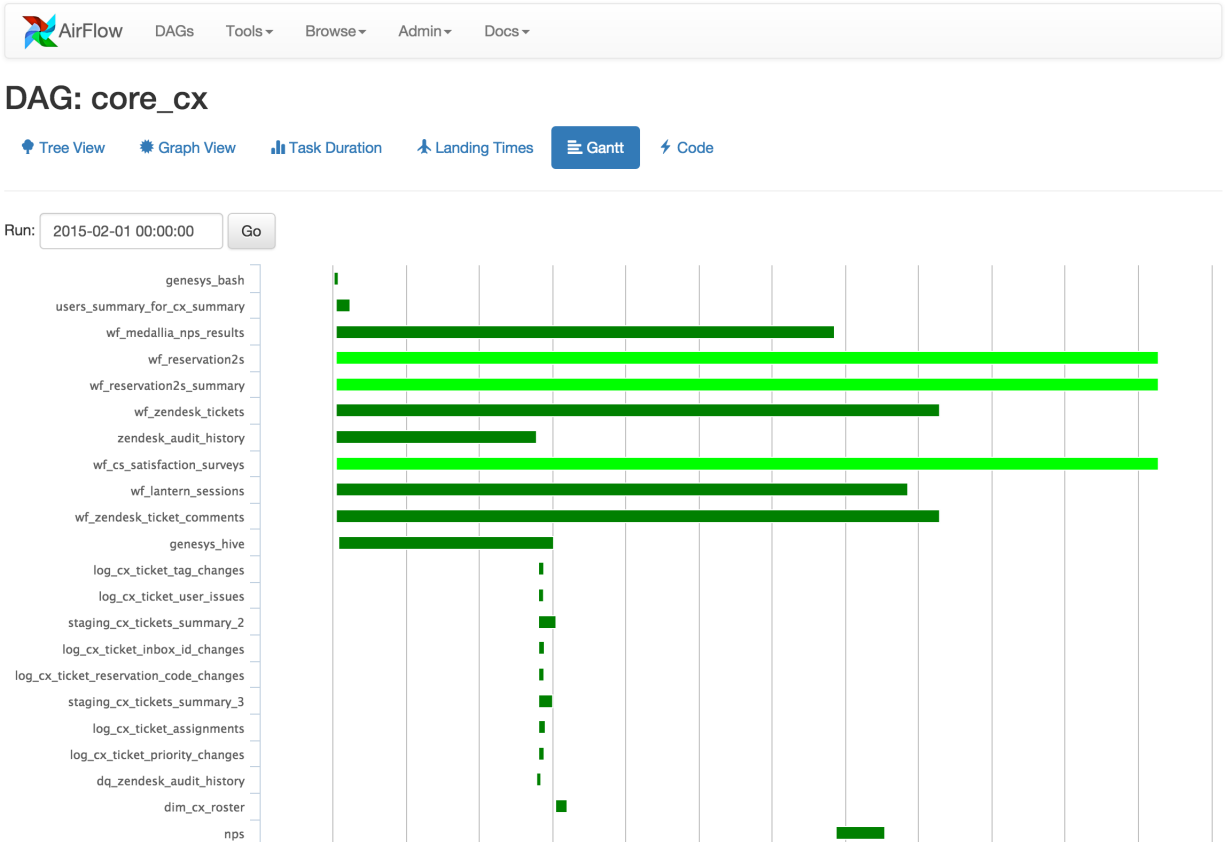
3.7.4 Variable View

The variable view allows you to list, create, edit or delete the key-value pair of a variable used during jobs. Value of a variable will be hidden if the key contains any words in ('password', 'secret', 'passwd', 'authorization', 'api_key', 'apikey', 'access_token') by default, but can be configured to show in clear-text.



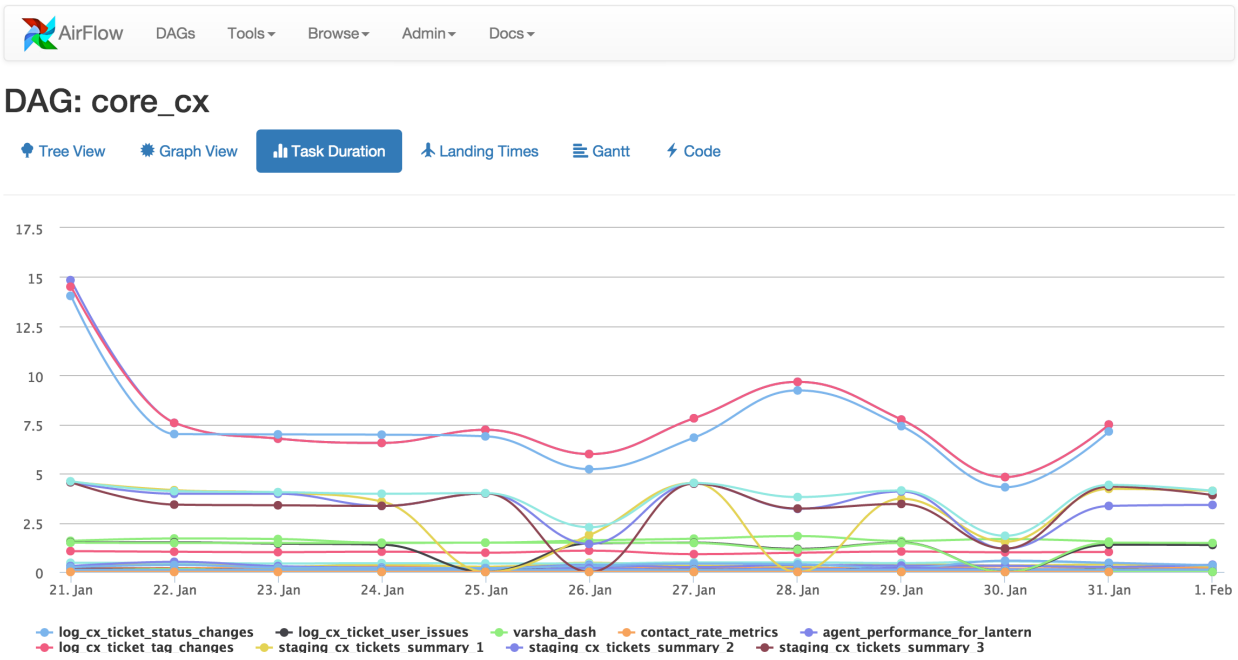
3.7.5 Gantt Chart

The Gantt chart lets you analyse task duration and overlap. You can quickly identify bottlenecks and where the bulk of the time is spent for specific DAG runs.



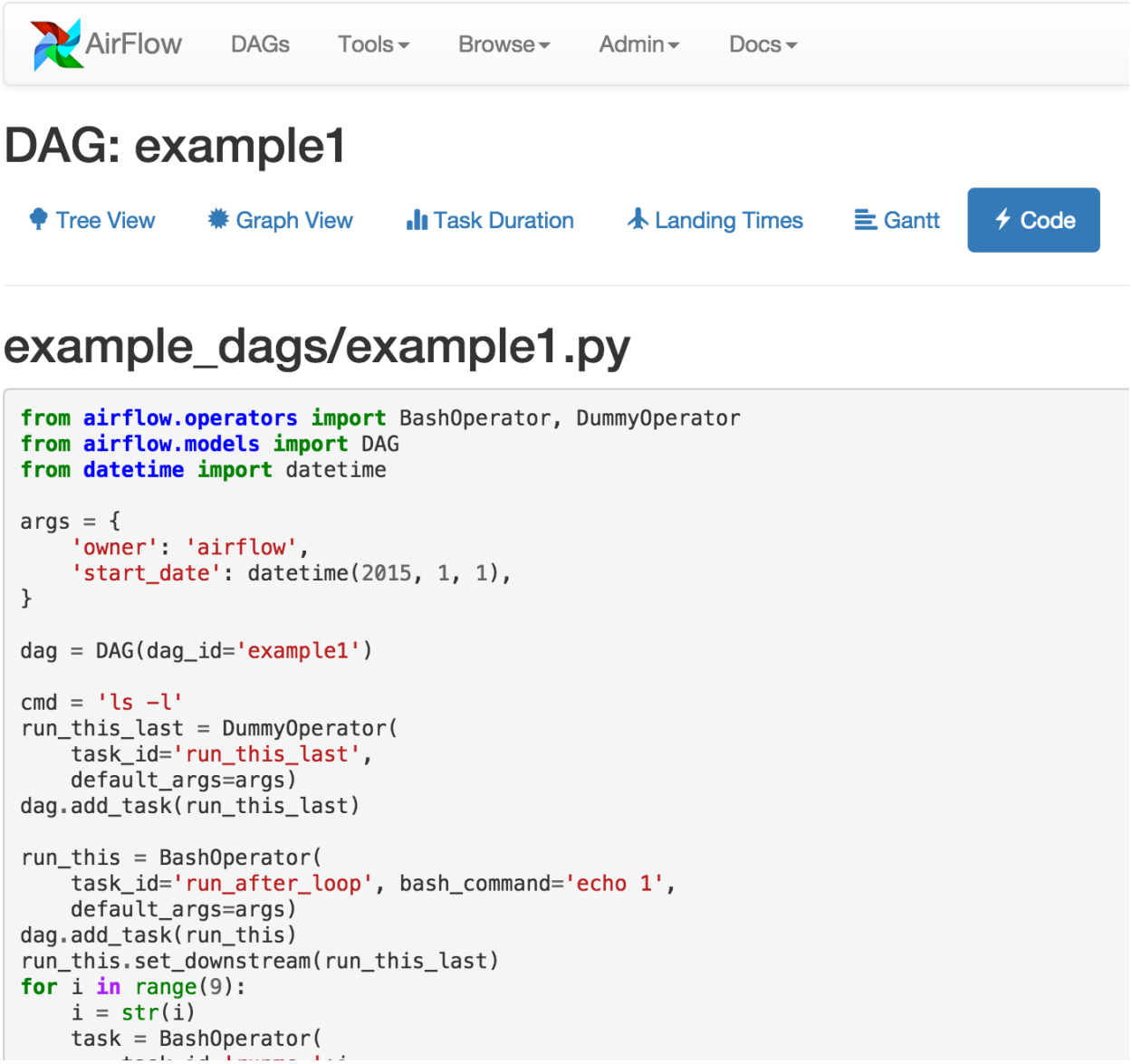
3.7.6 Task Duration

The duration of your different tasks over the past N runs. This view lets you find outliers and quickly understand where the time is spent in your DAG over many runs.



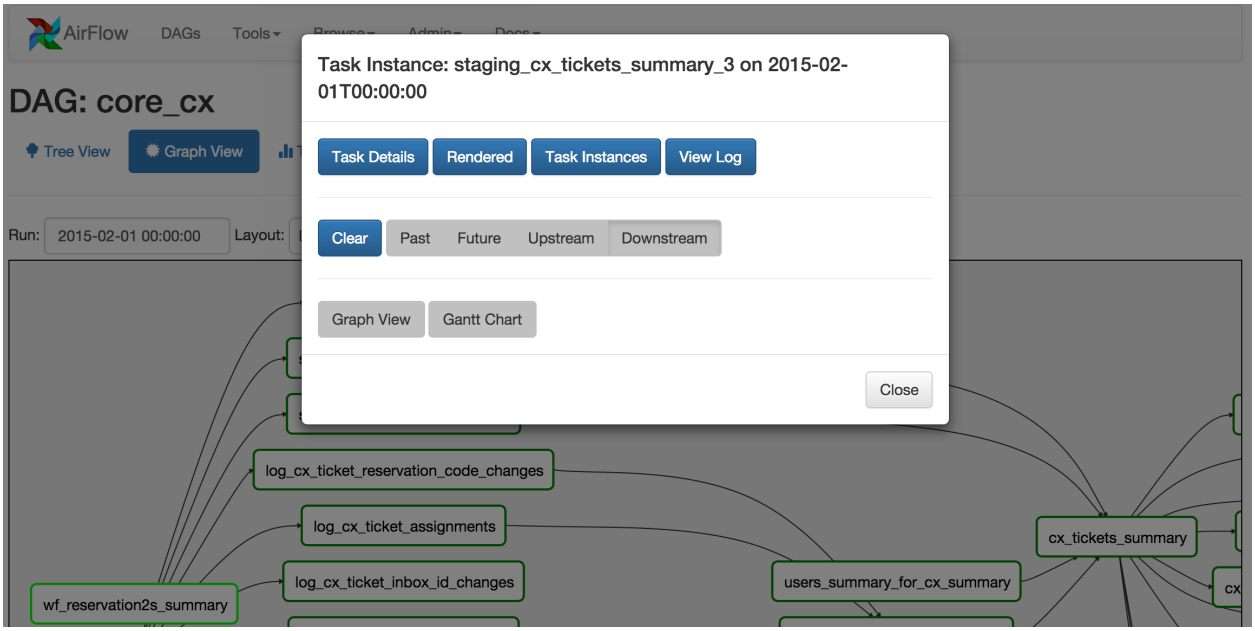
3.7.7 Code View

Transparency is everything. While the code for your pipeline is in source control, this is a quick way to get to the code that generates the DAG and provide yet more context.



3.7.8 Task Instance Context Menu

From the pages seen above (tree view, graph view, gantt, . . .), it is always possible to click on a task instance, and get to this rich context menu that can take you to more detailed metadata, and perform some actions.



3.8 Concepts

The Airflow Platform is a tool for describing, executing, and monitoring workflows.

3.8.1 Core Ideas

3.8.1.1 DAGs

In Airflow, a `DAG` – or a Directed Acyclic Graph – is a collection of all the tasks you want to run, organized in a way that reflects their relationships and dependencies.

For example, a simple DAG could consist of three tasks: A, B, and C. It could say that A has to run successfully before B can run, but C can run anytime. It could say that task A times out after 5 minutes, and B can be restarted up to 5 times in case it fails. It might also say that the workflow will run every night at 10pm, but shouldn't start until a certain date.

In this way, a DAG describes *how* you want to carry out your workflow; but notice that we haven't said anything about *what* we actually want to do! A, B, and C could be anything. Maybe A prepares data for B to analyze while C sends an email. Or perhaps A monitors your location so B can open your garage door while C turns on your house lights. The important thing is that the DAG isn't concerned with what its constituent tasks do; its job is to make sure that whatever they do happens at the right time, or in the right order, or with the right handling of any unexpected issues.

DAGs are defined in standard Python files that are placed in Airflow's `DAG_FOLDER`. Airflow will execute the code in each file to dynamically build the `DAG` objects. You can have as many DAGs as you want, each describing an arbitrary number of tasks. In general, each one should correspond to a single logical workflow.

Note: When searching for DAGs, Airflow will only consider files where the string "airflow" and "DAG" both appear in the contents of the `.py` file.

Scope

Airflow will load any `DAG` object it can import from a DAGfile. Critically, that means the DAG must appear in `globals()`. Consider the following two DAGs. Only `dag_1` will be loaded; the other one only appears in a local scope.

```
dag_1 = DAG('this_dag_will_be_discovered')

def my_function():
    dag_2 = DAG('but_this_dag_will_not')

my_function()
```

Sometimes this can be put to good use. For example, a common pattern with `SubDagOperator` is to define the subdag inside a function so that Airflow doesn't try to load it as a standalone DAG.

Default Arguments

If a dictionary of `default_args` is passed to a DAG, it will apply them to any of its operators. This makes it easy to apply a common parameter to many operators without having to type it many times.

```
default_args = {
    'start_date': datetime(2016, 1, 1),
    'owner': 'Airflow'
}

dag = DAG('my_dag', default_args=default_args)
op = DummyOperator(task_id='dummy', dag=dag)
print(op.owner) # Airflow
```

Context Manager

Added in Airflow 1.8

DAGs can be used as context managers to automatically assign new operators to that DAG.

```
with DAG('my_dag', start_date=datetime(2016, 1, 1)) as dag:
    op = DummyOperator('op')

op.dag is dag # True
```

3.8.1.2 Operators

While DAGs describe *how* to run a workflow, `Operators` determine what actually gets done.

An operator describes a single task in a workflow. Operators are usually (but not always) atomic, meaning they can stand on their own and don't need to share resources with any other operators. The DAG will make sure that operators run in the correct certain order; other than those dependencies, operators generally run independently. In fact, they may run on two completely different machines.

This is a subtle but very important point: in general, if two operators need to share information, like a filename or small amount of data, you should consider combining them into a single operator. If it absolutely can't be avoided, Airflow does have a feature for operator cross-communication called XCom that is described elsewhere in this document.

Airflow provides operators for many common tasks, including:

- `BashOperator` - executes a bash command
- `PythonOperator` - calls an arbitrary Python function
- `EmailOperator` - sends an email
- `HTTPOperator` - sends an HTTP request
- `MySqlOperator`, `SqliteOperator`, `PostgresOperator`, `MsSqlOperator`, `OracleOperator`, `JdbcOperator`, etc. - executes a SQL command
- `Sensor` - waits for a certain time, file, database row, S3 key, etc...

In addition to these basic building blocks, there are many more specific operators: `DockerOperator`, `HiveOperator`, `S3FileTransferOperator`, `PrestoToMysqlOperator`, `SlackOperator`... you get the idea!

The `airflow/contrib/` directory contains yet more operators built by the community. These operators aren't always as complete or well-tested as those in the main distribution, but allow users to more easily add new functionality to the platform.

Operators are only loaded by Airflow if they are assigned to a DAG.

DAG Assignment

Added in Airflow 1.8

Operators do not have to be assigned to DAGs immediately (previously `dag` was a required argument). However, once an operator is assigned to a DAG, it can not be transferred or unassigned. DAG assignment can be done explicitly when the operator is created, through deferred assignment, or even inferred from other operators.

```
dag = DAG('my_dag', start_date=datetime(2016, 1, 1))

# sets the DAG explicitly
explicit_op = DummyOperator(task_id='op1', dag=dag)

# deferred DAG assignment
deferred_op = DummyOperator(task_id='op2')
deferred_op.dag = dag

# inferred DAG assignment (linked operators must be in the same DAG)
inferred_op = DummyOperator(task_id='op3')
inferred_op.set_upstream(deferred_op)
```

Bitshift Composition

Added in Airflow 1.8

Traditionally, operator relationships are set with the `set_upstream()` and `set_downstream()` methods. In Airflow 1.8, this can be done with the Python bitshift operators `>>` and `<<`. The following four statements are all functionally equivalent:

```
op1 >> op2
op1.set_downstream(op2)
```

(continues on next page)

(continued from previous page)

```
op2 << op1
op2.set_upstream(op1)
```

When using the bitshift to compose operators, the relationship is set in the direction that the bitshift operator points. For example, `op1 >> op2` means that `op1` runs first and `op2` runs second. Multiple operators can be composed – keep in mind the chain is executed left-to-right and the rightmost object is always returned. For example:

```
op1 >> op2 >> op3 << op4
```

is equivalent to:

```
op1.set_downstream(op2)
op2.set_downstream(op3)
op3.set_upstream(op4)
```

For convenience, the bitshift operators can also be used with DAGs. For example:

```
dag >> op1 >> op2
```

is equivalent to:

```
op1.dag = dag
op1.set_downstream(op2)
```

We can put this all together to build a simple pipeline:

```
with DAG('my_dag', start_date=datetime(2016, 1, 1)) as dag:
    (
        DummyOperator(task_id='dummy_1')
        >> BashOperator(
            task_id='bash_1',
            bash_command='echo "HELLO!"')
        >> PythonOperator(
            task_id='python_1',
            python_callable=lambda: print("GOODBYE!"))
    )
```

3.8.1.3 Tasks

Once an operator is instantiated, it is referred to as a "task". The instantiation defines specific values when calling the abstract operator, and the parameterized task becomes a node in a DAG.

3.8.1.4 Task Instances

A task instance represents a specific run of a task and is characterized as the combination of a dag, a task, and a point in time. Task instances also have an indicative state, which could be "running", "success", "failed", "skipped", "up for retry", etc.

3.8.1.5 Workflows

You're now familiar with the core building blocks of Airflow. Some of the concepts may sound very similar, but the vocabulary can be conceptualized like this:

- DAG: a description of the order in which work should take place
- Operator: a class that acts as a template for carrying out some work
- Task: a parameterized instance of an operator
- Task Instance: a task that 1) has been assigned to a DAG and 2) has a state associated with a specific run of the DAG

By combining `DAGs` and `Operators` to create `TaskInstances`, you can build complex workflows.

3.8.2 Additional Functionality

In addition to the core Airflow objects, there are a number of more complex features that enable behaviors like limiting simultaneous access to resources, cross-communication, conditional execution, and more.

3.8.2.1 Hooks

Hooks are interfaces to external platforms and databases like Hive, S3, MySQL, Postgres, HDFS, and Pig. Hooks implement a common interface when possible, and act as a building block for operators. They also use the `airflow.models.Connection` model to retrieve hostnames and authentication information. Hooks keep authentication code and information out of pipelines, centralized in the metadata database.

Hooks are also very useful on their own to use in Python scripts, Airflow airflow.operators.PythonOperator, and in interactive environments like iPython or Jupyter Notebook.

3.8.2.2 Pools

Some systems can get overwhelmed when too many processes hit them at the same time. Airflow pools can be used to **limit the execution parallelism** on arbitrary sets of tasks. The list of pools is managed in the UI (`Menu -> Admin -> Pools`) by giving the pools a name and assigning it a number of worker slots. Tasks can then be associated with one of the existing pools by using the `pool` parameter when creating tasks (i.e., instantiating operators).

```
aggregate_db_message_job = BashOperator(
    task_id='aggregate_db_message_job',
    execution_timeout=timedelta(hours=3),
    pool='ep_data_pipeline_db_msg_agg',
    bash_command=aggregate_db_message_job_cmd,
    dag=dag)
aggregate_db_message_job.set_upstream(wait_for_empty_queue)
```

The `pool` parameter can be used in conjunction with `priority_weight` to define priorities in the queue, and which tasks get executed first as slots open up in the pool. The default `priority_weight` is `1`, and can be bumped to any number. When sorting the queue to evaluate which task should be executed next, we use the `priority_weight`, summed up with all of the `priority_weight` values from tasks downstream from this task. You can use this to bump a specific important task and the whole path to that task gets prioritized accordingly.

Tasks will be scheduled as usual while the slots fill up. Once capacity is reached, runnable tasks get queued and their state will show as such in the UI. As slots free up, queued tasks start running based on the `priority_weight` (of the task and its descendants).

Note that by default tasks aren't assigned to any pool and their execution parallelism is only limited to the executor's setting.

3.8.2.3 Connections

The connection information to external systems is stored in the Airflow metadata database and managed in the UI (`Menu -> Admin -> Connections`) A `conn_id` is defined there and hostname / login / password / schema information attached to it. Airflow pipelines can simply refer to the centrally managed `conn_id` without having to hard code any of this information anywhere.

Many connections with the same `conn_id` can be defined and when that is the case, and when the **hooks** uses the `get_connection` method from `BaseHook`, Airflow will choose one connection randomly, allowing for some basic load balancing and fault tolerance when used in conjunction with retries.

Many hooks have a default `conn_id`, where operators using that hook do not need to supply an explicit connection ID. For example, the default `conn_id` for the `PostgresHook` is `postgres_default`.

See *Managing Connections* for how to create and manage connections.

3.8.2.4 Queues

When using the CeleryExecutor, the celery queues that tasks are sent to can be specified. `queue` is an attribute of BaseOperator, so any task can be assigned to any queue. The default queue for the environment is defined in the `airflow.cfg`'s `celery -> default_queue`. This defines the queue that tasks get assigned to when not specified, as well as which queue Airflow workers listen to when started.

Workers can listen to one or multiple queues of tasks. When a worker is started (using the command `airflow worker`), a set of comma delimited queue names can be specified (e.g. `airflow worker -q spark`). This worker will then only pick up tasks wired to the specified queue(s).

This can be useful if you need specialized workers, either from a resource perspective (for say very lightweight tasks where one worker could take thousands of tasks without a problem), or from an environment perspective (you want a worker running from within the Spark cluster itself because it needs a very specific environment and security rights).

3.8.2.5 XComs

XComs let tasks exchange messages, allowing more nuanced forms of control and shared state. The name is an abbreviation of "cross-communication". XComs are principally defined by a key, value, and timestamp, but also track attributes like the task/DAG that created the XCom and when it should become visible. Any object that can be pickled can be used as an XCom value, so users should make sure to use objects of appropriate size.

XComs can be "pushed" (sent) or "pulled" (received). When a task pushes an XCom, it makes it generally available to other tasks. Tasks can push XComs at any time by calling the `xcom_push()` method. In addition, if a task returns a value (either from its Operator's `execute()` method, or from a PythonOperator's `python_callable` function), then an XCom containing that value is automatically pushed.

Tasks call `xcom_pull()` to retrieve XComs, optionally applying filters based on criteria like `key`, source `task_ids`, and source `dag_id`. By default, `xcom_pull()` filters for the keys that are automatically given to XComs when they are pushed by being returned from execute functions (as opposed to XComs that are pushed manually).

If `xcom_pull` is passed a single string for `task_ids`, then the most recent XCom value from that task is returned; if a list of `task_ids` is passed, then a corresponding list of XCom values is returned.

```
# inside a PythonOperator called 'pushing_task'
def push_function():
    return value

# inside another PythonOperator where provide_context=True
```

(continues on next page)

(continued from previous page)

```
def pull_function(**context):
    value = context['task_instance'].xcom_pull(task_ids='pushing_task')
```

It is also possible to pull XCom directly in a template, here's an example of what this may look like:

```
SELECT * FROM {{ task_instance.xcom_pull(task_ids='foo', key='table_name') }}
```

Note that XComs are similar to *Variables*, but are specifically designed for inter-task communication rather than global settings.

3.8.2.6 Variables

Variables are a generic way to store and retrieve arbitrary content or settings as a simple key value store within Airflow. Variables can be listed, created, updated and deleted from the UI (`Admin -> Variables`), code or CLI. In addition, json settings files can be bulk uploaded through the UI. While your pipeline code definition and most of your constants and variables should be defined in code and stored in source control, it can be useful to have some variables or configuration items accessible and modifiable through the UI.

```
from airflow.models import Variable
foo = Variable.get("foo")
bar = Variable.get("bar", deserialize_json=True)
```

The second call assumes `json` content and will be deserialized into `bar`. Note that `Variable` is a sqlalchemy model and can be used as such.

You can use a variable from a jinja template with the syntax :

```
echo {{ var.value.<variable_name> }}
```

or if you need to deserialize a json object from the variable :

```
echo {{ var.json.<variable_name> }}
```

3.8.2.7 Branching

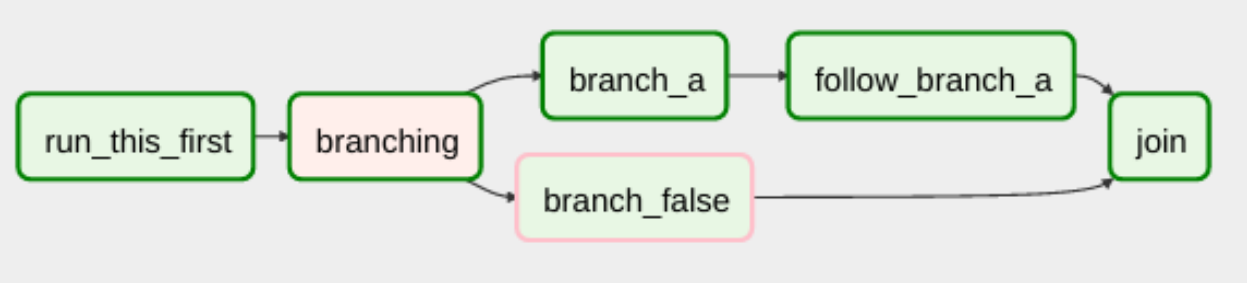
Sometimes you need a workflow to branch, or only go down a certain path based on an arbitrary condition which is typically related to something that happened in an upstream task. One way to do this is by using the `BranchPythonOperator`.

The `BranchPythonOperator` is much like the PythonOperator except that it expects a python_callable that returns a task_id. The task_id returned is followed, and all of the other paths are skipped. The task_id returned by the Python function has to be referencing a task directly downstream from the BranchPythonOperator task.

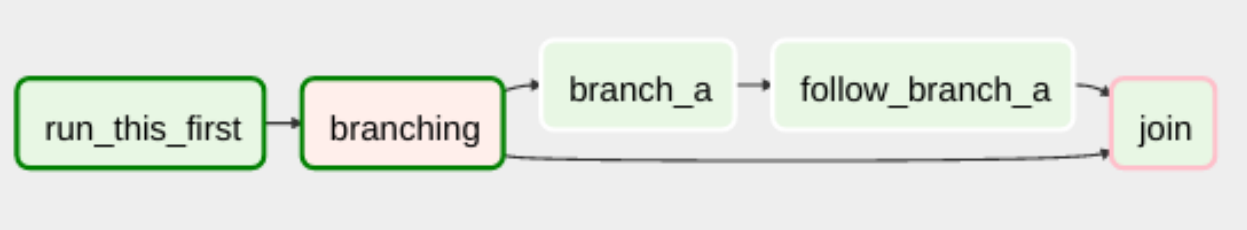
Note that using tasks with `depends_on_past=True` downstream from `BranchPythonOperator` is logically unsound as `skipped` status will invariably lead to block tasks that depend on their past successes. `skipped` states propagates where all directly upstream tasks are `skipped`.

If you want to skip some tasks, keep in mind that you can't have an empty path, if so make a dummy task.

like this, the dummy task "branch_false" is skipped



Not like this, where the join task is skipped

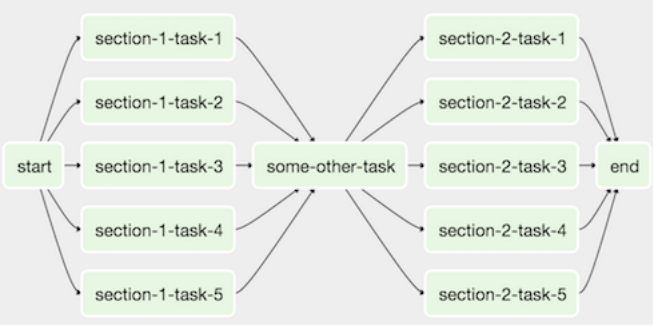


3.8.2.8 SubDAGs

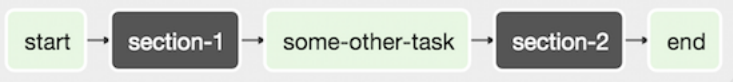
SubDAGs are perfect for repeating patterns. Defining a function that returns a DAG object is a nice design pattern when using Airflow.

Airbnb uses the *stage-check-exchange* pattern when loading data. Data is staged in a temporary table, after which data quality checks are performed against that table. Once the checks all pass the partition is moved into the production table.

As another example, consider the following DAG:



We can combine all of the parallel `task-*` operators into a single SubDAG, so that the resulting DAG resembles the following:



Note that SubDAG operators should contain a factory method that returns a DAG object. This will prevent the SubDAG from being treated like a separate DAG in the main UI. For example:

```
#dags/subdag.py
from airflow.models import DAG
from airflow.operators.dummy_operator import DummyOperator
```

(continues on next page)

(continued from previous page)

```
# Dag is returned by a factory method
def sub_dag(parent_dag_name, child_dag_name, start_date, schedule_interval):
  dag = DAG(
    '%s.%s' % (parent_dag_name, child_dag_name),
    schedule_interval=schedule_interval,
    start_date=start_date,
  )

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

  return dag
```

This SubDAG can then be referenced in your main DAG file:

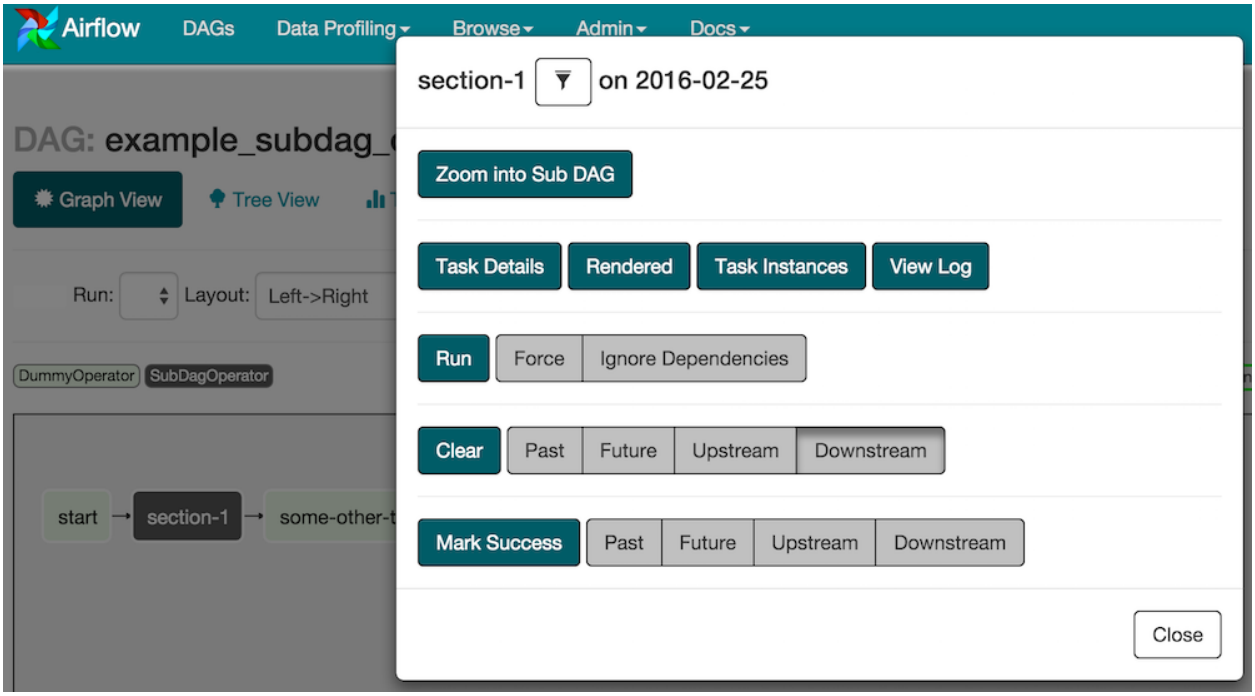
```
# main_dag.py
from datetime import datetime, timedelta
from airflow.models import DAG
from airflow.operators.subdag_operator import SubDagOperator
from dags.subdag import sub_dag


PARENT_DAG_NAME = 'parent_dag'
CHILD_DAG_NAME = 'child_dag'

main_dag = DAG(
  dag_id=PARENT_DAG_NAME,
  schedule_interval=timedelta(hours=1),
  start_date=datetime(2016, 1, 1)
)

sub_dag = SubDagOperator(
  subdag=sub_dag(PARENT_DAG_NAME, CHILD_DAG_NAME, main_dag.start_date,
                 main_dag.schedule_interval),
  task_id=CHILD_DAG_NAME,
  dag=main_dag,
)
```

You can zoom into a SubDagOperator from the graph view of the main DAG to show the tasks contained within the SubDAG:



Some other tips when using SubDAGs:

- by convention, a SubDAG's `dag_id` should be prefixed by its parent and a dot. As in `parent.child`
- share arguments between the main DAG and the SubDAG by passing arguments to the SubDAG operator (as demonstrated above)
- SubDAGs must have a schedule and be enabled. If the SubDAG's schedule is set to `None` or `@once`, the SubDAG will succeed without having done anything
- clearing a SubDagOperator also clears the state of the tasks within
- marking success on a SubDagOperator does not affect the state of the tasks within
- refrain from using `depends_on_past=True` in tasks within the SubDAG as this can be confusing
- it is possible to specify an executor for the SubDAG. It is common to use the SequentialExecutor if you want to run the SubDAG in-process and effectively limit its parallelism to one. Using LocalExecutor can be problematic as it may over-subscribe your worker, running multiple tasks in a single slot

See `airflow/example_dags` for a demonstration.

3.8.2.9 SLAs

Service Level Agreements, or time by which a task or DAG should have succeeded, can be set at a task level as a `timedelta`. If one or many instances have not succeeded by that time, an alert email is sent detailing the list of tasks that missed their SLA. The event is also recorded in the database and made available in the web UI under `Browse->Missed SLAs` where events can be analyzed and documented.

3.8.2.10 Trigger Rules

Though the normal workflow behavior is to trigger tasks when all their directly upstream tasks have succeeded, Airflow allows for more complex dependency settings.

All operators have a `trigger_rule` argument which defines the rule by which the generated task get triggered. The default value for `trigger_rule` is `all_success` and can be defined as "trigger this task when all directly upstream tasks have succeeded". All other rules described here are based on direct parent tasks and are values that can be passed to any operator while creating tasks:

- `all_success`: (default) all parents have succeeded
- `all_failed`: all parents are in a `failed` or `upstream_failed` state
- `all_done`: all parents are done with their execution
- `one_failed`: fires as soon as at least one parent has failed, it does not wait for all parents to be done
- `one_success`: fires as soon as at least one parent succeeds, it does not wait for all parents to be done
- `dummy`: dependencies are just for show, trigger at will

Note that these can be used in conjunction with `depends_on_past` (boolean) that, when set to `True`, keeps a task from getting triggered if the previous schedule for the task hasn't succeeded.

3.8.2.11 Latest Run Only

Standard workflow behavior involves running a series of tasks for a particular date/time range. Some workflows, however, perform tasks that are independent of run time but need to be run on a schedule, much like a standard cron job. In these cases, backfills or running jobs missed during a pause just wastes CPU cycles.

For situations like this, you can use the `LatestOnlyOperator` to skip tasks that are not being run during the most recent scheduled run for a DAG. The `LatestOnlyOperator` skips all immediate downstream tasks, and itself, if the time right now is not between its `execution_time` and the next scheduled `execution_time`.

One must be aware of the interaction between skipped tasks and trigger rules. Skipped tasks will cascade through trigger rules `all_success` and `all_failed` but not `all_done`, `one_failed`, `one_success`, and `dummy`. If you would like to use the `LatestOnlyOperator` with trigger rules that do not cascade skips, you will need to ensure that the `LatestOnlyOperator` is **directly** upstream of the task you would like to skip.

It is possible, through use of trigger rules to mix tasks that should run in the typical date/time dependent mode and those using the `LatestOnlyOperator`.

For example, consider the following dag:

```
#dags/latest_only_with_trigger.py
import datetime as dt

from airflow.models import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.latest_only_operator import LatestOnlyOperator
from airflow.utils.trigger_rule import TriggerRule


dag = DAG(
    dag_id='latest_only_with_trigger',
    schedule_interval=dt.timedelta(hours=4),
    start_date=dt.datetime(2016, 9, 20),
)

latest_only = LatestOnlyOperator(task_id='latest_only', dag=dag)

task1 = DummyOperator(task_id='task1', dag=dag)
task1.set_upstream(latest_only)
```

(continues on next page)

(continued from previous page)

```
task2 = DummyOperator(task_id='task2', dag=dag)

task3 = DummyOperator(task_id='task3', dag=dag)
task3.set_upstream([task1, task2])

task4 = DummyOperator(task_id='task4', dag=dag,
                      trigger_rule=TriggerRule.ALL_DONE)
task4.set_upstream([task1, task2])
```

In the case of this dag, the `latest_only` task will show up as skipped for all runs except the latest run. `task1` is directly downstream of `latest_only` and will also skip for all runs except the latest. `task2` is entirely independent of `latest_only` and will run in all scheduled periods. `task3` is downstream of `task1` and `task2` and because of the default `trigger_rule` being `all_success` will receive a cascaded skip from `task1`. `task4` is downstream of `task1` and `task2` but since its `trigger_rule` is set to `all_done` it will trigger as soon as `task1` has been skipped (a valid completion state) and `task2` has succeeded.

3.8.2.12 Zombies & Undeads

Task instances die all the time, usually as part of their normal life cycle, but sometimes unexpectedly.

Zombie tasks are characterized by the absence of an heartbeat (emitted by the job periodically) and a `running` status in the database. They can occur when a worker node can't reach the database, when Airflow processes are killed externally, or when a node gets rebooted for instance. Zombie killing is performed periodically by the scheduler's process.

Undead processes are characterized by the existence of a process and a matching heartbeat, but Airflow isn't aware of this task as `running` in the database. This mismatch typically occurs as the state of the database is altered, most likely by deleting rows in the "Task Instances" view in the UI. Tasks are instructed to verify their state as part of the heartbeat routine, and terminate themselves upon figuring out that they are in this "undead" state.

3.8.2.13 Cluster Policy

Your local airflow settings file can define a `policy` function that has the ability to mutate task attributes based on other task or DAG attributes. It receives a single argument as a reference to task objects, and is expected to alter its attributes.

For example, this function could apply a specific queue property when using a specific operator, or enforce a task timeout policy, making sure that no tasks run for more than 48 hours. Here's an example of what this may look like

inside your `airflow_settings.py`:

```
def policy(task):
    if task.__class__.__name__ == 'HivePartitionSensor':
        task.queue = "sensor_queue"
    if task.timeout > timedelta(hours=48):
        task.timeout = timedelta(hours=48)
```

3.8.2.14 Documentation & Notes

It's possible to add documentation or notes to your dags & task objects that become visible in the web interface ("Graph View" for dags, "Task Details" for tasks). There are a set of special task attributes that get rendered as rich content if defined:

attribute	rendered to
doc	monospace
doc_json	json
doc_yaml	yaml
doc_md	markdown
doc_rst	reStructuredText

Please note that for dags, doc_md is the only attribute interpreted.

This is especially useful if your tasks are built dynamically from configuration files, it allows you to expose the configuration that led to the related tasks in Airflow.

```
"""
### My great DAG
"""

dag = DAG('my_dag', default_args=default_args)
dag.doc_md = __doc__

t = BashOperator("foo", dag=dag)
t.doc_md = """\
#Title"
Here's a [url](www.airbnb.com)
"""
```

This content will get rendered as markdown respectively in the "Graph View" and "Task Details" pages.

3.8.2.15 Jinja Templating

Airflow leverages the power of Jinja Templating and this can be a powerful tool to use in combination with macros (see the *Macros* section).

For example, say you want to pass the execution date as an environment variable to a Bash script using the `BashOperator`.

```
# The execution date as YYYY-MM-DD
date = "{{ ds }}"
t = BashOperator(
    task_id='test_env',
    bash_command='/tmp/test.sh ',
```

(continues on next page)

(continued from previous page)

```
    dag=dag,
    env={'EXECUTION_DATE': date})
```

Here, `{{ ds }}` is a macro, and because the `env` parameter of the `BashOperator` is templated with Jinja, the execution date will be available as an environment variable named `EXECUTION_DATE` in your Bash script.

You can use Jinja templating with every parameter that is marked as "templated" in the documentation. Template substitution occurs just before the pre_execute function of your operator is called.

3.8.3 Packaged dags

While often you will specify dags in a single `.py` file it might sometimes be required to combine dag and its dependencies. For example, you might want to combine several dags together to version them together or you might want to manage them together or you might need an extra module that is not available by default on the system you are running airflow on. To allow this you can create a zip file that contains the dag(s) in the root of the zip file and have the extra modules unpacked in directories.

For instance you can create a zip file that looks like this:

```
my_dag1.py
my_dag2.py
package1/__init__.py
package1/functions.py
```

Airflow will scan the zip file and try to load `my_dag1.py` and `my_dag2.py`. It will not go into subdirectories as these are considered to be potential packages.

In case you would like to add module dependencies to your DAG you basically would do the same, but then it is more to use a virtualenv and pip.

```
virtualenv zip_dag
source zip_dag/bin/activate

mkdir zip_dag_contents
cd zip_dag_contents

pip install --install-option="--install-lib=$PWD" my_useful_package
cp ~/my_dag.py .

zip -r zip_dag.zip *
```

Note: the zip file will be inserted at the beginning of module search list (sys.path) and as such it will be available to any other code that resides within the same interpreter.

Note: packaged dags cannot be used with pickling turned on.

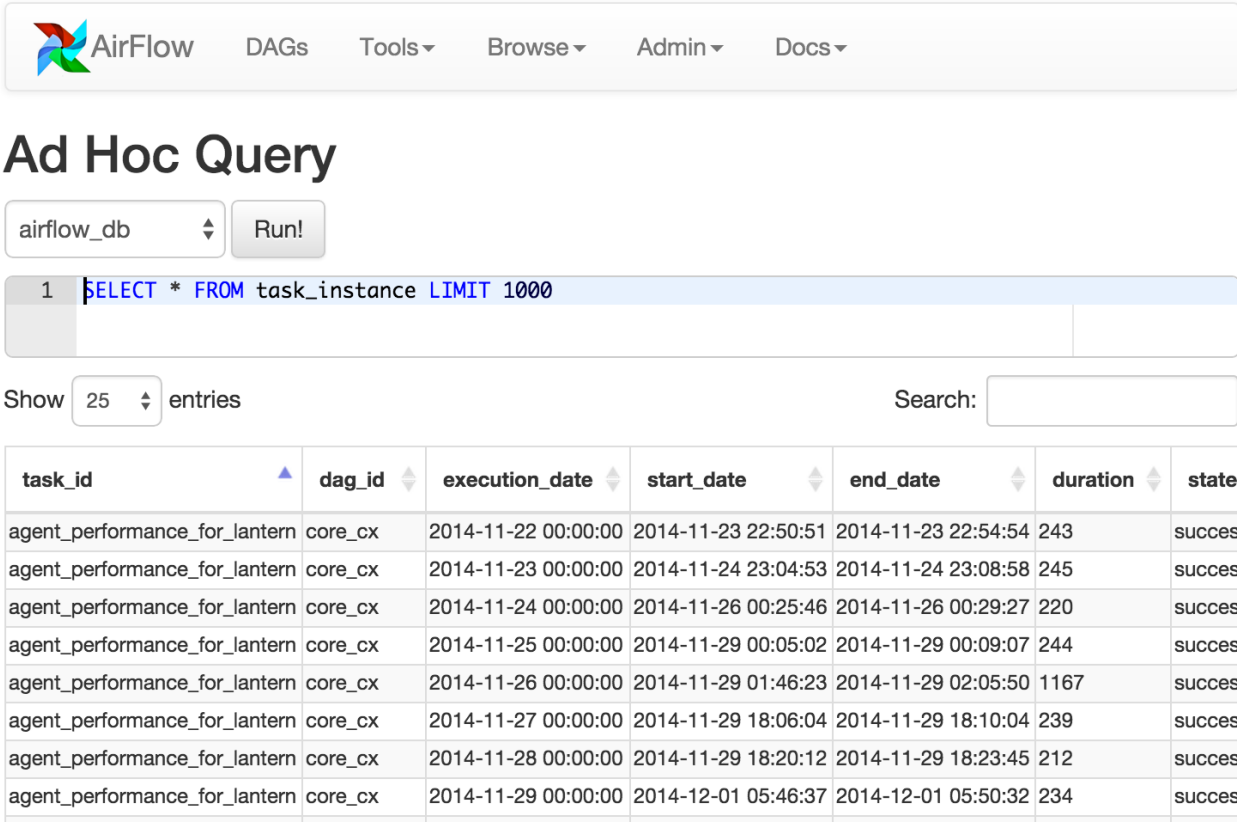
Note: packaged dags cannot contain dynamic libraries (eg. libz.so) these need to be available on the system if a module needs those. In other words only pure python modules can be packaged.

3.9 Data Profiling

Part of being productive with data is having the right weapons to profile the data you are working with. Airflow provides a simple query interface to write SQL and get results quickly, and a charting application letting you visualize data.

3.9.1 Adhoc Queries

The adhoc query UI allows for simple SQL interactions with the database connections registered in Airflow.



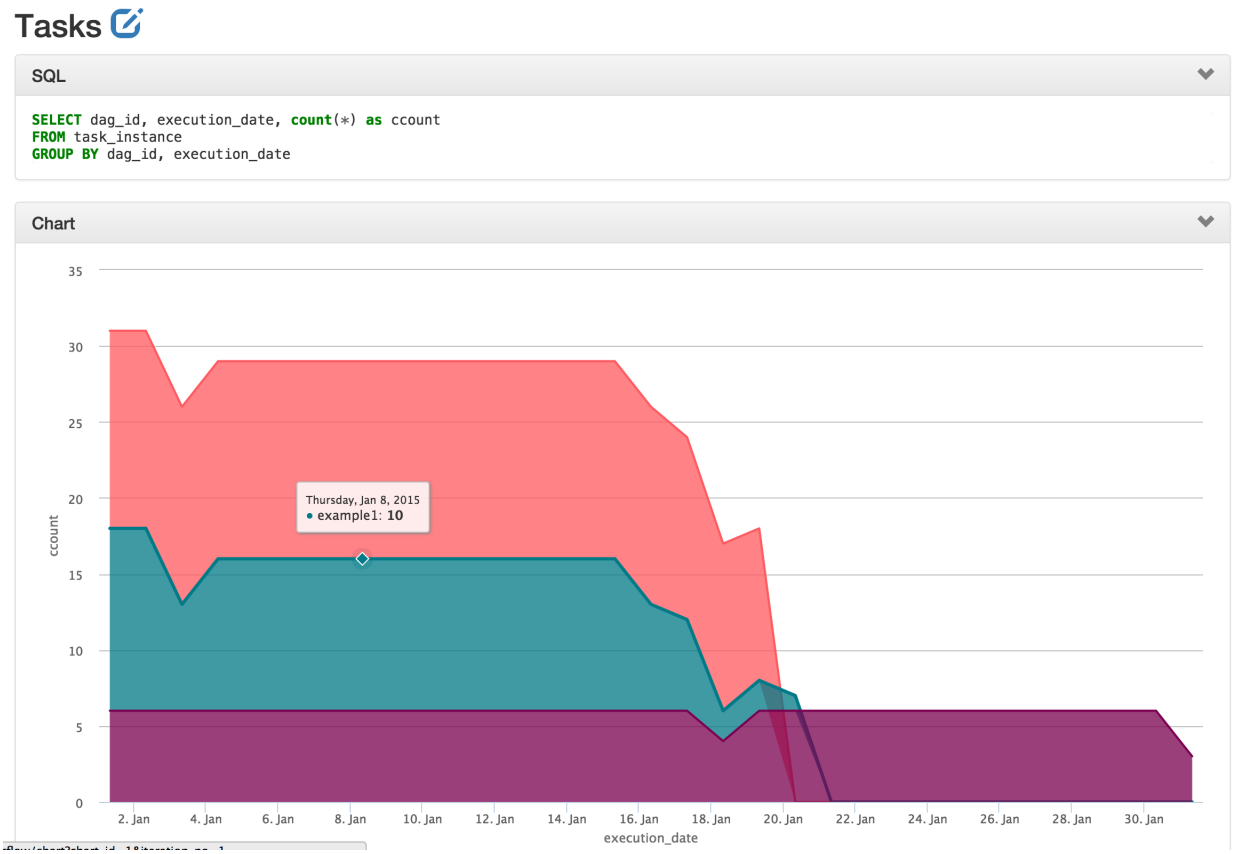
3.9.2 Charts

A simple UI built on top of flask-admin and highcharts allows building data visualizations and charts easily. Fill in a form with a label, SQL, chart type, pick a source database from your environment's connections, select a few other options, and save it for later use.

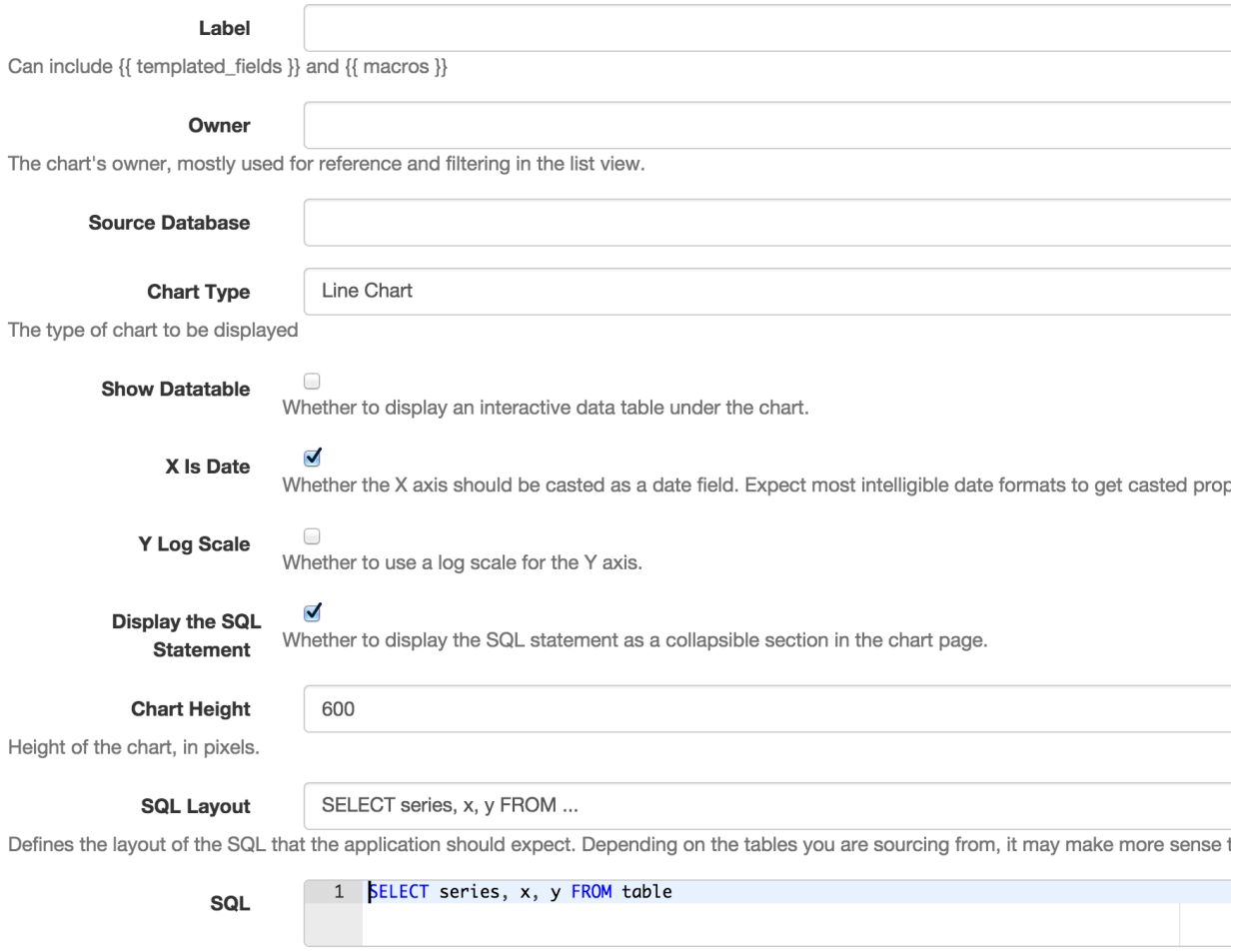
You can even use the same templating and macros available when writing airflow pipelines, parameterizing your queries and modifying parameters directly in the URL.

These charts are basic, but they're easy to create, modify and share.

3.9.2.1 Chart Screenshot



3.9.2.2 Chart Form Screenshot



3.10 Command Line Interface

Airflow has a very rich command line interface that allows for many types of operation on a DAG, starting services, and supporting development and testing.

```
usage: airflow [-h]
               {resetdb,render,variables,connections,create_user,pause,task_failed_
↪deps,version,trigger_dag,initdb,test,unpause,dag_state,run,list_tasks,backfill,list_
↪dags,kerberos,worker,webserver,flower,scheduler,task_state,pool,serve_logs,clear,
↪upgradedb,delete_dag}
               ...
```

3.10.1 Positional Arguments

subcommand Possible choices: resetdb, render, variables, connections, create_user, pause, task_failed_deps, version, trigger_dag, initdb, test, unpause, dag_state, run,

list_tasks, backfill, list_dags, kerberos, worker, webserver, flower, scheduler, task_state, pool, serve_logs, clear, upgradedb, delete_dag

sub-command help

3.10.2 Sub-commands:

3.10.2.1 resetdb

Burn down and rebuild the metadata database

```
airflow resetdb [-h] [-y]
```

Named Arguments

-y, --yes Do not prompt to confirm reset. Use with care!

Default: False

3.10.2.2 render

Render a task instance's template(s)

```
airflow render [-h] [-sd SUBDIR] dag_id task_id execution_date
```

Positional Arguments

dag_id The id of the dag

task_id The id of the task

execution_date The execution date of the DAG

Named Arguments

-sd, --subdir File location or directory from which to look for the dag

Default: /home/docs/airflow/dags

3.10.2.3 variables

CRUD operations on variables

```
airflow variables [-h] [-s KEY VAL] [-g KEY] [-j] [-d VAL] [-i FILEPATH]
                  [-e FILEPATH] [-x KEY]
```

Named Arguments

-s, --set	Set a variable
-g, --get	Get value of a variable
-j, --json	Deserialize JSON variable
	Default: False
-d, --default	Default value returned if variable does not exist
-i, --import	Import variables from JSON file
-e, --export	Export variables to JSON file
-x, --delete	Delete a variable

3.10.2.4 connections

List/Add/Delete connections

```
airflow connections [-h] [-l] [-a] [-d] [--conn_id CONN_ID]
                    [--conn_uri CONN_URI] [--conn_extra CONN_EXTRA]
                    [--conn_type CONN_TYPE] [--conn_host CONN_HOST]
                    [--conn_login CONN_LOGIN] [--conn_password CONN_PASSWORD]
                    [--conn_schema CONN_SCHEMA] [--conn_port CONN_PORT]
```

Named Arguments

-l, --list	List all connections
	Default: False
-a, --add	Add a connection
	Default: False
-d, --delete	Delete a connection
	Default: False
--conn_id	Connection id, required to add/delete a connection
--conn_uri	Connection URI, required to add a connection without conn_type
--conn_extra	Connection *Extra* field, optional when adding a connection
--conn_type	Connection type, required to add a connection without conn_uri
--conn_host	Connection host, optional when adding a connection
--conn_login	Connection login, optional when adding a connection
--conn_password	Connection password, optional when adding a connection
--conn_schema	Connection schema, optional when adding a connection
--conn_port	Connection port, optional when adding a connection

3.10.2.5 create_user

Create an admin account

```
airflow create_user [-h] [-r ROLE] [-u USERNAME] [-e EMAIL] [-f FIRSTNAME]
                    [-l LASTNAME] [-p PASSWORD] [--use_random_password]
```

Named Arguments

-r, --role Role of the user. Existing roles include Admin, User, Op, Viewer, and Public

-u, --username Username of the user

-e, --email Email of the user

-f, --firstname First name of the user

-l, --lastname Last name of the user

-p, --password Password of the user

--use_random_password Do not prompt for password. Use random string instead

Default: False

3.10.2.6 pause

Pause a DAG

```
airflow pause [-h] [-sd SUBDIR] dag_id
```

Positional Arguments

dag_id The id of the dag

Named Arguments

-sd, --subdir File location or directory from which to look for the dag

Default: /home/docs/airflow/dags

3.10.2.7 task_failed_deps

Returns the unmet dependencies for a task instance from the perspective of the scheduler. In other words, why a task instance doesn't get scheduled and then queued by the scheduler, and then run by an executor).

```
airflow task_failed_deps [-h] [-sd SUBDIR] dag_id task_id execution_date
```

Positional Arguments

dag_id	The id of the dag
task_id	The id of the task
execution_date	The execution date of the DAG

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag Default: /home/docs/airflow/dags

3.10.2.8 version

Show the version

```
airflow version [-h]
```

3.10.2.9 trigger_dag

Trigger a DAG run

```
airflow trigger_dag [-h] [-sd SUBDIR] [-r RUN_ID] [-c CONF] [-e EXEC_DATE]
                    dag_id
```

Positional Arguments

dag_id	The id of the dag

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag Default: /home/docs/airflow/dags
-r, --run_id	Helps to identify this run
-c, --conf	JSON string that gets pickled into the DagRun's conf attribute
-e, --exec_date	The execution date of the DAG

3.10.2.10 initdb

Initialize the metadata database

```
airflow initdb [-h]
```

3.10.2.11 test

Test a task instance. This will run a task without checking for dependencies or recording its state in the database.

```
airflow test [-h] [-sd SUBDIR] [-dr] [-tp TASK_PARAMS]
             dag_id task_id execution_date
```

Positional Arguments

dag_id	The id of the dag
task_id	The id of the task
execution_date	The execution date of the DAG

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag
	Default: /home/docs/airflow/dags
-dr, --dry_run	Perform a dry run
	Default: False
-tp, --task_params	Sends a JSON params dict to the task

3.10.2.12 unpause

Resume a paused DAG

```
airflow unpause [-h] [-sd SUBDIR] dag_id
```

Positional Arguments

dag_id	The id of the dag

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag
	Default: /home/docs/airflow/dags

3.10.2.13 dag_state

Get the status of a dag run

```
airflow dag_state [-h] [-sd SUBDIR] dag_id execution_date
```

Positional Arguments

dag_id The id of the dag

execution_date The execution date of the DAG

Named Arguments

-sd, --subdir File location or directory from which to look for the dag

Default: /home/docs/airflow/dags

3.10.2.14 run

Run a single task instance

```
airflow run [-h] [-sd SUBDIR] [-m] [-f] [--pool POOL] [--cfg_path CFG_PATH]
            [-l] [-A] [-i] [-I] [--ship_dag] [-p PICKLE] [-int]
            dag_id task_id execution_date
```

Positional Arguments

dag_id The id of the dag

task_id The id of the task

execution_date The execution date of the DAG

Named Arguments

-sd, --subdir File location or directory from which to look for the dag

Default: /home/docs/airflow/dags

-m, --mark_success Mark jobs as succeeded without running them

Default: False

-f, --force Ignore previous task instance state, rerun regardless if task already succeeded/failed

Default: False

--pool Resource pool to use

--cfg_path Path to config file to use instead of airflow.cfg

-l, --local Run the task using the LocalExecutor

Default: False

-A, --ignore_all_dependencies Ignores all non-critical dependencies, including ignore_ti_state and ignore_task_deps

Default: False

-i, --ignore_dependencies Ignore task-specific dependencies, e.g. upstream, depends_on_past, and retry delay dependencies

Default: False

-I, --ignore_depends_on_past Ignore depends_on_past dependencies (but respect upstream dependencies)

Default: False

--ship_dag Pickles (serializes) the DAG and ships it to the worker

Default: False

-p, --pickle Serialized pickle object of the entire dag (used internally)

-int, --interactive Do not capture standard output and error streams (useful for interactive debugging)

Default: False

3.10.2.15 list_tasks

List the tasks within a DAG

```
airflow list_tasks [-h] [-t] [-sd SUBDIR] dag_id
```

Positional Arguments

dag_id The id of the dag

Named Arguments

-t, --tree Tree view

Default: False

-sd, --subdir File location or directory from which to look for the dag

Default: /home/docs/airflow/dags

3.10.2.16 backfill

Run subsections of a DAG for a specified date range. If reset_dag_run option is used, backfill will first prompt users whether airflow should clear all the previous dag_run and task_instances within the backfill date range.If rerun_failed_tasks is used, backfill will auto re-run the previous failed task instances within the backfill date range.

```
airflow backfill [-h] [-t TASK_REGEX] [-s START_DATE] [-e END_DATE] [-m] [-l]
                 [-x] [-i] [-I] [-sd SUBDIR] [--pool POOL]
                 [--delay_on_limit DELAY_ON_LIMIT] [-dr] [-v] [-c CONF]
                 [--reset_dagruns] [--rerun_failed_tasks]
                 dag_id
```

Positional Arguments

dag_id The id of the dag

Named Arguments

-t, --task_regex The regex to filter specific task_ids to backfill (optional)

-s, --start_date Override start_date YYYY-MM-DD

-e, --end_date Override end_date YYYY-MM-DD

-m, --mark_success Mark jobs as succeeded without running them

Default: False

-l, --local Run the task using the LocalExecutor

Default: False

-x, --donot_pickle Do not attempt to pickle the DAG object to send over to the workers, just tell the workers to run their version of the code.

Default: False

-i, --ignore_dependencies Skip upstream tasks, run only the tasks matching the regexp. Only works in conjunction with task_regex

Default: False

-I, --ignore_first_depends_on_past Ignores depends_on_past dependencies for the first set of tasks only (subsequent executions in the backfill DO respect depends_on_past).

Default: False

-sd, --subdir File location or directory from which to look for the dag

Default: /home/docs/airflow/dags

--pool Resource pool to use

--delay_on_limit Amount of time in seconds to wait when the limit on maximum active dag runs (max_active_runs) has been reached before trying to execute a dag run again.

Default: 1.0

-dr, --dry_run Perform a dry run

Default: False

-v, --verbose Make logging output more verbose

Default: False

-c, --conf JSON string that gets pickled into the DagRun's conf attribute

--reset_dagruns if set, the backfill will delete existing backfill-related DAG runs and start anew with fresh, running DAG runs

Default: False

--rerun_failed_tasks if set, the backfill will auto-rerun all the failed tasks for the backfill date range instead of throwing exceptions

Default: False

3.10.2.17 list_dags

List all the DAGs

```
airflow list_dags [-h] [-sd SUBDIR] [-r]
```

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag
	Default: /home/docs/airflow/dags
-r, --report	Show DagBag loading report
	Default: False

3.10.2.18 kerberos

Start a kerberos ticket renewer

```
airflow kerberos [-h] [-kt [KEYTAB]] [--pid [PID]] [-D] [--stdout STDOUT]
                 [--stderr STDERR] [-l LOG_FILE]
                 [principal]
```

Positional Arguments

principal	kerberos principal
	Default: airflow

Named Arguments

-kt, --keytab	keytab
	Default: airflow.keytab
--pid	PID file location
-D, --daemon	Daemonize instead of running in the foreground
	Default: False
--stdout	Redirect stdout to this file
--stderr	Redirect stderr to this file
-l, --log-file	Location of the log file

3.10.2.19 worker

Start a Celery worker node

```
airflow worker [-h] [-p] [-q QUEUES] [-c CONCURRENCY] [-cn CELERY_HOSTNAME]
               [--pid [PID]] [-D] [--stdout STDOUT] [--stderr STDERR]
               [-l LOG_FILE]
```

Named Arguments

-p, --do_pickle Attempt to pickle the DAG object to send over to the workers, instead of letting workers run their version of the code.

Default: False

-q, --queues Comma delimited list of queues to serve

Default: default

-c, --concurrency The number of worker processes

Default: 16

-cn, --celery_hostname Set the hostname of celery worker if you have multiple workers on a single machine.

--pid PID file location

-D, --daemon Daemonize instead of running in the foreground

Default: False

--stdout Redirect stdout to this file

--stderr Redirect stderr to this file

-l, --log-file Location of the log file

3.10.2.20 webserver

Start a Airflow webserver instance

```
airflow webserver [-h] [-p PORT] [-w WORKERS]
                  [-k {sync,eventlet,gevent,tornado}] [-t WORKER_TIMEOUT]
                  [-hn HOSTNAME] [--pid [PID]] [-D] [--stdout STDOUT]
                  [--stderr STDERR] [-A ACCESS_LOGFILE] [-E ERROR_LOGFILE]
                  [-l LOG_FILE] [--ssl_cert SSL_CERT] [--ssl_key SSL_KEY] [-d]
```

Named Arguments

-p, --port The port on which to run the server

Default: 8080

-w, --workers Number of workers to run the webserver on

Default: 4

-k, --workerclass Possible choices: sync, eventlet, gevent, tornado

The worker class to use for Gunicorn

Default: sync

-t, --worker_timeout The timeout for waiting on webserver workers

Default: 120

-hn, --hostname Set the hostname on which to run the web server

Default: 0.0.0.0

--pid	PID file location
-D, --daemon	Daemonize instead of running in the foreground Default: False
--stdout	Redirect stdout to this file
--stderr	Redirect stderr to this file
-A, --access_logfile	The logfile to store the webserver access log. Use '-' to print to stderr. Default: -
-E, --error_logfile	The logfile to store the webserver error log. Use '-' to print to stderr. Default: -
-l, --log-file	Location of the log file
--ssl_cert	Path to the SSL certificate for the webserver
--ssl_key	Path to the key to use with the SSL certificate
-d, --debug	Use the server that ships with Flask in debug mode Default: False

3.10.2.21 flower

Start a Celery Flower

```
airflow flower [-h] [-hn HOSTNAME] [-p PORT] [-fc FLOWER_CONF] [-u URL_PREFIX]
               [-a BROKER_API] [--pid [PID]] [-D] [--stdout STDOUT]
               [--stderr STDERR] [-l LOG_FILE]
```

Named Arguments

-hn, --hostname	Set the hostname on which to run the server Default: 0.0.0.0
-p, --port	The port on which to run the server Default: 5555
-fc, --flower_conf	Configuration file for flower
-u, --url_prefix	URL prefix for Flower
-a, --broker_api	Broker api
--pid	PID file location
-D, --daemon	Daemonize instead of running in the foreground Default: False
--stdout	Redirect stdout to this file
--stderr	Redirect stderr to this file
-l, --log-file	Location of the log file

3.10.2.22 scheduler

Start a scheduler instance

```
airflow scheduler [-h] [-d DAG_ID] [-sd SUBDIR] [-r RUN_DURATION]
                  [-n NUM_RUNS] [-p] [--pid [PID]] [-D] [--stdout STDOUT]
                  [--stderr STDERR] [-l LOG_FILE]
```

Named Arguments

-d, --dag_id	The id of the dag to run
-sd, --subdir	File location or directory from which to look for the dag
	Default: /home/docs/airflow/dags
-r, --run-duration	Set number of seconds to execute before exiting
-n, --num_runs	Set the number of runs to execute before exiting
	Default: -1
-p, --do_pickle	Attempt to pickle the DAG object to send over to the workers, instead of letting workers run their version of the code.
	Default: False
--pid	PID file location
-D, --daemon	Daemonize instead of running in the foreground
	Default: False
--stdout	Redirect stdout to this file
--stderr	Redirect stderr to this file
-l, --log-file	Location of the log file

3.10.2.23 task_state

Get the status of a task instance

```
airflow task_state [-h] [-sd SUBDIR] dag_id task_id execution_date
```

Positional Arguments

dag_id	The id of the dag
task_id	The id of the task
execution_date	The execution date of the DAG

Named Arguments

-sd, --subdir	File location or directory from which to look for the dag
	Default: /home/docs/airflow/dags

3.10.2.24 pool

CRUD operations on pools

```
airflow pool [-h] [-s NAME SLOT_COUNT POOL_DESCRIPTION] [-g NAME] [-x NAME]
```

Named Arguments

-s, --set	Set pool slot count and description, respectively
-g, --get	Get pool info
-x, --delete	Delete a pool

3.10.2.25 serve_logs

Serve logs generate by worker

```
airflow serve_logs [-h]
```

3.10.2.26 clear

Clear a set of task instance, as if they never ran

```
airflow clear [-h] [-t TASK_REGEX] [-s START_DATE] [-e END_DATE] [-sd SUBDIR]
              [-u] [-d] [-c] [-f] [-r] [-x] [-dx]
              dag_id
```

Positional Arguments

dag_id	The id of the dag

Named Arguments

-t, --task_regex	The regex to filter specific task_ids to backfill (optional)
-s, --start_date	Override start_date YYYY-MM-DD
-e, --end_date	Override end_date YYYY-MM-DD
-sd, --subdir	File location or directory from which to look for the dag
	Default: /home/docs/airflow/dags
-u, --upstream	Include upstream tasks
	Default: False
-d, --downstream	Include downstream tasks
	Default: False
-c, --no_confirm	Do not request confirmation
	Default: False

-f, --only_failed Only failed jobs

Default: False

-r, --only_running Only running jobs

Default: False

-x, --exclude_subdags Exclude subdags

Default: False

-dx, --dag_regex Search dag_id as regex instead of exact string

Default: False

3.10.2.27 upgradedb

Upgrade the metadata database to latest version

```
airflow upgradedb [-h]
```

3.10.2.28 delete_dag

Delete all DB records related to the specified DAG

```
airflow delete_dag [-h] [-y] dag_id
```

Positional Arguments

dag_id The id of the dag

Named Arguments

-y, --yes Do not prompt to confirm reset. Use with care!

Default: False

3.11 Scheduling & Triggers

The Airflow scheduler monitors all tasks and all DAGs, and triggers the task instances whose dependencies have been met. Behind the scenes, it monitors and stays in sync with a folder for all DAG objects it may contain, and periodically (every minute or so) inspects active tasks to see whether they can be triggered.

The Airflow scheduler is designed to run as a persistent service in an Airflow production environment. To kick it off, all you need to do is execute `airflow scheduler`. It will use the configuration specified in `airflow.cfg`.

Note that if you run a DAG on a `schedule_interval` of one day, the run stamped `2016-01-01` will be trigger soon after `2016-01-01T23:59`. In other words, the job instance is started once the period it covers has ended.

Let's Repeat That The scheduler runs your job one `schedule_interval` AFTER the start date, at the END of the period.

The scheduler starts an instance of the executor specified in the your `airflow.cfg`. If it happens to be the `LocalExecutor`, tasks will be executed as subprocesses; in the case of `CeleryExecutor` and `MesosExecutor`, tasks are executed remotely.

To start a scheduler, simply run the command:

```
airflow scheduler
```

3.11.1 DAG Runs

A DAG Run is an object representing an instantiation of the DAG in time.

Each DAG may or may not have a schedule, which informs how `DAG Runs` are created. `schedule_interval` is defined as a DAG arguments, and receives preferably a cron expression as a `str`, or a `datetime.timedelta` object. Alternatively, you can also use one of these cron "preset":

preset	meaning	cron
`None`	Don't schedule, use for exclusively "externally triggered" DAGs	
`@once`	Schedule once and only once	
`@hourly`	Run once an hour at the beginning of the hour	`0 * * * *`
`@daily`	Run once a day at midnight	`0 0 * * *`
`@weekly`	Run once a week at midnight on Sunday morning	`0 0 * * 0`
`@monthly`	Run once a month at midnight of the first day of the month	`0 0 1 * *`
`@yearly`	Run once a year at midnight of January 1	`0 0 1 1 *`

Your DAG will be instantiated for each schedule, while creating a `DAG Run` entry for each schedule.

DAG runs have a state associated to them (running, failed, success) and informs the scheduler on which set of schedules should be evaluated for task submissions. Without the metadata at the DAG run level, the Airflow scheduler would have much more work to do in order to figure out what tasks should be triggered and come to a crawl. It might also create undesired processing when changing the shape of your DAG, by say adding in new tasks.

3.11.2 Backfill and Catchup

An Airflow DAG with a `start_date`, possibly an `end_date`, and a `schedule_interval` defines a series of intervals which the scheduler turn into individual Dag Runs and execute. A key capability of Airflow is that these DAG Runs are atomic, idempotent items, and the scheduler, by default, will examine the lifetime of the DAG (from start to end/now, one interval at a time) and kick off a DAG Run for any interval that has not been run (or has been cleared). This concept is called Catchup.

If your DAG is written to handle its own catchup (IE not limited to the interval, but instead to "Now" for instance.), then you will want to turn catchup off (Either on the DAG itself with `dag.catchup = False`) or by default at the configuration file level with `catchup_by_default = False`. What this will do, is to instruct the scheduler to only create a DAG Run for the most current instance of the DAG interval series.

```
"""
Code that goes along with the Airflow tutorial located at:
https://github.com/airbnb/airflow/blob/master/airflow/example_dags/tutorial.py
"""
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta
```

(continues on next page)

(continued from previous page)

```
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 12, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    'schedule_interval': '@hourly',
}

dag = DAG('tutorial', catchup=False, default_args=default_args)
```

In the example above, if the DAG is picked up by the scheduler daemon on 2016-01-02 at 6 AM, (or from the command line), a single DAG Run will be created, with an `execution_date` of 2016-01-01, and the next one will be created just after midnight on the morning of 2016-01-03 with an execution date of 2016-01-02.

If the `dag.catchup` value had been True instead, the scheduler would have created a DAG Run for each completed interval between 2015-12-01 and 2016-01-02 (but not yet one for 2016-01-02, as that interval hasn't completed) and the scheduler will execute them sequentially. This behavior is great for atomic datasets that can easily be split into periods. Turning catchup off is great if your DAG Runs perform backfill internally.

3.11.3 External Triggers

Note that `DAG Runs` can also be created manually through the CLI while running an `airflow trigger_dag` command, where you can define a specific `run_id`. The `DAG Runs` created externally to the scheduler get associated to the trigger's timestamp, and will be displayed in the UI alongside scheduled `DAG runs`.

3.11.4 To Keep in Mind

- The first `DAG Run` is created based on the minimum `start_date` for the tasks in your DAG.
- Subsequent `DAG Runs` are created by the scheduler process, based on your DAG's `schedule_interval`, sequentially.
- When clearing a set of tasks' state in hope of getting them to re-run, it is important to keep in mind the `DAG Run`'s state too as it defines whether the scheduler should look into triggering tasks for that run.

Here are some of the ways you can **unblock tasks**:

- From the UI, you can **clear** (as in delete the status of) individual task instances from the task instances dialog, while defining whether you want to includes the past/future and the upstream/downstream dependencies. Note that a confirmation window comes next and allows you to see the set you are about to clear. You can also clear all task instances associated with the dag.
- The CLI command `airflow clear -h` has lots of options when it comes to clearing task instance states, including specifying date ranges, targeting task_ids by specifying a regular expression, flags for including upstream and downstream relatives, and targeting task instances in specific states (`failed`, or `success`)
- Clearing a task instance will no longer delete the task instance record. Instead it updates max_tries and set the current task instance state to be None.
- Marking task instances as successful can be done through the UI. This is mostly to fix false negatives, or for instance when the fix has been applied outside of Airflow.

- The `airflow backfill` CLI subcommand has a flag to `--mark_success` and allows selecting subsections of the DAG as well as specifying date ranges.

3.12 Plugins

Airflow has a simple plugin manager built-in that can integrate external features to its core by simply dropping files in your `$AIRFLOW_HOME/plugins` folder.

The python modules in the `plugins` folder get imported, and **hooks**, **operators**, **sensors**, **macros**, **executors** and web **views** get integrated to Airflow's main collections and become available for use.

3.12.1 What for?

Airflow offers a generic toolbox for working with data. Different organizations have different stacks and different needs. Using Airflow plugins can be a way for companies to customize their Airflow installation to reflect their ecosystem.

Plugins can be used as an easy way to write, share and activate new sets of features.

There's also a need for a set of more complex applications to interact with different flavors of data and metadata.

Examples:

- A set of tools to parse Hive logs and expose Hive metadata (CPU /IO / phases/ skew /...)
- An anomaly detection framework, allowing people to collect metrics, set thresholds and alerts
- An auditing tool, helping understand who accesses what
- A config-driven SLA monitoring tool, allowing you to set monitored tables and at what time they should land, alert people, and expose visualizations of outages
- ...

3.12.2 Why build on top of Airflow?

Airflow has many components that can be reused when building an application:

- A web server you can use to render your views
- A metadata database to store your models
- Access to your databases, and knowledge of how to connect to them
- An array of workers that your application can push workload to
- Airflow is deployed, you can just piggy back on its deployment logistics
- Basic charting capabilities, underlying libraries and abstractions

3.12.3 Interface

To create a plugin you will need to derive the `airflow.plugins_manager.AirflowPlugin` class and reference the objects you want to plug into Airflow. Here's what the class you need to derive looks like:

```
class AirflowPlugin(object):
    # The name of your plugin (str)
    name = None
    # A list of class(es) derived from BaseOperator
    operators = []
    # A list of class(es) derived from BaseSensorOperator
    sensors = []
    # A list of class(es) derived from BaseHook
    hooks = []
    # A list of class(es) derived from BaseExecutor
    executors = []
    # A list of references to inject into the macros namespace
    macros = []
    # A list of objects created from a class derived
    # from flask_admin.BaseView
    admin_views = []
    # A list of Blueprint object created from flask.Blueprint
    flask_blueprints = []
    # A list of menu links (flask_admin.base.MenuLink)
    menu_links = []
```

3.12.4 Example

The code below defines a plugin that injects a set of dummy object definitions in Airflow.

```
# This is the class you derive to create a plugin
from airflow.plugins_manager import AirflowPlugin

from flask import Blueprint
from flask_admin import BaseView, expose
from flask_admin.base import MenuLink

# Importing base classes that we need to derive
from airflow.hooks.base_hook import BaseHook
from airflow.models import BaseOperator
from airflow.sensors.base_sensor_operator import BaseSensorOperator
from airflow.executors.base_executor import BaseExecutor

# Will show up under airflow.hooks.test_plugin.PluginHook
class PluginHook(BaseHook):
    pass

# Will show up under airflow.operators.test_plugin.PluginOperator
class PluginOperator(BaseOperator):
    pass

# Will show up under airflow.sensors.test_plugin.PluginSensorOperator
class PluginSensorOperator(BaseSensorOperator):
    pass

# Will show up under airflow.executors.test_plugin.PluginExecutor
class PluginExecutor(BaseExecutor):
    pass

# Will show up under airflow.macros.test_plugin.plugin_macro
def plugin_macro():
```

(continues on next page)

(continued from previous page)

```
    pass

# Creating a flask admin BaseView
class TestView(BaseView):
    @expose('/')
    def test(self):
        # in this example, put your test_plugin/test.html template at airflow/plugins/
↪templates/test_plugin/test.html
        return self.render("test_plugin/test.html", content="Hello galaxy!")
v = TestView(category="Test Plugin", name="Test View")

# Creating a flask blueprint to integrate the templates and static folder
bp = Blueprint(
    "test_plugin", __name__,
    template_folder='templates', # registers airflow/plugins/templates as a Jinja␣
↪template folder
    static_folder='static',
    static_url_path='/static/test_plugin')

ml = MenuLink(
    category='Test Plugin',
    name='Test Menu Link',
    url='https://airflow.incubator.apache.org/')

# Defining the plugin class
class AirflowTestPlugin(AirflowPlugin):
    name = "test_plugin"
    operators = [PluginOperator]
    sensors = [PluginSensorOperator]
    hooks = [PluginHook]
    executors = [PluginExecutor]
    macros = [plugin_macro]
    admin_views = [v]
    flask_blueprints = [bp]
    menu_links = [ml]
```

3.13 Security

By default, all gates are opened. An easy way to restrict access to the web application is to do it at the network level, or by using SSH tunnels.

It is however possible to switch on authentication by either using one of the supplied backends or creating your own.

Be sure to checkout *Experimental Rest API* for securing the API.

3.13.1 Web Authentication

3.13.1.1 Password

One of the simplest mechanisms for authentication is requiring users to specify a password before logging in. Password authentication requires the used of the `password` subpackage in your requirements file. Password hashing uses bcrypt before storing passwords.

```
[webserver]
authenticate = True
auth_backend = airflow.contrib.auth.backends.password_auth
```

When password auth is enabled, an initial user credential will need to be created before anyone can login. An initial user was not created in the migrations for this authentication backend to prevent default Airflow installations from attack. Creating a new user has to be done via a Python REPL on the same machine Airflow is installed.

```
# navigate to the airflow installation directory
$ cd ~/airflow
$ python
Python 2.7.9 (default, Feb 10 2015, 03:28:08)
Type "help", "copyright", "credits" or "license" for more information.
>>> import airflow
>>> from airflow import models, settings
>>> from airflow.contrib.auth.backends.password_auth import PasswordUser
>>> user = PasswordUser(models.User())
>>> user.username = 'new_user_name'
>>> user.email = 'new_user_email@example.com'
>>> user.password = 'set_the_password'
>>> session = settings.Session()
>>> session.add(user)
>>> session.commit()
>>> session.close()
>>> exit()
```

3.13.1.2 LDAP

To turn on LDAP authentication configure your `airflow.cfg` as follows. Please note that the example uses an encrypted connection to the ldap server as you probably do not want passwords be readable on the network level. It is however possible to configure without encryption if you really want to.

Additionally, if you are using Active Directory, and are not explicitly specifying an OU that your users are in, you will need to change `search_scope` to "SUBTREE".

Valid search_scope options can be found in the ldap3 Documentation

```
[webserver]
authenticate = True
auth_backend = airflow.contrib.auth.backends.ldap_auth

[ldap]
# set a connection without encryption: uri = ldap://<your.ldap.server>:<port>
uri = ldaps://<your.ldap.server>:<port>
user_filter = objectClass=*
# in case of Active Directory you would use: user_name_attr = sAMAccountName
user_name_attr = uid
# group_member_attr should be set accordingly with *_filter
# eg :
#     group_member_attr = groupMembership
#     superuser_filter = groupMembership=CN=airflow-super-users...
group_member_attr = memberOf
superuser_filter = memberOf=CN=airflow-super-users,OU=Groups,OU=RWC,OU=US,OU=NORAM,
↪DC=example,DC=com
data_profiler_filter = memberOf=CN=airflow-data-profilers,OU=Groups,OU=RWC,OU=US,
↪OU=NORAM,DC=example,DC=com
```

(continues on next page)

(continued from previous page)

```
bind_user = cn=Manager,dc=example,dc=com
bind_password = insecure
basedn = dc=example,dc=com
cacert = /etc/ca/ldap_ca.crt
# Set search_scope to one of them:  BASE, LEVEL , SUBTREE
# Set search_scope to SUBTREE if using Active Directory, and not specifying an␣
↪Organizational Unit
search_scope = LEVEL
```

The superuser_filter and data_profiler_filter are optional. If defined, these configurations allow you to specify LDAP groups that users must belong to in order to have superuser (admin) and data-profiler permissions. If undefined, all users will be superusers and data profilers.

3.13.1.3 Roll your own

Airflow uses `flask_login` and exposes a set of hooks in the `airflow.default_login` module. You can alter the content and make it part of the `PYTHONPATH` and configure it as a backend in `airflow.cfg`.

```
[webserver]
authenticate = True
auth_backend = mypackage.auth
```

3.13.2 Multi-tenancy

You can filter the list of dags in webserver by owner name when authentication is turned on by setting `webserver:filter_by_owner` in your config. With this, a user will see only the dags which it is owner of, unless it is a superuser.

```
[webserver]
filter_by_owner = True
```

3.13.3 Kerberos

Airflow has initial support for Kerberos. This means that airflow can renew kerberos tickets for itself and store it in the ticket cache. The hooks and dags can make use of ticket to authenticate against kerberized services.

3.13.3.1 Limitations

Please note that at this time, not all hooks have been adjusted to make use of this functionality. Also it does not integrate kerberos into the web interface and you will have to rely on network level security for now to make sure your service remains secure.

Celery integration has not been tried and tested yet. However, if you generate a key tab for every host and launch a ticket renewer next to every worker it will most likely work.

3.13.3.2 Enabling kerberos

Airflow

To enable kerberos you will need to generate a (service) key tab.

```
# in the kadmin.local or kadmin shell, create the airflow principal
kadmin:  addprinc -randkey airflow/fully.qualified.domain.name@YOUR-REALM.COM

# Create the airflow keytab file that will contain the airflow principal
kadmin:  xst -norandkey -k airflow.keytab airflow/fully.qualified.domain.name
```

Now store this file in a location where the airflow user can read it (chmod 600). And then add the following to your `airflow.cfg`

```
[core]
security = kerberos

[kerberos]
keytab = /etc/airflow/airflow.keytab
reinit_frequency = 3600
principal = airflow
```

Launch the ticket renewer by

```
# run ticket renewer
airflow kerberos
```

Hadoop

If want to use impersonation this needs to be enabled in `core-site.xml` of your hadoop config.

```
<property>
  <name>hadoop.proxyuser.airflow.groups</name>
  <value>*</value>
</property>

<property>
  <name>hadoop.proxyuser.airflow.users</name>
  <value>*</value>
</property>

<property>
  <name>hadoop.proxyuser.airflow.hosts</name>
  <value>*</value>
</property>
```

Of course if you need to tighten your security replace the asterisk with something more appropriate.

3.13.3.3 Using kerberos authentication

The hive hook has been updated to take advantage of kerberos authentication. To allow your DAGs to use it, simply update the connection details with, for example:

```
{ "use_beeline": true, "principal": "hive/_HOST@EXAMPLE.COM"}
```

Adjust the principal to your settings. The _HOST part will be replaced by the fully qualified domain name of the server.

You can specify if you would like to use the dag owner as the user for the connection or the user specified in the login section of the connection. For the login user, specify the following as extra:

```
{ "use_beeline": true, "principal": "hive/_HOST@EXAMPLE.COM", "proxy_user": "login"}
```

For the DAG owner use:

```
{ "use_beeline": true, "principal": "hive/_HOST@EXAMPLE.COM", "proxy_user": "owner"}
```

and in your DAG, when initializing the HiveOperator, specify:

```
run_as_owner=True
```

3.13.4 OAuth Authentication

3.13.4.1 GitHub Enterprise (GHE) Authentication

The GitHub Enterprise authentication backend can be used to authenticate users against an installation of GitHub Enterprise using OAuth2. You can optionally specify a team whitelist (composed of slug cased team names) to restrict login to only members of those teams.

```
[webserver]
authenticate = True
auth_backend = airflow.contrib.auth.backends.github_enterprise_auth

[github_enterprise]
host = github.example.com
client_id = oauth_key_from_github_enterprise
client_secret = oauth_secret_from_github_enterprise
oauth_callback_route = /example/ghe_oauth/callback
allowed_teams = 1, 345, 23
```

Note: If you do not specify a team whitelist, anyone with a valid account on your GHE installation will be able to login to Airflow.

Setting up GHE Authentication

An application must be setup in GHE before you can use the GHE authentication backend. In order to setup an application:

1. Navigate to your GHE profile
2. Select 'Applications' from the left hand nav
3. Select the 'Developer Applications' tab
4. Click 'Register new application'
5. Fill in the required information (the 'Authorization callback URL' must be fully qualified e.g. http://airflow.example.com/example/ghe_oauth/callback)
6. Click 'Register application'
7. Copy 'Client ID', 'Client Secret', and your callback route to your airflow.cfg according to the above example

Using GHE Authentication with github.com

It is possible to use GHE authentication with github.com:

1. Create an Oauth App
2. Copy 'Client ID', 'Client Secret' to your airflow.cfg according to the above example
3. Set `host = github.com` and `oauth_callback_route = /oauth/callback` in airflow.cfg

3.13.4.2 Google Authentication

The Google authentication backend can be used to authenticate users against Google using OAuth2. You must specify the domains to restrict login, separated with a comma, to only members of those domains.

```
[webserver]
authenticate = True
auth_backend = airflow.contrib.auth.backends.google_auth

[google]
client_id = google_client_id
client_secret = google_client_secret
oauth_callback_route = /oauth2callback
domain = "example1.com,example2.com"
```

Setting up Google Authentication

An application must be setup in the Google API Console before you can use the Google authentication backend. In order to setup an application:

1. Navigate to https://console.developers.google.com/apis/
2. Select 'Credentials' from the left hand nav
3. Click 'Create credentials' and choose 'OAuth client ID'
4. Choose 'Web application'
5. Fill in the required information (the 'Authorized redirect URIs' must be fully qualified e.g. http://airflow.example.com/oauth2callback)
6. Click 'Create'
7. Copy 'Client ID', 'Client Secret', and your redirect URI to your airflow.cfg according to the above example

3.13.5 SSL

SSL can be enabled by providing a certificate and key. Once enabled, be sure to use "https://" in your browser.

```
[webserver]
web_server_ssl_cert = <path to cert>
web_server_ssl_key = <path to key>
```

Enabling SSL will not automatically change the web server port. If you want to use the standard port 443, you'll need to configure that too. Be aware that super user privileges (or cap_net_bind_service on Linux) are required to listen on port 443.

```
# Optionally, set the server to listen on the standard SSL port.
web_server_port = 443
base_url = http://<hostname or IP>:443
```

Enable CeleryExecutor with SSL. Ensure you properly generate client and server certs and keys.

```
[celery]
CELERY_SSL_ACTIVE = True
CELERY_SSL_KEY = <path to key>
CELERY_SSL_CERT = <path to cert>
CELERY_SSL_CACERT = <path to cacert>
```

3.13.6 Impersonation

Airflow has the ability to impersonate a unix user while running task instances based on the task's `run_as_user` parameter, which takes a user's name.

NOTE: For impersonations to work, Airflow must be run with *sudo* as subtasks are run with *sudo -u* and permissions of files are changed. Furthermore, the unix user needs to exist on the worker. Here is what a simple sudoers file entry could look like to achieve this, assuming as airflow is running as the *airflow* user. Note that this means that the airflow user must be trusted and treated the same way as the root user.

```
airflow ALL=(ALL) NOPASSWD: ALL
```

Subtasks with impersonation will still log to the same folder, except that the files they log to will have permissions changed such that only the unix user can write to it.

3.13.6.1 Default Impersonation

To prevent tasks that don't use impersonation to be run with *sudo* privileges, you can set the `core:default_impersonation` config which sets a default user impersonate if *run_as_user* is not set.

```
[core]
default_impersonation = airflow
```

3.14 Time zones

Support for time zones is enabled by default. Airflow stores datetime information in UTC internally and in the database. It allows you to run your DAGs with time zone dependent schedules. At the moment Airflow does not convert them to the end user's time zone in the user interface. There it will always be displayed in UTC. Also templates used in Operators are not converted. Time zone information is exposed and it is up to the writer of DAG what do with it.

This is handy if your users live in more than one time zone and you want to display datetime information according to each user's wall clock.

Even if you are running Airflow in only one time zone it is still good practice to store data in UTC in your database (also before Airflow became time zone aware this was also to recommended or even required setup). The main reason is Daylight Saving Time (DST). Many countries have a system of DST, where clocks are moved forward in spring and backward in autumn. If you're working in local time, you're likely to encounter errors twice a year, when the transitions happen. (The pendulum and pytz documentation discusses these issues in greater detail.) This probably

doesn't matter for a simple DAG, but it's a problem if you are in, for example, financial services where you have end of day deadlines to meet.

The time zone is set in *airflow.cfg*. By default it is set to utc, but you change it to use the system's settings or an arbitrary IANA time zone, e.g. *Europe/Amsterdam*. It is dependent on *pendulum*, which is more accurate than *pytz*. Pendulum is installed when you install Airflow.

Please note that the Web UI currently only runs in UTC.

3.14.1 Concepts

3.14.1.1 Naïve and aware datetime objects

Python's datetime.datetime objects have a tzinfo attribute that can be used to store time zone information, represented as an instance of a subclass of datetime.tzinfo. When this attribute is set and describes an offset, a datetime object is aware. Otherwise, it's naive.

You can use timezone.is_aware() and timezone.is_naive() to determine whether datetimes are aware or naive.

Because Airflow uses time-zone-aware datetime objects. If your code creates datetime objects they need to be aware too.

```
from airflow.utils import timezone

now = timezone.utcnow()
a_date = timezone.datetime(2017,1,1)
```

3.14.1.2 Interpretation of naive datetime objects

Although Airflow operates fully time zone aware, it still accepts naive date time objects for *start_dates* and *end_dates* in your DAG definitions. This is mostly in order to preserve backwards compatibility. In case a naive *start_date* or *end_date* is encountered the default time zone is applied. It is applied in such a way that it is assumed that the naive date time is already in the default time zone. In other words if you have a default time zone setting of *Europe/Amsterdam* and create a naive datetime *start_date* of *datetime(2017,1,1)* it is assumed to be a *start_date* of Jan 1, 2017 Amsterdam time.

```
default_args=dict(
    start_date=datetime(2016, 1, 1),
    owner='Airflow'
)

dag = DAG('my_dag', default_args=default_args)
op = DummyOperator(task_id='dummy', dag=dag)
print(op.owner) # Airflow
```

Unfortunately, during DST transitions, some datetimes don't exist or are ambiguous. In such situations, pendulum raises an exception. That's why you should always create aware datetime objects when time zone support is enabled.

In practice, this is rarely an issue. Airflow gives you aware datetime objects in the models and DAGs, and most often, new datetime objects are created from existing ones through timedelta arithmetic. The only datetime that's often created in application code is the current time, and timezone.utcnow() automatically does the right thing.

3.14.1.3 Default time zone

The default time zone is the time zone defined by the *default_timezone* setting under *[core]*. If you just installed Airflow it will be set to *utc*, which is recommended. You can also set it to *system* or an IANA time zone (e.g.'Europe/Amsterdam'). DAGs are also evaluated on Airflow workers, it is therefore important to make sure this setting is equal on all Airflow nodes.

```
[core]
default_timezone = utc
```

3.14.2 Time zone aware DAGs

Creating a time zone aware DAG is quite simple. Just make sure to supply a time zone aware *start_date*. It is recommended to use *pendulum* for this, but *pytz* (to be installed manually) can also be used for this.

```
import pendulum

local_tz = pendulum.timezone("Europe/Amsterdam")

default_args=dict(
    start_date=datetime(2016, 1, 1, tzinfo=local_tz),
    owner='Airflow'
)

dag = DAG('my_tz_dag', default_args=default_args)
op = DummyOperator(task_id='dummy', dag=dag)
print(dag.timezone) # <Timezone [Europe/Amsterdam]>
```

3.14.2.1 Templates

Airflow returns time zone aware datetimes in templates, but does not convert them to local time so they remain in UTC. It is left up to the DAG to handle this.

```
import pendulum

local_tz = pendulum.timezone("Europe/Amsterdam")
local_tz.convert(execution_date)
```

3.14.2.2 Cron schedules

In case you set a cron schedule, Airflow assumes you will always want to run at the exact same time. It will then ignore day light savings time. Thus, if you have a schedule that says run at end of interval every day at 08:00 GMT+1 it will always run end of interval 08:00 GMT+1, regardless if day light savings time is in place.

3.14.2.3 Time deltas

For schedules with time deltas Airflow assumes you always will want to run with the specified interval. So if you specify a timedelta(hours=2) you will always want to run to hours later. In this case day light savings time will be taken into account.

3.15 Experimental Rest API

Airflow exposes an experimental Rest API. It is available through the webserver. Endpoints are available at /api/experimental/. Please note that we expect the endpoint definitions to change.

3.15.1 Endpoints

This is a place holder until the swagger definitions are active

- /api/experimental/dags/<DAG_ID>/tasks/<TASK_ID> returns info for a task (GET).
- /api/experimental/dags/<DAG_ID>/dag_runs creates a dag_run for a given dag id (POST).

3.15.2 CLI

For some functions the cli can use the API. To configure the CLI to use the API when available configure as follows:

```
[cli]
api_client = airflow.api.client.json_client
endpoint_url = http://<WEBSERVER>:<PORT>
```

3.15.3 Authentication

Authentication for the API is handled separately to the Web Authentication. The default is to not require any authentication on the API – i.e. wide open by default. This is not recommended if your Airflow webserver is publicly accessible, and you should probably use the deny all backend:

```
[api]
auth_backend = airflow.api.auth.backend.deny_all
```

Two "real" methods for authentication are currently supported for the API.

To enabled Password authentication, set the following in the configuration:

```
[api]
auth_backend = airflow.contrib.auth.backends.password_auth
```

It's usage is similar to the Password Authentication used for the Web interface.

To enable Kerberos authentication, set the following in the configuration:

```
[api]
auth_backend = airflow.api.auth.backend.kerberos_auth

[kerberos]
keytab = <KEYTAB>
```

The Kerberos service is configured as `airflow/fully.qualified.domainname@REALM`. Make sure this principal exists in the keytab file.

3.16 Integration

3.16.1 Reverse Proxy

Airflow can be set up behind a reverse proxy, with the ability to set its endpoint with great flexibility.

For example, you can configure your reverse proxy to get:

```
https://lab.mycompany.com/myorg/airflow/
```

To do so, you need to set the following setting in your *airflow.cfg*:

```
base_url = http://my_host/myorg/airflow
```

Additionally if you use Celery Executor, you can get Flower in */myorg/flower* with:

```
flower_url_prefix = /myorg/flower
```

Your reverse proxy (ex: nginx) should be configured as follow:

- pass the url and http header as it for the Airflow webserver, without any rewrite, for example:

```
server {
  listen 80;
  server_name lab.mycompany.com;

  location /myorg/airflow/ {
      proxy_pass http://localhost:8080;
      proxy_set_header Host $host;
      proxy_redirect off;
      proxy_http_version 1.1;
      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection "upgrade";
  }
}
```

- rewrite the url for the flower endpoint:

```
server {
    listen 80;
    server_name lab.mycompany.com;

    location /myorg/flower/ {
        rewrite ^/myorg/flower/(.*)$ /$1 break;  # remove prefix from http header
        proxy_pass http://localhost:5555;
        proxy_set_header Host $host;
        proxy_redirect off;
        proxy_http_version 1.1;
```

(continues on next page)

(continued from previous page)

```
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }
}
```

3.16.2 Azure: Microsoft Azure

Airflow has limited support for Microsoft Azure: interfaces exist only for Azure Blob Storage and Azure Data Lake. Hook, Sensor and Operator for Blob Storage and Azure Data Lake Hook are in contrib section.

3.16.2.1 Azure Blob Storage

All classes communicate via the Window Azure Storage Blob protocol. Make sure that a Airflow connection of type *wasb* exists. Authorization can be done by supplying a login (=Storage account name) and password (=KEY), or login and SAS token in the extra field (see connection *wasb_default* for an example).

- *WasbBlobSensor*: Checks if a blob is present on Azure Blob storage.
- *WasbPrefixSensor*: Checks if blobs matching a prefix are present on Azure Blob storage.
- *FileToWasbOperator*: Uploads a local file to a container as a blob.
- *WasbHook*: Interface with Azure Blob Storage.

WasbBlobSensor

WasbPrefixSensor

FileToWasbOperator

WasbHook

3.16.2.2 Azure File Share

Cloud variant of a SMB file share. Make sure that a Airflow connection of type *wasb* exists. Authorization can be done by supplying a login (=Storage account name) and password (=Storage account key), or login and SAS token in the extra field (see connection *wasb_default* for an example).

AzureFileShareHook

3.16.2.3 Logging

Airflow can be configured to read and write task logs in Azure Blob Storage. See *Writing Logs to Azure Blob Storage*.

3.16.2.4 Azure Data Lake

AzureDataLakeHook communicates via a REST API compatible with WebHDFS. Make sure that a Airflow connection of type *azure_data_lake* exists. Authorization can be done by supplying a login (=Client ID), password (=Client Secret) and extra fields tenant (Tenant) and account_name (Account Name)

(see connection *azure_data_lake_default* for an example).

- *AzureDataLakeHook*: Interface with Azure Data Lake.

AzureDataLakeHook

3.16.3 AWS: Amazon Web Services

Airflow has extensive support for Amazon Web Services. But note that the Hooks, Sensors and Operators are in the contrib section.

3.16.3.1 AWS EMR

- *EmrAddStepsOperator* : Adds steps to an existing EMR JobFlow.
- *EmrCreateJobFlowOperator* : Creates an EMR JobFlow, reading the config from the EMR connection.
- *EmrTerminateJobFlowOperator* : Terminates an EMR JobFlow.
- *EmrHook* : Interact with AWS EMR.

EmrAddStepsOperator

class airflow.contrib.operators.emr_add_steps_operator.**EmrAddStepsOperator**(*job_flow_id*, *aws_conn_id='s3_defa*, *steps=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

An operator that adds steps to an existing EMR job_flow.

Parameters

- **job_flow_id** – id of the JobFlow to add steps to. (templated)
- **aws_conn_id** (*str*) – aws connection to uses
- **steps** (*list*) – boto3 style steps to be added to the jobflow. (templated)

EmrCreateJobFlowOperator

class airflow.contrib.operators.emr_create_job_flow_operator.**EmrCreateJobFlowOperator**(*aws_c*, *emr_c*, *job_flo*, **args*, ***kwa*

Bases: *airflow.models.BaseOperator*

Creates an EMR JobFlow, reading the config from the EMR connection. A dictionary of JobFlow overrides can be passed that override the config from the connection.

Parameters

- **aws_conn_id** (*str*) – aws connection to uses
- **emr_conn_id** (*str*) – emr connection to use

- **job_flow_overrides** – boto3 style arguments to override emr_connection extra. (templated)

EmrTerminateJobFlowOperator

class airflow.contrib.operators.emr_terminate_job_flow_operator.**EmrTerminateJobFlowOperator**

Bases: *airflow.models.BaseOperator*

Operator to terminate EMR JobFlows.

Parameters

- **job_flow_id** – id of the JobFlow to terminate. (templated)
- **aws_conn_id** (*str*) – aws connection to uses

EmrHook

class airflow.contrib.hooks.emr_hook.**EmrHook**(*emr_conn_id=None*, **args*, ***kwargs*)

Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS EMR. emr_conn_id is only neccessary for using the create_job_flow method.

create_job_flow(*job_flow_overrides*)

Creates a job flow using the config from the EMR connection. Keys of the json extra hash may have the arguments of the boto3 run_job_flow method. Overrides for this config may be passed as the job_flow_overrides.

3.16.3.2 AWS S3

- *S3Hook* : Interact with AWS S3.
- *S3FileTransformOperator* : Copies data from a source S3 location to a temporary location on the local filesystem.
- *S3ListOperator* : Lists the files matching a key prefix from a S3 location.
- *S3ToGoogleCloudStorageOperator* : Syncs an S3 location with a Google Cloud Storage bucket.
- *S3ToHiveTransfer* : Moves data from S3 to Hive. The operator downloads a file from S3, stores the file locally before loading it into a Hive table.

S3Hook

class airflow.hooks.S3_hook.**S3Hook**(*aws_conn_id='aws_default'*)

Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS S3, using the boto3 library.

check_for_bucket(*bucket_name*)

Check if bucket_name exists.

Parameters **bucket_name** (*str*) – the name of the bucket

check_for_key(*key*, *bucket_name=None*)
Checks if a key exists in a bucket

Parameters

- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which the file is stored

check_for_prefix(*bucket_name*, *prefix*, *delimiter*)
Checks that a prefix exists in a bucket

check_for_wildcard_key(*wildcard_key*, *bucket_name=None*, *delimiter=''*)
Checks that a key matching a wildcard expression exists in a bucket

get_bucket(*bucket_name*)
Returns a boto3.S3.Bucket object

Parameters **bucket_name** (*str*) – the name of the bucket

get_key(*key*, *bucket_name=None*)
Returns a boto3.s3.Object

Parameters

- **key** (*str*) – the path to the key
- **bucket_name** (*str*) – the name of the bucket

get_wildcard_key(*wildcard_key*, *bucket_name=None*, *delimiter=''*)
Returns a boto3.s3.Object object matching the wildcard expression

Parameters

- **wildcard_key** (*str*) – the path to the key
- **bucket_name** (*str*) – the name of the bucket

list_keys(*bucket_name*, *prefix=''*, *delimiter=''*, *page_size=None*, *max_items=None*)
Lists keys in a bucket under prefix and not containing delimiter

Parameters

- **bucket_name** (*str*) – the name of the bucket
- **prefix** (*str*) – a key prefix
- **delimiter** (*str*) – the delimiter marks key hierarchy.
- **page_size** (*int*) – pagination size
- **max_items** (*int*) – maximum items to return

list_prefixes(*bucket_name*, *prefix=''*, *delimiter=''*, *page_size=None*, *max_items=None*)
Lists prefixes in a bucket under prefix

Parameters

- **bucket_name** (*str*) – the name of the bucket
- **prefix** (*str*) – a key prefix
- **delimiter** (*str*) – the delimiter marks key hierarchy.
- **page_size** (*int*) – pagination size
- **max_items** (*int*) – maximum items to return

load_bytes(*bytes_data*, *key*, *bucket_name=None*, *replace=False*, *encrypt=False*)
Loads bytes to S3

This is provided as a convenience to drop a string in S3. It uses the boto infrastructure to ship a file to s3.

Parameters

- **bytes_data** (*bytes*) – bytes to set as content for the key.
- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which to store the file
- **replace** (*bool*) – A flag to decide whether or not to overwrite the key if it already exists
- **encrypt** (*bool*) – If True, the file will be encrypted on the server-side by S3 and will be stored in an encrypted form while at rest in S3.

load_file(*filename*, *key*, *bucket_name=None*, *replace=False*, *encrypt=False*)
Loads a local file to S3

Parameters

- **filename** (*str*) – name of the file to load.
- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which to store the file
- **replace** (*bool*) – A flag to decide whether or not to overwrite the key if it already exists. If replace is False and the key exists, an error will be raised.
- **encrypt** (*bool*) – If True, the file will be encrypted on the server-side by S3 and will be stored in an encrypted form while at rest in S3.

load_string(*string_data*, *key*, *bucket_name=None*, *replace=False*, *encrypt=False*, *encoding='utf-8'*)
Loads a string to S3

This is provided as a convenience to drop a string in S3. It uses the boto infrastructure to ship a file to s3.

Parameters

- **string_data** (*str*) – string to set as content for the key.
- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which to store the file
- **replace** (*bool*) – A flag to decide whether or not to overwrite the key if it already exists
- **encrypt** (*bool*) – If True, the file will be encrypted on the server-side by S3 and will be stored in an encrypted form while at rest in S3.

read_key(*key*, *bucket_name=None*)
Reads a key from S3

Parameters

- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which the file is stored

select_key(*key*, *bucket_name=None*, *expression='SELECT * FROM S3Object'*, *expression_type='SQL'*, *input_serialization={'CSV': {}}*, *output_serialization={'CSV': {}}*)
Reads a key with S3 Select.

Parameters

- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which the file is stored
- **expression** (*str*) – S3 Select expression
- **expression_type** (*str*) – S3 Select expression type
- **input_serialization** (*dict*) – S3 Select input data serialization format
- **output_serialization** (*dict*) – S3 Select output data serialization format

Returns retrieved subset of original data by S3 Select

Return type str

See also:

For more details about S3 Select parameters: http://boto3.readthedocs.io/en/latest/reference/services/s3.html#S3.Client.select_object_content

S3FileTransformOperator

class airflow.operators.s3_file_transform_operator.**S3FileTransformOperator**(*source_s3_key*, *dest_s3_key*, *transform_script=None*, *select_expression=None*, *source_aws_conn_id=*, *dest_aws_conn_id='av*, *replace=False*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Copies data from a source S3 location to a temporary location on the local filesystem. Runs a transformation on this file as specified by the transformation script and uploads the output to a destination S3 location.

The locations of the source and the destination files in the local filesystem is provided as an first and second arguments to the transformation script. The transformation script is expected to read the data from source, transform it and write the output to the local destination file. The operator then takes over control and uploads the local destination file to S3.

S3 Select is also available to filter the source contents. Users can omit the transformation script if S3 Select expression is specified.

Parameters

- **source_s3_key** (*str*) – The key to be retrieved from S3. (templated)
- **source_aws_conn_id** (*str*) – source s3 connection
- **dest_s3_key** (*str*) – The key to be written from S3. (templated)
- **dest_aws_conn_id** (*str*) – destination s3 connection
- **replace** (*bool*) – Replace dest S3 key if it already exists
- **transform_script** (*str*) – location of the executable transformation script

- **select_expression** (*str*) – S3 Select expression

S3ListOperator

class airflow.contrib.operators.s3_list_operator.**S3ListOperator**(*bucket*, *prefix=''*, *delimiter=''*, *aws_conn_id='aws_default'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

List all objects from the bucket with the given string prefix in name.

This operator returns a python list with the name of objects which can be used by *xcom* in the downstream task.

Parameters

- **bucket** (*string*) – The S3 bucket where to find the objects. (templated)
- **prefix** (*string*) – Prefix string to filters the objects whose name begin with such prefix. (templated)
- **delimiter** (*string*) – the delimiter marks key hierarchy. (templated)
- **aws_conn_id** (*string*) – The connection ID to use when connecting to S3 storage.

Example: The following operator would list all the files (excluding subfolders) from the S3 `customers/2018/04/` key in the `data` bucket.

```
s3_file = S3ListOperator(
    task_id='list_3s_files',
    bucket='data',
    prefix='customers/2018/04/',
    delimiter='/',
    aws_conn_id='aws_customers_conn'
)
```

S3ToGoogleCloudStorageOperator

class airflow.contrib.operators.s3_to_gcs_operator.**S3ToGoogleCloudStorageOperator**(*bucket*, *prefix=''*, *delimiter=''*, *aws_conn_id*, *dest_gcs_co*, *dest_gcs=No*, *delegate_to=Nor*, *replace=False*, **args*, ***kwargs*)

Bases: *airflow.contrib.operators.s3_list_operator.S3ListOperator*

Synchronizes an S3 key, possibly a prefix, with a Google Cloud Storage destination path.

Parameters

- **bucket** (*string*) – The S3 bucket where to find the objects. (templated)
- **prefix** (*string*) – Prefix string which filters objects whose name begin with such prefix. (templated)
- **delimiter** (*string*) – the delimiter marks key hierarchy. (templated)
- **aws_conn_id** (*string*) – The source S3 connection
- **dest_gcs_conn_id** (*string*) – The destination connection ID to use when connecting to Google Cloud Storage.
- **dest_gcs** (*string*) – The destination Google Cloud Storage bucket and prefix where you want to store the files. (templated)
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **replace** (*bool*) – Whether you want to replace existing destination files or not.

Example: .. code-block:: python

s3_to_gcs_op = S3ToGoogleCloudStorageOperator(task_id='s3_to_gcs_example', bucket='my-s3-bucket', prefix='data/customers-201804', dest_gcs_conn_id='google_cloud_default', dest_gcs='gs://my.gcs.bucket/some/customers/', replace=False, dag=my-dag)

Note that `bucket`, `prefix`, `delimiter` and `dest_gcs` are templated, so you can use variables in them if you wish.

S3ToHiveTransfer

3.16.3.3 AWS EC2 Container Service

- *ECSOperator* : Execute a task on AWS EC2 Container Service.

ECSOperator

class airflow.contrib.operators.ecs_operator.**ECSOperator**(*task_definition*, *cluster*, *overrides*, *aws_conn_id=None*, *region_name=None*, *launch_type='EC2'*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Execute a task on AWS EC2 Container Service

Parameters

- **task_definition** (*str*) – the task definition name on EC2 Container Service
- **cluster** (*str*) – the cluster name on EC2 Container Service
- **aws_conn_id** (*str*) – connection id of AWS credentials / region name. If None, credential boto3 strategy will be used (http://boto3.readthedocs.io/en/latest/guide/configuration.html).
- **region_name** – region name to use in AWS Hook. Override the region_name in connection (if provided)
- **launch_type** – the launch type on which to run your task ('EC2' or 'FARGATE')

Param overrides: the same parameter that boto3 will receive (templated): http://boto3.readthedocs.org/en/latest/reference/services/ecs.html#ECS.Client.run_task

Type overrides: dict

Type launch_type: str

3.16.3.4 AWS Batch Service

- *AWSBatchOperator* : Execute a task on AWS Batch Service.

AWSBatchOperator

class airflow.contrib.operators.awsbatch_operator.**AWSBatchOperator**(*job_name*, *job_definition*, *job_queue*, *overrides*, *max_retries=4200*, *aws_conn_id=None*, *region_name=None*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Execute a job on AWS Batch Service

Parameters

- **job_name** (*str*) – the name for the job that will run on AWS Batch
- **job_definition** (*str*) – the job definition name on AWS Batch
- **job_queue** (*str*) – the queue name on AWS Batch

- **max_retries** (*int*) – exponential backoff retries while waiter is not merged, 4200 = 48 hours
- **aws_conn_id** (*str*) – connection id of AWS credentials / region name. If None, credential boto3 strategy will be used (http://boto3.readthedocs.io/en/latest/guide/configuration.html).
- **region_name** – region name to use in AWS Hook. Override the region_name in connection (if provided)

Param overrides: the same parameter that boto3 will receive on containerOverrides (templated): http://boto3.readthedocs.io/en/latest/reference/services/batch.html#submit_job

Type overrides: dict

3.16.3.5 AWS RedShift

- *AwsRedshiftClusterSensor* : Waits for a Redshift cluster to reach a specific status.
- *RedshiftHook* : Interact with AWS Redshift, using the boto3 library.
- *RedshiftToS3Transfer* : Executes an unload command to S3 as CSV with or without headers.
- *S3ToRedshiftTransfer* : Executes an copy command from S3 as CSV with or without headers.

AwsRedshiftClusterSensor

class airflow.contrib.sensors.aws_redshift_cluster_sensor.**AwsRedshiftClusterSensor**(*cluster_identifier*, *target_status='available'*, *aws_conn_id='aws_default'*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a Redshift cluster to reach a specific status.

Parameters

- **cluster_identifier** (*str*) – The identifier for the cluster being pinged.
- **target_status** (*str*) – The cluster status desired.

poke(*context*)

Function that the sensors defined while deriving this class should override.

RedshiftHook

class airflow.contrib.hooks.redshift_hook.**RedshiftHook**(*aws_conn_id='aws_default'*)

Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS Redshift, using the boto3 library

cluster_status(*cluster_identifier*)

Return status of a cluster

Parameters **cluster_identifier** (*str*) – unique identifier of a cluster

create_cluster_snapshot(*snapshot_identifier*, *cluster_identifier*)
Creates a snapshot of a cluster

Parameters

- **snapshot_identifier** (*str*) – unique identifier for a snapshot of a cluster
- **cluster_identifier** (*str*) – unique identifier of a cluster

delete_cluster(*cluster_identifier*, *skip_final_cluster_snapshot=True*, *final_cluster_snapshot_identifier=''*)
Delete a cluster and optionally create a snapshot

Parameters

- **cluster_identifier** (*str*) – unique identifier of a cluster
- **skip_final_cluster_snapshot** (*bool*) – determines cluster snapshot creation
- **final_cluster_snapshot_identifier** (*str*) – name of final cluster snapshot

describe_cluster_snapshots(*cluster_identifier*)
Gets a list of snapshots for a cluster

Parameters **cluster_identifier** (*str*) – unique identifier of a cluster

restore_from_cluster_snapshot(*cluster_identifier*, *snapshot_identifier*)
Restores a cluster from its snapshot

Parameters

- **cluster_identifier** (*str*) – unique identifier of a cluster
- **snapshot_identifier** (*str*) – unique identifier for a snapshot of a cluster

RedshiftToS3Transfer

S3ToRedshiftTransfer

3.16.4 Databricks

Databricks has contributed an Airflow operator which enables submitting runs to the Databricks platform. Internally the operator talks to the `api/2.0/jobs/runs/submit` endpoint.

3.16.4.1 DatabricksSubmitRunOperator

class airflow.contrib.operators.databricks_operator.**DatabricksSubmitRunOperator**(*json=None*, *spark_jar_task=* *note-* *book_task=Non* *new_cluster=N* *ex-* *ist-* *ing_cluster_id=* *li-* *braries=None*, *run_name=Non* *time-* *out_seconds=N* *databricks_con* *polling_period_* *databricks_retr* *do_xcom_push=* ***kwargs*)

Bases: *airflow.models.BaseOperator*

Submits an Spark job run to Databricks using the api/2.0/jobs/runs/submit API endpoint.

There are two ways to instantiate this operator.

In the first way, you can take the JSON payload that you typically use to call the `api/2.0/jobs/runs/submit` endpoint and pass it directly to our `DatabricksSubmitRunOperator` through the `json` parameter. For example

```
json = {
  'new_cluster': {
    'spark_version': '2.1.0-db3-scala2.11',
    'num_workers': 2
  },
  'notebook_task': {
    'notebook_path': '/Users/airflow@example.com/PrepareData',
  },
}
notebook_run = DatabricksSubmitRunOperator(task_id='notebook_run', json=json)
```

Another way to accomplish the same thing is to use the named parameters of the `DatabricksSubmitRunOperator` directly. Note that there is exactly one named parameter for each top level parameter in the `runs/submit` endpoint. In this method, your code would look like this:

```
new_cluster = {
  'spark_version': '2.1.0-db3-scala2.11',
  'num_workers': 2
}
notebook_task = {
  'notebook_path': '/Users/airflow@example.com/PrepareData',
}
notebook_run = DatabricksSubmitRunOperator(
    task_id='notebook_run',
    new_cluster=new_cluster,
    notebook_task=notebook_task)
```

In the case where both the json parameter **AND** the named parameters are provided, they will be merged together.

If there are conflicts during the merge, the named parameters will take precedence and override the top level `json` keys.

Currently the named parameters that `DatabricksSubmitRunOperator` supports are

- `spark_jar_task`
- `notebook_task`
- `new_cluster`
- `existing_cluster_id`
- `libraries`
- `run_name`
- `timeout_seconds`

Parameters

- **`json`** (*`dict`*) – A JSON object containing API parameters which will be passed directly to the `api/2.0/jobs/runs/submit` endpoint. The other named parameters (i.e. `spark_jar_task`, `notebook_task`..) to this operator will be merged with this json dictionary if they are provided. If there are conflicts during the merge, the named parameters will take precedence and override the top level json keys. (templated)

  See also:

  For more information about templating see *Jinja Templating*. https://docs.databricks.com/api/latest/jobs.html#runs-submit

- **`spark_jar_task`** (*`dict`*) – The main class and parameters for the JAR task. Note that the actual JAR is specified in the `libraries`. *EITHER* `spark_jar_task` *OR* `notebook_task` should be specified. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/jobs.html#jobssparkjartask

- **`notebook_task`** (*`dict`*) – The notebook path and parameters for the notebook task. *EITHER* `spark_jar_task` *OR* `notebook_task` should be specified. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/jobs.html#jobsnotebooktask

- **`new_cluster`** (*`dict`*) – Specs for a new cluster on which this task will be run. *EITHER* `new_cluster` *OR* `existing_cluster_id` should be specified. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/jobs.html#jobsclusterspecnewcluster

- **`existing_cluster_id`** (*`string`*) – ID for existing cluster on which to run this task. *EITHER* `new_cluster` *OR* `existing_cluster_id` should be specified. This field will be templated.

- **`libraries`** (*`list of dicts`*) – Libraries which this run will use. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/libraries.html#managedlibrarieslibrary

- **run_name** (*string*) – The run name used for this task. By default this will be set to the Airflow `task_id`. This `task_id` is a required parameter of the superclass `BaseOperator`. This field will be templated.
- **timeout_seconds** (*int32*) – The timeout for this run. By default a value of 0 is used which means to have no timeout. This field will be templated.
- **databricks_conn_id** (*string*) – The name of the Airflow connection to use. By default and in the common case this will be `databricks_default`. To use token based authentication, provide the key `token` in the extra field for the connection.
- **polling_period_seconds** (*int*) – Controls the rate which we poll for the result of this run. By default the operator will poll every 30 seconds.
- **databricks_retry_limit** (*int*) – Amount of times retry if the Databricks backend is unreachable. Its value must be greater than or equal to 1.
- **do_xcom_push** (*boolean*) – Whether we should push run_id and run_page_url to xcom.

3.16.5 GCP: Google Cloud Platform

Airflow has extensive support for the Google Cloud Platform. But note that most Hooks and Operators are in the contrib section. Meaning that they have a *beta* status, meaning that they can have breaking changes between minor releases.

See the *GCP connection type* documentation to configure connections to GCP.

3.16.5.1 Logging

Airflow can be configured to read and write task logs in Google Cloud Storage. See *Writing Logs to Google Cloud Storage*.

3.16.5.2 BigQuery

BigQuery Operators

- *BigQueryCheckOperator* : Performs checks against a SQL query that will return a single row with different values.
- *BigQueryValueCheckOperator* : Performs a simple value check using SQL code.
- *BigQueryIntervalCheckOperator* : Checks that the values of metrics given as SQL expressions are within a certain tolerance of the ones from days_back before.
- *BigQueryCreateEmptyTableOperator* : Creates a new, empty table in the specified BigQuery dataset optionally with schema.
- *BigQueryCreateExternalTableOperator* : Creates a new, external table in the dataset with the data in Google Cloud Storage.
- *BigQueryOperator* : Executes BigQuery SQL queries in a specific BigQuery database.
- *BigQueryToBigQueryOperator* : Copy a BigQuery table to another BigQuery table.
- *BigQueryToCloudStorageOperator* : Transfers a BigQuery table to a Google Cloud Storage bucket

BigQueryCheckOperator

class airflow.contrib.operators.bigquery_check_operator.**BigQueryCheckOperator**(*sql*, *bigquery_conn_id='b*, **args*, ***kwargs*)

Bases: *airflow.operators.check_operator.CheckOperator*

Performs checks against BigQuery. The `BigQueryCheckOperator` expects a sql query that will return a single row. Each value on that first row is evaluated using python `bool` casting. If any of the values return `False` the check is failed and errors out.

Note that Python bool casting evals the following as `False`:

- `False`
- `0`
- Empty string (`""`)
- Empty list (`[]`)
- Empty dictionary or set (`{}`)

Given a query like `SELECT COUNT(*) FROM foo`, it will fail only if the count `== 0`. You can craft much more complex query that could, for instance, check that the table has the same number of rows as the source table upstream, or that the count of today's partition is greater than yesterday's partition, or that a set of metrics are less than 3 standard deviation for the 7 day average.

This operator can be used as a data quality check in your pipeline, and depending on where you put it in your DAG, you have the choice to stop the critical path, preventing from publishing dubious data, or on the side and receive email alterts without stopping the progress of the DAG.

Parameters

- **sql** (*string*) – the sql to be executed
- **bigquery_conn_id** (*string*) – reference to the BigQuery database

BigQueryValueCheckOperator

class airflow.contrib.operators.bigquery_check_operator.**BigQueryValueCheckOperator**(*sql*, *pass_value*, *tolerance=None*, *bigquery_conn*, **args*, ***kwargs*)

Bases: *airflow.operators.check_operator.ValueCheckOperator*

Performs a simple value check using sql code.

Parameters **sql** (*string*) – the sql to be executed

BigQueryIntervalCheckOperator

class airflow.contrib.operators.bigquery_check_operator.**BigQueryIntervalCheckOperator**(*table*, *metrics_th* *date_fi* *days_b* *7*, *big-* *query_* **args*, ***kwa*

Bases: *airflow.operators.check_operator.IntervalCheckOperator*

Checks that the values of metrics given as SQL expressions are within a certain tolerance of the ones from days_back before.

This method constructs a query like so

```
SELECT {metrics_threshold_dict_key} FROM {table}
    WHERE {date_filter_column}=<date>
```

Parameters

- **table** (*str*) – the table name
- **days_back** (*int*) – number of days between ds and the ds we want to check against. Defaults to 7 days
- **metrics_threshold** (*dict*) – a dictionary of ratios indexed by metrics, for example 'COUNT(*)': 1.5 would require a 50 percent or less difference between the current day, and the prior days_back.

BigQueryGetDataOperator

class airflow.contrib.operators.bigquery_get_data.**BigQueryGetDataOperator**(*dataset_id*, *ta-* *ble_id*, *max_results='100'*, *se-* *lected_fields=None*, *big-* *query_conn_id='bigque* *del-* *e-* *gate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Fetches the data from a BigQuery table (alternatively fetch data for selected columns) and returns data in a python list. The number of elements in the returned list will be equal to the number of rows fetched. Each element in the list will again be a list where element would represent the columns values for that row.

Example Result: `[['Tony', '10'], ['Mike', '20'], ['Steve', '15']]`

Note: If you pass fields to `selected_fields` which are in different order than the order of columns already in BQ table, the data will still be in the order of BQ table. For example if the BQ table has 3 columns as `[A,B,C]` and you pass 'B,A' in the `selected_fields` the data would still be of the form `'A,B'`.

Example:

```
get_data = BigQueryGetDataOperator(
    task_id='get_data_from_bq',
    dataset_id='test_dataset',
    table_id='Transaction_partitions',
    max_results='100',
    selected_fields='DATE',
    bigquery_conn_id='airflow-service-account'
)
```

Parameters

- **dataset_id** – The dataset ID of the requested table. (templated)
- **table_id** (*string*) – The table ID of the requested table. (templated)
- **max_results** (*string*) – The maximum number of records (rows) to be fetched from the table. (templated)
- **selected_fields** (*string*) – List of fields to return (comma-separated). If unspecified, all fields are returned.
- **bigquery_conn_id** (*string*) – reference to a specific BigQuery hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

BigQueryCreateEmptyTableOperator

class airflow.contrib.operators.bigquery_operator.**BigQueryCreateEmptyTableOperator**(*dataset_id*, *table_id*, *project_id=* *schema_fie* *gcs_schem* *time_partit* *bigquery_con* *google_clo* *delegate_to=N* **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Creates a new, empty table in the specified BigQuery dataset, optionally with schema.

The schema to be used for the BigQuery table may be specified in one of two ways. You may either directly pass the schema fields in, or you may point the operator to a Google cloud storage object name. The object in Google cloud storage must be a JSON file with the schema fields in it. You can also create a table without schema.

Parameters

- **project_id** (*string*) – The project to create the table into. (templated)
- **dataset_id** (*string*) – The dataset to create the table into. (templated)
- **table_id** (*string*) – The Name of the table to be created. (templated)
- **schema_fields** (*list*) – If set, the schema field list as defined here: https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs#configuration.load.schema

  Example:

```
schema_fields=[{"name": "emp_name", "type": "STRING", "mode":
↪"REQUIRED"},
               {"name": "salary", "type": "INTEGER", "mode":
↪"NULLABLE"}]
```

- **gcs_schema_object** (*string*) – Full path to the JSON file containing schema (templated). For example: `gs://test-bucket/dir1/dir2/employee_schema.json`
- **time_partitioning** (*dict*) – configure optional time partitioning fields i.e. partition by field, type and expiration as per API specifications.

  See also:

  https://cloud.google.com/bigquery/docs/reference/rest/v2/tables#timePartitioning
- **bigquery_conn_id** (*string*) – Reference to a specific BigQuery hook.
- **google_cloud_storage_conn_id** (*string*) – Reference to a specific Google cloud storage hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

Example (with schema JSON in GCS):

```
CreateTable = BigQueryCreateEmptyTableOperator(
    task_id='BigQueryCreateEmptyTableOperator_task',
    dataset_id='ODS',
    table_id='Employees',
    project_id='internal-gcp-project',
    gcs_schema_object='gs://schema-bucket/employee_schema.json',
    bigquery_conn_id='airflow-service-account',
    google_cloud_storage_conn_id='airflow-service-account'
)
```

Corresponding Schema file (`employee_schema.json`):

```
[
  {
    "mode": "NULLABLE",
    "name": "emp_name",
    "type": "STRING"
  },
  {
    "mode": "REQUIRED",
    "name": "salary",
    "type": "INTEGER"
```

(continues on next page)

(continued from previous page)

```
    }
]
```

Example (with schema in the DAG):

```
CreateTable = BigQueryCreateEmptyTableOperator(
    task_id='BigQueryCreateEmptyTableOperator_task',
    dataset_id='ODS',
    table_id='Employees',
    project_id='internal-gcp-project',
    schema_fields=[{"name": "emp_name", "type": "STRING", "mode": "REQUIRED"},
                   {"name": "salary", "type": "INTEGER", "mode": "NULLABLE"}],
    bigquery_conn_id='airflow-service-account',
    google_cloud_storage_conn_id='airflow-service-account'
)
```

BigQueryCreateExternalTableOperator

class airflow.contrib.operators.bigquery_operator.**BigQueryCreateExternalTableOperator**(*bucket source des- ti- na- tion_p schem schem source com- pres- sion=' skip_le field_a ', max_b quote_ al- low_qu al- low_ja big- query_ google del- e- gate_t src_fm *args, **kwa

Bases: *airflow.models.BaseOperator*

Creates a new external table in the dataset with the data in Google Cloud Storage.

The schema to be used for the BigQuery table may be specified in one of two ways. You may either directly pass the schema fields in, or you may point the operator to a Google cloud storage object name. The object in

Google cloud storage must be a JSON file with the schema fields in it.

Parameters

- **bucket** (*string*) – The bucket to point the external table to. (templated)
- **source_objects** – List of Google cloud storage URIs to point table to. (templated) If source_format is 'DATASTORE_BACKUP', the list must only contain a single URI.
- **destination_project_dataset_table** (*string*) – The dotted (<project>.)<dataset>.<table> BigQuery table to load data into (templated). If <project> is not included, project will be the project defined in the connection json.
- **schema_fields** (*list*) – If set, the schema field list as defined here: https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs#configuration.load.schema

  Example:

```
schema_fields=[{"name": "emp_name", "type": "STRING", "mode":
↪"REQUIRED"},
               {"name": "salary", "type": "INTEGER", "mode":
↪"NULLABLE"}]
```

  Should not be set when source_format is 'DATASTORE_BACKUP'.
- **schema_object** – If set, a GCS object path pointing to a .json file that contains the schema for the table. (templated)
- **schema_object** – string
- **source_format** (*string*) – File format of the data.
- **compression** (*string*) – [Optional] The compression type of the data source. Possible values include GZIP and NONE. The default value is NONE. This setting is ignored for Google Cloud Bigtable, Google Cloud Datastore backups and Avro formats.
- **skip_leading_rows** (*int*) – Number of rows to skip when loading from a CSV.
- **field_delimiter** (*string*) – The delimiter to use for the CSV.
- **max_bad_records** (*int*) – The maximum number of bad records that BigQuery can ignore when running the job.
- **quote_character** (*string*) – The value that is used to quote data sections in a CSV file.
- **allow_quoted_newlines** (*boolean*) – Whether to allow quoted newlines (true) or not (false).
- **allow_jagged_rows** (*bool*) – Accept rows that are missing trailing optional columns. The missing values are treated as nulls. If false, records with missing trailing columns are treated as bad records, and if there are too many bad records, an invalid error is returned in the job result. Only applicable to CSV, ignored for other formats.
- **bigquery_conn_id** (*string*) – Reference to a specific BigQuery hook.
- **google_cloud_storage_conn_id** (*string*) – Reference to a specific Google cloud storage hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **src_fmt_configs** (*dict*) – configure optional fields specific to the source format

BigQueryOperator

class airflow.contrib.operators.bigquery_operator.**BigQueryOperator**(*bql=None*, *sql=None*, *destination_dataset_table=False*, *write_disposition='WRITE_EMPTY'*, *allow_large_results=False*, *flatten_results=False*, *bigquery_conn_id='bigquery_default'*, *delegate_to=None*, *udf_config=False*, *use_legacy_sql=True*, *maximum_billing_tier=None*, *maximum_bytes_billed=None*, *create_disposition='CREATE_IF_NEEDED'*, *schema_update_options=()*, *query_params=None*, *priority='INTERACTIVE'*, *time_partitioning={}*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes BigQuery SQL queries in a specific BigQuery database

Parameters

- **bql** (*Can receive a str representing a sql statement, a list of str (sql statements), or reference to a template file. Template reference are recognized by str ending in '.sql'.*) – (Deprecated. Use *sql* parameter instead) the sql code to be executed (templated)
- **sql** (*Can receive a str representing a sql statement, a list of str (sql statements), or reference to a template file. Template reference are recognized by str ending in '.sql'.*) – the sql code to be executed (templated)
- **destination_dataset_table** (*string*) – A dotted (<project>.|<project>:)<dataset>.<table> that, if set, will store the results of the query. (templated)
- **write_disposition** (*string*) – Specifies the action that occurs if the destination table already exists. (default: 'WRITE_EMPTY')
- **create_disposition** (*string*) – Specifies whether the job is allowed to create new tables. (default: 'CREATE_IF_NEEDED')
- **allow_large_results** (*boolean*) – Whether to allow large results.

- **flatten_results** (*boolean*) – If true and query uses legacy SQL dialect, flattens all nested and repeated fields in the query results. `allow_large_results` must be `true` if this is set to `false`. For standard SQL queries, this flag is ignored and results are never flattened.
- **bigquery_conn_id** (*string*) – reference to a specific BigQuery hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **udf_config** (*list*) – The User Defined Function configuration for the query. See https://cloud.google.com/bigquery/user-defined-functions for details.
- **use_legacy_sql** (*boolean*) – Whether to use legacy SQL (true) or standard SQL (false).
- **maximum_billing_tier** (*integer*) – Positive integer that serves as a multiplier of the basic price. Defaults to None, in which case it uses the value set in the project.
- **maximum_bytes_billed** (*float*) – Limits the bytes billed for this job. Queries that will have bytes billed beyond this limit will fail (without incurring a charge). If unspecified, this will be set to your project default.
- **schema_update_options** (*tuple*) – Allows the schema of the destination table to be updated as a side effect of the load job.
- **query_params** (*dict*) – a dictionary containing query parameter types and values, passed to BigQuery.
- **priority** (*string*) – Specifies a priority for the query. Possible values include INTERACTIVE and BATCH. The default value is INTERACTIVE.
- **time_partitioning** (*dict*) – configure optional time partitioning fields i.e. partition by field, type and expiration as per API specifications. Note that 'field' is not available in conjunction with dataset.table$partition.

BigQueryTableDeleteOperator

class airflow.contrib.operators.bigquery_table_delete_operator.**BigQueryTableDeleteOperator**

Bases: *airflow.models.BaseOperator*

Deletes BigQuery tables

Parameters

- **deletion_dataset_table** (*string*) – A dotted (<project>.|<project>:)<dataset>.<table> that indicates which table will be deleted. (templated)
- **bigquery_conn_id** (*string*) – reference to a specific BigQuery hook.

- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **ignore_if_missing** (*boolean*) – if True, then return success even if the requested table does not exist.

BigQueryToBigQueryOperator

class airflow.contrib.operators.bigquery_to_bigquery.**BigQueryToBigQueryOperator**(*source_project_dataset_tables*, *destination_project_dataset_table*, *write_disposition='WRITE_EMPTY'*, *create_disposition='CREATE_IF_NEEDED'*, *bigquery_conn_id='bigquery_default'*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Copies data from one BigQuery table to another.

See also:

For more details about these parameters: https://cloud.google.com/bigquery/docs/reference/v2/jobs#configuration.copy

Parameters

- **source_project_dataset_tables** (*list|string*) – One or more dotted (project:|project.)<dataset>.<table> BigQuery tables to use as the source data. If <project> is not included, project will be the project defined in the connection json. Use a list if there are multiple source tables. (templated)
- **destination_project_dataset_table** (*string*) – The destination BigQuery table. Format is: (project:|project.)<dataset>.<table> (templated)
- **write_disposition** (*string*) – The write disposition if the table already exists.
- **create_disposition** (*string*) – The create disposition if the table doesn't exist.
- **bigquery_conn_id** (*string*) – reference to a specific BigQuery hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

BigQueryToCloudStorageOperator

class airflow.contrib.operators.bigquery_to_gcs.**BigQueryToCloudStorageOperator**(*source_project_d*
destination_cloud_stora
compression='NONE',
export_format='CS
field_delimiter='
',
print_header=Tr
bigquery_conn_id=
delegate_to=None,
**args*,
***kwargs*)

Bases: *airflow.models.BaseOperator*

Transfers a BigQuery table to a Google Cloud Storage bucket.

See also:

For more details about these parameters: https://cloud.google.com/bigquery/docs/reference/v2/jobs

Parameters

- **source_project_dataset_table** (*string*) – The dotted (<project>.|<project>:)<dataset>.<table> BigQuery table to use as the source data. If <project> is not included, project will be the project defined in the connection json. (templated)
- **destination_cloud_storage_uris** (*list*) – The destination Google Cloud Storage URI (e.g. gs://some-bucket/some-file.txt). (templated) Follows convention defined here: https://cloud.google.com/bigquery/exporting-data-from-bigquery#exportingmultiple
- **compression** (*string*) – Type of compression to use.
- **export_format** – File format to export.
- **field_delimiter** (*string*) – The delimiter to use when extracting to a CSV.
- **print_header** (*boolean*) – Whether to print a header for a CSV file extract.
- **bigquery_conn_id** (*string*) – reference to a specific BigQuery hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

BigQueryHook

class airflow.contrib.hooks.bigquery_hook.**BigQueryHook**(*bigquery_conn_id='bigquery_default'*, *delegate_to=None*, *use_legacy_sql=True*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*, *airflow.hooks.dbapi_hook.DbApiHook*, airflow.utils.log.logging_mixin.LoggingMixin

Interact with BigQuery. This hook uses the Google Cloud Platform connection.

get_conn()
Returns a BigQuery PEP 249 connection object.

get_pandas_df(*sql*, *parameters=None*, *dialect=None*)
Returns a Pandas DataFrame for the results produced by a BigQuery query. The DbApiHook method must be overridden because Pandas doesn't support PEP 249 connections, except for SQLite. See:

https://github.com/pydata/pandas/blob/master/pandas/io/sql.py#L447 https://github.com/pydata/pandas/issues/6900

Parameters

- **sql** (*string*) – The BigQuery SQL to execute.
- **parameters** (*mapping or iterable*) – The parameters to render the SQL query with (not used, leave to override superclass method)
- **dialect** (*string in {'legacy', 'standard'}*) – Dialect of BigQuery SQL – legacy SQL or standard SQL defaults to use *self.use_legacy_sql* if not specified

get_service()
Returns a BigQuery service object.

insert_rows(*table*, *rows*, *target_fields=None*, *commit_every=1000*)
Insertion is currently unsupported. Theoretically, you could use BigQuery's streaming API to insert rows into a table, but this hasn't been implemented.

table_exists(*project_id*, *dataset_id*, *table_id*)
Checks for the existence of a table in Google BigQuery.

Parameters

- **project_id** (*string*) – The Google cloud project in which to look for the table. The connection supplied to the hook must provide access to the specified project.
- **dataset_id** (*string*) – The name of the dataset in which to look for the table.
- **table_id** (*string*) – The name of the table to check the existence of.

3.16.5.3 Cloud DataFlow

DataFlow Operators

- *DataFlowJavaOperator* : launching Cloud Dataflow jobs written in Java.
- *DataflowTemplateOperator* : launching a templated Cloud DataFlow batch job.
- *DataFlowPythonOperator* : launching Cloud Dataflow jobs written in python.

DataFlowJavaOperator

class airflow.contrib.operators.dataflow_operator.**DataFlowJavaOperator**(*jar*, *dataflow_default_options=N*, *options=None*, *gcp_conn_id='google_cloud*, *delegate_to=None*, *poll_sleep=10*, *job_class=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Java Cloud DataFlow batch job. The parameters of the operation will be passed to the job.

It's a good practice to define dataflow_* parameters in the default_args of the dag like the project, zone and staging location.

```
default_args = {
    'dataflow_default_options': {
        'project': 'my-gcp-project',
        'zone': 'europe-west1-d',
        'stagingLocation': 'gs://my-staging-bucket/staging/'
    }
}
```

You need to pass the path to your dataflow as a file reference with the `jar` parameter, the jar needs to be a self executing jar (see documentation here: https://beam.apache.org/documentation/runners/dataflow/#self-executing-jar). Use `options` to pass on options to your job.

```
t1 = DataFlowOperation(
    task_id='datapflow_example',
    jar='{{var.value.gcp_dataflow_base}}pipeline/build/libs/pipeline-example-1.0.
↪jar',
    options={
        'autoscalingAlgorithm': 'BASIC',
        'maxNumWorkers': '50',
        'start': '{{ds}}',
        'partitionType': 'DAY',
        'labels': {'foo' : 'bar'}
    },
    gcp_conn_id='gcp-airflow-service-account',
    dag=my-dag)
```

Both `jar` and `options` are templated so you can use variables in them.

```
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date':
        (2016, 8, 1),
    'email': ['alex@vanboxel.be'],
    'email_on_failure': False,
    'email_on_retry': False,
```

(continues on next page)

(continued from previous page)

```
    'retries': 1,
    'retry_delay': timedelta(minutes=30),
    'dataflow_default_options': {
        'project': 'my-gcp-project',
        'zone': 'us-central1-f',
        'stagingLocation': 'gs://bucket/tmp/dataflow/staging/',
    }
}

dag = DAG('test-dag', default_args=default_args)

task = DataFlowJavaOperator(
    gcp_conn_id='gcp_default',
    task_id='normalize-cal',
    jar='{{var.value.gcp_dataflow_base}}pipeline-ingress-cal-normalize-1.0.jar',
    options={
        'autoscalingAlgorithm': 'BASIC',
        'maxNumWorkers': '50',
        'start': '{{ds}}',
        'partitionType': 'DAY'

    },
    dag=dag)
```

DataflowTemplateOperator

class airflow.contrib.operators.dataflow_operator.**DataflowTemplateOperator**(*template*, *dataflow_default_optio* *pa-* *ram-* *e-* *ters=None*, *gcp_conn_id='google_* *del-* *e-* *gate_to=None*, *poll_sleep=10*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Templated Cloud DataFlow batch job. The parameters of the operation will be passed to the job. It's a good practice to define dataflow_* parameters in the default_args of the dag like the project, zone and staging location.

See also:

https://cloud.google.com/dataflow/docs/reference/rest/v1b3/LaunchTemplateParameters https://cloud.google.com/dataflow/docs/reference/rest/v1b3/RuntimeEnvironment

```
default_args = {
    'dataflow_default_options': {
        'project': 'my-gcp-project'
        'zone': 'europe-west1-d',
        'tempLocation': 'gs://my-staging-bucket/staging/'
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

You need to pass the path to your dataflow template as a file reference with the `template` parameter. Use `parameters` to pass on parameters to your job. Use `environment` to pass on runtime environment variables to your job.

```
t1 = DataflowTemplateOperator(
    task_id='datapflow_example',
    template='{{var.value.gcp_dataflow_base}}',
    parameters={
        'inputFile': "gs://bucket/input/my_input.txt",
        'outputFile': "gs://bucket/output/my_output.txt"
    },
    gcp_conn_id='gcp-airflow-service-account',
    dag=my-dag)
```

`template`, `dataflow_default_options` and `parameters` are templated so you can use variables in them.

DataFlowPythonOperator

class airflow.contrib.operators.dataflow_operator.**DataFlowPythonOperator**(*py_file*, *py_options=None*, *dataflow_default_options=*, *options=None*, *gcp_conn_id='google_cl*, *delegate_to=None*, *poll_sleep=10*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

execute(*context*)

Execute the python dataflow job.

DataFlowHook

class airflow.contrib.hooks.gcp_dataflow_hook.**DataFlowHook**(*gcp_conn_id='google_cloud_default'*, *delegate_to=None*, *poll_sleep=10*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

get_conn()

Returns a Google Cloud Storage service object.

3.16.5.4 Cloud DataProc

DataProc Operators

- *DataprocClusterCreateOperator* : Create a new cluster on Google Cloud Dataproc.
- *DataprocClusterDeleteOperator* : Delete a cluster on Google Cloud Dataproc.
- *DataprocClusterScaleOperator* : Scale up or down a cluster on Google Cloud Dataproc.
- *DataProcPigOperator* : Start a Pig query Job on a Cloud DataProc cluster.
- *DataProcHiveOperator* : Start a Hive query Job on a Cloud DataProc cluster.
- *DataProcSparkSqlOperator* : Start a Spark SQL query Job on a Cloud DataProc cluster.
- *DataProcSparkOperator* : Start a Spark Job on a Cloud DataProc cluster.
- *DataProcHadoopOperator* : Start a Hadoop Job on a Cloud DataProc cluster.
- *DataProcPySparkOperator* : Start a PySpark Job on a Cloud DataProc cluster.
- *DataprocWorkflowTemplateInstantiateOperator* : Instantiate a WorkflowTemplate on Google Cloud Dataproc.
- *DataprocWorkflowTemplateInstantiateInlineOperator* : Instantiate a WorkflowTemplate Inline on Google Cloud Dataproc.

DataprocClusterCreateOperator

class airflow.contrib.operators.dataproc_operator.**DataprocClusterCreateOperator**(*cluster_name*, *project_id*, *num_workers*, *zone*, *network_uri=None*, *subnetwork_uri=None*, *internal_ip_only=N*, *tags=None*, *storage_bucket=No*, *init_actions_ur*, *init_action_tim*, *metadata=None*, *image_version=N*, *properties=None*, *master_machin*, *standard-4'*, *master_disk_size=5*, *worker_machin*, *standard-4'*, *worker_disk_si*, *num_preemptib*, *labels=None*, *region='global'*, *gcp_conn_id='*, *delegate_to=None*, *service_account=*, *service_account_s*, *idle_delete_ttl=*, *auto_delete_tim*, *auto_delete_ttl=*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Create a new cluster on Google Cloud Dataproc. The operator will wait until the creation is successful or an error occurs in the creation process.

The parameters allow to configure the cluster. Please refer to

https://cloud.google.com/dataproc/docs/reference/rest/v1/projects.regions.clusters

for a detailed explanation on the different parameters. Most of the configuration parameters detailed in the link are available as a parameter to this operator.

Parameters

- **cluster_name** (*string*) – The name of the DataProc cluster to create. (templated)
- **project_id** (*string*) – The ID of the google cloud project in which to create the cluster. (templated)
- **num_workers** (*int*) – The # of workers to spin up
- **storage_bucket** (*string*) – The storage bucket to use, setting to None lets dataproc generate a custom one for you
- **init_actions_uris** (*list[string]*) – List of GCS uri's containing dataproc initialization scripts
- **init_action_timeout** (*string*) – Amount of time executable scripts in init_actions_uris has to complete
- **metadata** (*dict*) – dict of key-value google compute engine metadata entries to add to all instances
- **image_version** (*string*) – the version of software inside the Dataproc cluster
- **properties** (*dict*) – dict of properties to set on config files (e.g. spark-defaults.conf), see https://cloud.google.com/dataproc/docs/reference/rest/v1/projects.regions.clusters#SoftwareConfig
- **master_machine_type** (*string*) – Compute engine machine type to use for the master node
- **master_disk_size** (*int*) – Disk size for the master node
- **worker_machine_type** (*string*) – Compute engine machine type to use for the worker nodes
- **worker_disk_size** (*int*) – Disk size for the worker nodes
- **num_preemptible_workers** (*int*) – The # of preemptible worker nodes to spin up
- **labels** (*dict*) – dict of labels to add to the cluster
- **zone** (*string*) – The zone where the cluster will be located. (templated)
- **network_uri** (*string*) – The network uri to be used for machine communication, cannot be specified with subnetwork_uri
- **subnetwork_uri** (*string*) – The subnetwork uri to be used for machine communication, cannot be specified with network_uri
- **internal_ip_only** (*bool*) – If true, all instances in the cluster will only have internal IP addresses. This can only be enabled for subnetwork enabled networks
- **tags** (*list[string]*) – The GCE tags to add to all instances
- **region** – leave as 'global', might become relevant in the future. (templated)

- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **service_account** (*string*) – The service account of the dataproc instances.
- **service_account_scopes** (*list[string]*) – The URIs of service account scopes to be included.
- **idle_delete_ttl** (*int*) – The longest duration that cluster would keep alive while staying idle. Passing this threshold will cause cluster to be auto-deleted. A duration in seconds.
- **auto_delete_time** (*datetime.datetime*) – The time when cluster will be auto-deleted.
- **auto_delete_ttl** (*int*) – The life duration of cluster, the cluster will be auto-deleted at the end of this duration. A duration in seconds. (If auto_delete_time is set this parameter will be ignored)

DataprocClusterScaleOperator

class airflow.contrib.operators.dataproc_operator.**DataprocClusterScaleOperator**(*cluster_name*, *project_id*, *region='global'*, *gcp_conn_id='g*
delete_to=None, *num_workers=2*, *num_preemptible*
graceful_decommissio
**args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Scale, up or down, a cluster on Google Cloud Dataproc. The operator will wait until the cluster is re-scaled.

Example:

t1 = DataprocClusterScaleOperator(task_id='dataproc_scale', project_id='my-project', cluster_name='cluster-1', num_workers=10, num_preemptible_workers=10, graceful_decommission_timeout='1h' dag=dag)

See also:

For more detail on about scaling clusters have a look at the reference: https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/scaling-clusters

Parameters

- **cluster_name** (*string*) – The name of the cluster to scale. (templated)
- **project_id** (*string*) – The ID of the google cloud project in which the cluster runs. (templated)

- **region** (*string*) – The region for the dataproc cluster. (templated)
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **num_workers** (*int*) – The new number of workers
- **num_preemptible_workers** (*int*) – The new number of preemptible workers
- **graceful_decommission_timeout** (*string*) – Timeout for graceful YARN decomissioning. Maximum value is 1d
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

DataprocClusterDeleteOperator

class airflow.contrib.operators.dataproc_operator.**DataprocClusterDeleteOperator**(*cluster_name*, *project_id*, *region='global'*, *gcp_conn_id='*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Delete a cluster on Google Cloud Dataproc. The operator will wait until the cluster is destroyed.

Parameters

- **cluster_name** (*string*) – The name of the cluster to create. (templated)
- **project_id** (*string*) – The ID of the google cloud project in which the cluster runs. (templated)
- **region** (*string*) – leave as 'global', might become relevant in the future. (templated)
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

DataProcPigOperator

class airflow.contrib.operators.dataproc_operator.**DataProcPigOperator**(*query=None*, *query_uri=None*, *variables=None*, *job_name='{{task.task_id}}_{* *cluster_name='cluster-1'*, *dataproc_pig_properties=None*, *dataproc_pig_jars=None*, *gcp_conn_id='google_cloud_* *delegate_to=None*, *region='global'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Pig query Job on a Cloud DataProc cluster. The parameters of the operation will be passed to the cluster.

It's a good practice to define dataproc_* parameters in the default_args of the dag like the cluster name and UDFs.

```
default_args = {
    'cluster_name': 'cluster-1',
    'dataproc_pig_jars': [
        'gs://example/udf/jar/datafu/1.2.0/datafu.jar',
        'gs://example/udf/jar/gpig/1.2/gpig.jar'
    ]
}
```

You can pass a pig script as string or file reference. Use variables to pass on variables for the pig script to be resolved on the cluster or use the parameters to be resolved in the script as template parameters.

Example:

```
t1 = DataProcPigOperator(
        task_id='dataproc_pig',
        query='a_pig_script.pig',
        variables={'out': 'gs://example/output/{{ds}}'},
        dag=dag)
```

See also:

For more detail on about job submission have a look at the reference: https://cloud.google.com/dataproc/reference/rest/v1/projects.regions.jobs

Parameters

- **query** (*string*) – The query or reference to the query file (pg or pig extension). (templated)
- **query_uri** (*string*) – The uri of a pig script on Cloud Storage.
- **variables** (*dict*) – Map of named parameters for the query. (templated)

- **job_name** (*string*) – The job name used in the DataProc cluster. This name by default is the task_id appended with the execution data, but can be templated. The name will always be appended with a random number to avoid name clashes. (templated)
- **cluster_name** (*string*) – The name of the DataProc cluster. (templated)
- **dataproc_pig_properties** (*dict*) – Map for the Pig properties. Ideal to put in default arguments
- **dataproc_pig_jars** (*list*) – URIs to jars provisioned in Cloud Storage (example: for UDFs and libs) and are ideal to put in default arguments.
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **region** (*string*) – The specified region where the dataproc cluster is created.

DataProcHiveOperator

class airflow.contrib.operators.dataproc_operator.**DataProcHiveOperator**(*query=None*, *query_uri=None*, *variables=None*, *job_name='{{task.task_id}}_*, *cluster_name='cluster-1'*, *dataproc_hive_properties=None*, *dataproc_hive_jars=None*, *gcp_conn_id='google_cloud*, *delegate_to=None*, *region='global'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Hive query Job on a Cloud DataProc cluster.

Parameters

- **query** (*string*) – The query or reference to the query file (q extension).
- **query_uri** (*string*) – The uri of a hive script on Cloud Storage.
- **variables** (*dict*) – Map of named parameters for the query.
- **job_name** (*string*) – The job name used in the DataProc cluster. This name by default is the task_id appended with the execution data, but can be templated. The name will always be appended with a random number to avoid name clashes.
- **cluster_name** (*string*) – The name of the DataProc cluster.

- **dataproc_hive_properties** (*dict*) – Map for the Pig properties. Ideal to put in default arguments
- **dataproc_hive_jars** (*list*) – URIs to jars provisioned in Cloud Storage (example: for UDFs and libs) and are ideal to put in default arguments.
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **region** (*string*) – The specified region where the dataproc cluster is created.

DataProcSparkSqlOperator

class airflow.contrib.operators.dataproc_operator.**DataProcSparkSqlOperator**(*query=None*, *query_uri=None*, *variables=None*, *job_name='{{task.task*, *cluster_name='cluster-1'*, *dataproc_spark_propertie*, *dataproc_spark_jars=Non*, *gcp_conn_id='google_*, *delegate_to=None*, *region='global'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Spark SQL query Job on a Cloud DataProc cluster.

Parameters

- **query** (*string*) – The query or reference to the query file (q extension). (templated)
- **query_uri** (*string*) – The uri of a spark sql script on Cloud Storage.
- **variables** (*dict*) – Map of named parameters for the query. (templated)
- **job_name** (*string*) – The job name used in the DataProc cluster. This name by default is the task_id appended with the execution data, but can be templated. The name will always be appended with a random number to avoid name clashes. (templated)
- **cluster_name** (*string*) – The name of the DataProc cluster. (templated)
- **dataproc_spark_properties** (*dict*) – Map for the Pig properties. Ideal to put in default arguments
- **dataproc_spark_jars** (*list*) – URIs to jars provisioned in Cloud Storage (example: for UDFs and libs) and are ideal to put in default arguments.

- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **region** (*string*) – The specified region where the dataproc cluster is created.

DataProcSparkOperator

class airflow.contrib.operators.dataproc_operator.**DataProcSparkOperator**(*main_jar=None*, *main_class=None*, *arguments=None*, *archives=None*, *files=None*, *job_name='{{task.task_id}}*, *cluster_name='cluster-1'*, *dataproc_spark_properties=N*, *dataproc_spark_jars=None*, *gcp_conn_id='google_clou*, *delegate_to=None*, *region='global'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Spark Job on a Cloud DataProc cluster.

Parameters

- **main_jar** (*string*) – URI of the job jar provisioned on Cloud Storage. (use this or the main_class, not both together).
- **main_class** (*string*) – Name of the job class. (use this or the main_jar, not both together).
- **arguments** (*list*) – Arguments for the job. (templated)
- **archives** (*list*) – List of archived files that will be unpacked in the work directory. Should be stored in Cloud Storage.
- **files** (*list*) – List of files to be copied to the working directory
- **job_name** (*string*) – The job name used in the DataProc cluster. This name by default is the task_id appended with the execution data, but can be templated. The name will always be appended with a random number to avoid name clashes. (templated)
- **cluster_name** (*string*) – The name of the DataProc cluster. (templated)
- **dataproc_spark_properties** (*dict*) – Map for the Pig properties. Ideal to put in default arguments

- **dataproc_spark_jars** (*list*) – URIs to jars provisioned in Cloud Storage (example: for UDFs and libs) and are ideal to put in default arguments.
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **region** (*string*) – The specified region where the dataproc cluster is created.

DataProcHadoopOperator

class airflow.contrib.operators.dataproc_operator.**DataProcHadoopOperator**(*main_jar=None*, *main_class=None*, *arguments=None*, *archives=None*, *files=None*, *job_name='{{task.task_id*, *cluster_name='cluster-1'*, *dataproc_hadoop_properties*, *dataproc_hadoop_jars=None*, *gcp_conn_id='google_cl*, *delegate_to=None*, *region='global'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Hadoop Job on a Cloud DataProc cluster.

Parameters

- **main_jar** (*string*) – URI of the job jar provisioned on Cloud Storage. (use this or the main_class, not both together).
- **main_class** (*string*) – Name of the job class. (use this or the main_jar, not both together).
- **arguments** (*list*) – Arguments for the job. (templated)
- **archives** (*list*) – List of archived files that will be unpacked in the work directory. Should be stored in Cloud Storage.
- **files** (*list*) – List of files to be copied to the working directory
- **job_name** (*string*) – The job name used in the DataProc cluster. This name by default is the task_id appended with the execution data, but can be templated. The name will always be appended with a random number to avoid name clashes. (templated)
- **cluster_name** (*string*) – The name of the DataProc cluster. (templated)

- **dataproc_hadoop_properties** (*dict*) – Map for the Pig properties. Ideal to put in default arguments
- **dataproc_hadoop_jars** (*list*) – URIs to jars provisioned in Cloud Storage (example: for UDFs and libs) and are ideal to put in default arguments.
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **region** (*string*) – The specified region where the dataproc cluster is created.

DataProcPySparkOperator

class airflow.contrib.operators.dataproc_operator.**DataProcPySparkOperator**(*main*, *arguments=None*, *archives=None*, *pyfiles=None*, *files=None*, *job_name='{{task.task_*, *cluster_name='cluster-1'*, *dataproc_pyspark_properti*, *dataproc_pyspark_jars=No*, *gcp_conn_id='google_c*, *delegate_to=None*, *region='global'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a PySpark Job on a Cloud DataProc cluster.

Parameters

- **main** (*string*) – [Required] The Hadoop Compatible Filesystem (HCFS) URI of the main Python file to use as the driver. Must be a .py file.
- **arguments** (*list*) – Arguments for the job. (templated)
- **archives** (*list*) – List of archived files that will be unpacked in the work directory. Should be stored in Cloud Storage.
- **files** (*list*) – List of files to be copied to the working directory
- **pyfiles** (*list*) – List of Python files to pass to the PySpark framework. Supported file types: .py, .egg, and .zip

- **job_name** (*string*) – The job name used in the DataProc cluster. This name by default is the task_id appended with the execution data, but can be templated. The name will always be appended with a random number to avoid name clashes. (templated)
- **cluster_name** (*string*) – The name of the DataProc cluster.
- **dataproc_pyspark_properties** (*dict*) – Map for the Pig properties. Ideal to put in default arguments
- **dataproc_pyspark_jars** (*list*) – URIs to jars provisioned in Cloud Storage (example: for UDFs and libs) and are ideal to put in default arguments.
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **region** (*string*) – The specified region where the dataproc cluster is created.

DataprocWorkflowTemplateInstantiateOperator

class airflow.contrib.operators.dataproc_operator.**DataprocWorkflowTemplateInstantiateOperat**

Bases: *airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateBaseOperator*

Instantiate a WorkflowTemplate on Google Cloud Dataproc. The operator will wait until the WorkflowTemplate is finished executing.

See also:

Please refer to: https://cloud.google.com/dataproc/docs/reference/rest/v1beta2/projects.regions.workflowTemplates/instantiate

Parameters

- **template_id** (*string*) – The id of the template. (templated)
- **project_id** (*string*) – The ID of the google cloud project in which the template runs
- **region** (*string*) – leave as 'global', might become relevant in the future
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

DataprocWorkflowTemplateInstantiateInlineOperator

class airflow.contrib.operators.dataproc_operator.**DataprocWorkflowTemplateInstantiateInline**

Bases: *airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateBaseOperator*

Instantiate a WorkflowTemplate Inline on Google Cloud Dataproc. The operator will wait until the WorkflowTemplate is finished executing.

See also:

Please refer to: https://cloud.google.com/dataproc/docs/reference/rest/v1beta2/projects.regions.workflowTemplates/instantiateInline

Parameters

- **template** (*map*) – The template contents. (templated)
- **project_id** (*string*) – The ID of the google cloud project in which the template runs
- **region** (*string*) – leave as 'global', might become relevant in the future
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

3.16.5.5 Cloud Datastore

Datastore Operators

- *DatastoreExportOperator* : Export entities from Google Cloud Datastore to Cloud Storage.
- *DatastoreImportOperator* : Import entities from Cloud Storage to Google Cloud Datastore.

DatastoreExportOperator

class airflow.contrib.operators.datastore_export_operator.**DatastoreExportOperator**(*bucket*, *namespace=None*, *datastore_conn_id*, *cloud_storage*, *delegate_to=None*, *entity_filter=N*, *labels=None*, *polling_inter*, *overwrite_existin*, *xcom_push=*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Export entities from Google Cloud Datastore to Cloud Storage

Parameters

- **bucket** (*string*) – name of the cloud storage bucket to backup data
- **namespace** (*str*) – optional namespace path in the specified Cloud Storage bucket to backup data. If this namespace does not exist in GCS, it will be created.

- **datastore_conn_id** (*string*) – the name of the Datastore connection id to use
- **cloud_storage_conn_id** (*string*) – the name of the cloud storage connection id to force-write backup
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **entity_filter** (*dict*) – description of what data from the project is included in the export, refer to https://cloud.google.com/datastore/docs/reference/rest/Shared.Types/EntityFilter
- **labels** (*dict*) – client-assigned labels for cloud storage
- **polling_interval_in_seconds** (*int*) – number of seconds to wait before polling for execution status again
- **overwrite_existing** (*bool*) – if the storage bucket + namespace is not empty, it will be emptied prior to exports. This enables overwriting existing backups.
- **xcom_push** (*bool*) – push operation name to xcom for reference

DatastoreImportOperator

class airflow.contrib.operators.datastore_import_operator.**DatastoreImportOperator**(*bucket*, *file*, *namespace=None*, *entity_filter=N*, *labels=None*, *datastore_conn_id*, *delegate_to=No*, *polling_inter*, *xcom_push=*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Import entities from Cloud Storage to Google Cloud Datastore

Parameters

- **bucket** (*string*) – container in Cloud Storage to store data
- **file** (*string*) – path of the backup metadata file in the specified Cloud Storage bucket. It should have the extension .overall_export_metadata
- **namespace** (*str*) – optional namespace of the backup metadata file in the specified Cloud Storage bucket.
- **entity_filter** (*dict*) – description of what data from the project is included in the export, refer to https://cloud.google.com/datastore/docs/reference/rest/Shared.Types/EntityFilter

- **labels** (*dict*) – client-assigned labels for cloud storage
- **datastore_conn_id** (*string*) – the name of the connection id to use
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **polling_interval_in_seconds** (*int*) – number of seconds to wait before polling for execution status again
- **xcom_push** (*bool*) – push operation name to xcom for reference

DatastoreHook

class airflow.contrib.hooks.datastore_hook.**DatastoreHook**(*datastore_conn_id='google_cloud_datastore_def*
delegate_to=None)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Interact with Google Cloud Datastore. This hook uses the Google Cloud Platform connection.

This object is not threads safe. If you want to make multiple requests simultaneously, you will need to create a hook per thread.

allocate_ids(*partialKeys*)

Allocate IDs for incomplete keys. see https://cloud.google.com/datastore/docs/reference/rest/v1/projects/allocateIds

Parameters **partialKeys** – a list of partial keys

Returns a list of full keys.

begin_transaction()

Get a new transaction handle

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/beginTransaction

Returns a transaction handle

commit(*body*)

Commit a transaction, optionally creating, deleting or modifying some entities.

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/commit

Parameters **body** – the body of the commit request

Returns the response body of the commit request

delete_operation(*name*)

Deletes the long-running operation

Parameters **name** – the name of the operation resource

export_to_storage_bucket(*bucket*, *namespace=None*, *entity_filter=None*, *labels=None*)

Export entities from Cloud Datastore to Cloud Storage for backup

get_conn(*version='v1'*)

Returns a Google Cloud Storage service object.

get_operation(*name*)
Gets the latest state of a long-running operation

Parameters **name** – the name of the operation resource

import_from_storage_bucket(*bucket*, *file*, *namespace=None*, *entity_filter=None*, *labels=None*)
Import a backup from Cloud Storage to Cloud Datastore

lookup(*keys*, *read_consistency=None*, *transaction=None*)
Lookup some entities by key

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/lookup

Parameters

- **keys** – the keys to lookup
- **read_consistency** – the read consistency to use. default, strong or eventual. Cannot be used with a transaction.
- **transaction** – the transaction to use, if any.

Returns the response body of the lookup request.

poll_operation_until_done(*name*, *polling_interval_in_seconds*)
Poll backup operation state until it's completed

rollback(*transaction*)
Roll back a transaction

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/rollback

Parameters **transaction** – the transaction to roll back

run_query(*body*)
Run a query for entities.

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/runQuery

Parameters **body** – the body of the query request

Returns the batch of query results.

3.16.5.6 Cloud ML Engine

Cloud ML Engine Operators

- *MLEngineBatchPredictionOperator* : Start a Cloud ML Engine batch prediction job.
- *MLEngineModelOperator* : Manages a Cloud ML Engine model.
- *MLEngineTrainingOperator* : Start a Cloud ML Engine training job.
- *MLEngineVersionOperator* : Manages a Cloud ML Engine model version.

MLEngineBatchPredictionOperator

class airflow.contrib.operators.mlengine_operator.**MLEngineBatchPredictionOperator**(*project_id*, *job_id*, *region*, *data_format*, *input_paths*, *output_path*, *model_name*, *version_name=*, *uri=None*, *max_worker*, *runtime_version*, *gcp_conn_id*, *delegate_to=No*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Google Cloud ML Engine prediction job.

NOTE: For model origin, users should consider exactly one from the three options below: 1. Populate 'uri' field only, which should be a GCS location that points to a tensorflow savedModel directory. 2. Populate 'model_name' field only, which refers to an existing model, and the default version of the model will be used. 3. Populate both 'model_name' and 'version_name' fields, which refers to a specific version of a specific model.

In options 2 and 3, both model and version name should contain the minimal identifier. For instance, call

```
MLEngineBatchPredictionOperator(
    ...,
    model_name='my_model',
    version_name='my_version',
    ...)
```

if the desired model version is "projects/my_project/models/my_model/versions/my_version".

See https://cloud.google.com/ml-engine/reference/rest/v1/projects.jobs for further documentation on the parameters.

Parameters

- **project_id** (*string*) – The Google Cloud project name where the prediction job is submitted. (templated)
- **job_id** (*string*) – A unique id for the prediction job on Google Cloud ML Engine. (templated)
- **data_format** (*string*) – The format of the input data. It will default to 'DATA_FORMAT_UNSPECIFIED' if is not provided or is not one of ["TEXT", "TF_RECORD", "TF_RECORD_GZIP"].

- **input_paths** (*list of string*) – A list of GCS paths of input data for batch prediction. Accepting wildcard operator *, but only at the end. (templated)
- **output_path** (*string*) – The GCS path where the prediction results are written to. (templated)
- **region** (*string*) – The Google Compute Engine region to run the prediction job in. (templated)
- **model_name** (*string*) – The Google Cloud ML Engine model to use for prediction. If version_name is not provided, the default version of this model will be used. Should not be None if version_name is provided. Should be None if uri is provided. (templated)
- **version_name** (*string*) – The Google Cloud ML Engine model version to use for prediction. Should be None if uri is provided. (templated)
- **uri** (*string*) – The GCS path of the saved model to use for prediction. Should be None if model_name is provided. It should be a GCS path pointing to a tensorflow SavedModel. (templated)
- **max_worker_count** (*int*) – The maximum number of workers to be used for parallel processing. Defaults to 10 if not specified.
- **runtime_version** (*string*) – The Google Cloud ML Engine runtime version to use for batch prediction.
- **gcp_conn_id** (*string*) – The connection ID used for connection to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have doamin-wide delegation enabled.

Raises: ValueError: if a unique model/version origin cannot be determined.

MLEngineModelOperator

class airflow.contrib.operators.mlengine_operator.**MLEngineModelOperator**(*project_id*, *model*, *operation='create'*, *gcp_conn_id='google_clou*
delegate_to=None, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Operator for managing a Google Cloud ML Engine model.

Parameters

- **project_id** (*string*) – The Google Cloud project name to which MLEngine model belongs. (templated)
- **model** (*dict*) – A dictionary containing the information about the model. If the *operation* is *create*, then the *model* parameter should contain all the information about this model such as *name*.

If the *operation* is *get*, the *model* parameter should contain the *name* of the model.

- **operation** – The operation to perform. Available operations are:
  - `create`: Creates a new model as provided by the *model* parameter.
  - `get`: Gets a particular model where the name is specified in *model*.
- **gcp_conn_id** (*string*) – The connection ID to use when fetching connection info.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

MLEngineTrainingOperator

class airflow.contrib.operators.mlengine_operator.**MLEngineTrainingOperator**(*project_id*, *job_id*, *package_uris*, *training_python_module*, *training_args*, *region*, *scale_tier=None*, *runtime_version=None*, *python_version=None*, *job_dir=None*, *gcp_conn_id='google_*, *delegate_to=None*, *mode='PRODUCTION*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Operator for launching a MLEngine training job.

Parameters

- **project_id** (*string*) – The Google Cloud project name within which MLEngine training job should run (templated).
- **job_id** (*string*) – A unique templated id for the submitted Google MLEngine training job. (templated)
- **package_uris** (*string*) – A list of package locations for MLEngine training job, which should include the main training program + any additional dependencies. (templated)
- **training_python_module** (*string*) – The Python module name to run within MLEngine training job after installing 'package_uris' packages. (templated)
- **training_args** (*string*) – A list of templated command line arguments to pass to the MLEngine training program. (templated)
- **region** (*string*) – The Google Compute Engine region to run the MLEngine training job in (templated).

- **scale_tier** (*string*) – Resource tier for MLEngine training job. (templated)
- **runtime_version** (*string*) – The Google Cloud ML runtime version to use for training. (templated)
- **python_version** (*string*) – The version of Python used in training. (templated)
- **job_dir** (*string*) – A Google Cloud Storage path in which to store training outputs and other data needed for training. (templated)
- **gcp_conn_id** (*string*) – The connection ID to use when fetching connection info.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **mode** (*string*) – Can be one of 'DRY_RUN'/'CLOUD'. In 'DRY_RUN' mode, no real training job will be launched, but the MLEngine training job request will be printed out. In 'CLOUD' mode, a real MLEngine training job creation request will be issued.

MLEngineVersionOperator

class airflow.contrib.operators.mlengine_operator.**MLEngineVersionOperator**(*project_id*, *model_name*, *version_name=None*, *version=None*, *operation='create'*, *gcp_conn_id='google_c*
delegate_to=None, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Operator for managing a Google Cloud ML Engine version.

Parameters

- **project_id** (*string*) – The Google Cloud project name to which MLEngine model belongs.
- **model_name** (*string*) – The name of the Google Cloud ML Engine model that the version belongs to. (templated)
- **version_name** (*string*) – A name to use for the version being operated upon. If not None and the *version* argument is None or does not have a value for the *name* key, then this will be populated in the payload for the *name* key. (templated)
- **version** (*dict*) – A dictionary containing the information about the version. If the *operation* is *create*, *version* should contain all the information about this version such as name, and deploymentUrl. If the *operation* is *get* or *delete*, the *version* parameter should contain the *name* of the version. If it is None, the only *operation* possible would be *list*. (templated)
- **operation** (*string*) – The operation to perform. Available operations are:

- `create`: Creates a new version in the model specified by *model_name*, in which case the *version* parameter should contain all the information to create that version (e.g. *name*, *deploymentUrl*).
- `get`: Gets full information of a particular version in the model specified by *model_name*. The name of the version should be specified in the *version* parameter.
- `list`: Lists all available versions of the model specified by *model_name*.
- `delete`: Deletes the version specified in *version* parameter from the model specified by *model_name*). The name of the version should be specified in the *version* parameter.

- **gcp_conn_id** (*string*) – The connection ID to use when fetching connection info.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

Cloud ML Engine Hook

MLEngineHook

class airflow.contrib.hooks.gcp_mlengine_hook.**MLEngineHook**(*gcp_conn_id='google_cloud_default'*, *delegate_to=None*)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

create_job(*project_id*, *job*, *use_existing_job_fn=None*)

Launches a MLEngine job and wait for it to reach a terminal state.

Parameters

- **project_id** (*string*) – The Google Cloud project id within which MLEngine job will be launched.
- **job** (*dict*) – MLEngine Job object that should be provided to the MLEngine API, such as:

```
{
  'jobId': 'my_job_id',
  'trainingInput': {
    'scaleTier': 'STANDARD_1',
    ...
  }
}
```

- **use_existing_job_fn** (*function*) – In case that a MLEngine job with the same job_id already exist, this method (if provided) will decide whether we should use this existing job, continue waiting for it to finish and returning the job object. It should accepts a MLEngine job object, and returns a boolean value indicating whether it is OK to reuse the existing job. If 'use_existing_job_fn' is not provided, we by default reuse the existing MLEngine job.

Returns The MLEngine job object if the job successfully reach a terminal state (which might be FAILED or CANCELLED state).

Return type dict

create_model(*project_id*, *model*)

Create a Model. Blocks until finished.

create_version(*project_id*, *model_name*, *version_spec*)
Creates the Version on Google Cloud ML Engine.

Returns the operation if the version was created successfully and raises an error otherwise.

delete_version(*project_id*, *model_name*, *version_name*)
Deletes the given version of a model. Blocks until finished.

get_conn()
Returns a Google MLEngine service object.

get_model(*project_id*, *model_name*)
Gets a Model. Blocks until finished.

list_versions(*project_id*, *model_name*)
Lists all available versions of a model. Blocks until finished.

set_default_version(*project_id*, *model_name*, *version_name*)
Sets a version to be the default. Blocks until finished.

3.16.5.7 Cloud Storage

Storage Operators

- *FileToGoogleCloudStorageOperator* : Uploads a file to Google Cloud Storage.
- *GoogleCloudStorageCreateBucketOperator* : Creates a new cloud storage bucket.
- *GoogleCloudStorageListOperator* : List all objects from the bucket with the give string prefix and delimiter in name.
- *GoogleCloudStorageDownloadOperator* : Downloads a file from Google Cloud Storage.
- *GoogleCloudStorageToBigQueryOperator* : Loads files from Google cloud storage into BigQuery.
- *GoogleCloudStorageToGoogleCloudStorageOperator* : Copies objects from a bucket to another, with renaming if requested.

FileToGoogleCloudStorageOperator

class airflow.contrib.operators.file_to_gcs.**FileToGoogleCloudStorageOperator**(*src*, *dst*, *bucket*, *google_cloud_stora*, *mime_type='applic*, *stream'*, *del-*, *e-*, *gate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Uploads a file to Google Cloud Storage

Parameters

- **src** (*string*) – Path to the local file. (templated)
- **dst** (*string*) – Destination path within the specified bucket. (templated)

- **bucket** (*string*) – The bucket to upload to. (templated)
- **google_cloud_storage_conn_id** (*string*) – The Airflow connection ID to upload with
- **mime_type** (*string*) – The mime-type string
- **delegate_to** (*string*) – The account to impersonate, if any

execute(*context*)
Uploads the file to Google cloud storage

GoogleCloudStorageCreateBucketOperator

class airflow.contrib.operators.gcs_operator.**GoogleCloudStorageCreateBucketOperator**(*bucket_na stor- age_class lo- ca- tion='US project_id la- bels=Non google_cl del- e- gate_to=N *args, **kwargs*

Bases: *airflow.models.BaseOperator*

Creates a new bucket. Google Cloud Storage uses a flat namespace, so you can't create a bucket with a name that is already in use.

See also:

For more information, see Bucket Naming Guidelines: https://cloud.google.com/storage/docs/bucketnaming.html#requirements

Parameters

- **bucket_name** (*string*) – The name of the bucket. (templated)
- **storage_class** (*string*) – This defines how objects in the bucket are stored and determines the SLA and the cost of storage (templated). Values include
  - MULTI_REGIONAL
  - REGIONAL
  - STANDARD
  - NEARLINE
  - COLDLINE.

  If this value is not specified when the bucket is created, it will default to STANDARD.
- **location** (*string*) – The location of the bucket. (templated) Object data for objects in the bucket resides in physical storage within this region. Defaults to US.

  See also:

https://developers.google.com/storage/docs/bucket-locations

- **project_id** (*string*) – The ID of the GCP Project. (templated)
- **labels** (*dict*) – User-provided labels, in key/value pairs.
- **google_cloud_storage_conn_id** (*string*) – The connection ID to use when connecting to Google cloud storage.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

Example: The following Operator would create a new bucket `test-bucket` with `MULTI_REGIONAL` storage class in `EU` region

```
CreateBucket = GoogleCloudStorageCreateBucketOperator(
    task_id='CreateNewBucket',
    bucket_name='test-bucket',
    storage_class='MULTI_REGIONAL',
    location='EU',
    labels={'env': 'dev', 'team': 'airflow'},
    google_cloud_storage_conn_id='airflow-service-account'
)
```

GoogleCloudStorageDownloadOperator

class airflow.contrib.operators.gcs_download_operator.**GoogleCloudStorageDownloadOperator**(*bu ob je fil na st go de e- go *a **

Bases: *airflow.models.BaseOperator*

Downloads a file from Google Cloud Storage.

Parameters

- **bucket** (*string*) – The Google cloud storage bucket where the object is. (templated)
- **object** (*string*) – The name of the object to download in the Google cloud storage bucket. (templated)
- **filename** (*string*) – The file path on the local file system (where the operator is being executed) that the file should be downloaded to. (templated) If no filename passed, the downloaded data will not be stored on the local file system.
- **store_to_xcom_key** (*string*) – If this param is set, the operator will push the contents of the downloaded file to XCom with the key set in this parameter. If not set, the downloaded data will not be pushed to XCom. (templated)
- **google_cloud_storage_conn_id** (*string*) – The connection ID to use when connecting to Google cloud storage.

- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

GoogleCloudStorageListOperator

class airflow.contrib.operators.gcs_list_operator.**GoogleCloudStorageListOperator**(*bucket*, *prefix=None*, *delimiter=None*, *google_cloud_*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

List all objects from the bucket with the give string prefix and delimiter in name.

This operator returns a python list with the name of objects which can be used by *xcom* in the downstream task.

Parameters

- **bucket** (*string*) – The Google cloud storage bucket to find the objects. (templated)
- **prefix** (*string*) – Prefix string which filters objects whose name begin with this prefix. (templated)
- **delimiter** (*string*) – The delimiter by which you want to filter the objects. (templated) For e.g to lists the CSV files from in a directory in GCS you would use delimiter='.csv'.
- **google_cloud_storage_conn_id** (*string*) – The connection ID to use when connecting to Google cloud storage.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

Example: The following Operator would list all the Avro files from `sales/sales-2017` folder in `data` bucket.

```
GCS_Files = GoogleCloudStorageListOperator(
    task_id='GCS_Files',
    bucket='data',
    prefix='sales/sales-2017/',
    delimiter='.avro',
    google_cloud_storage_conn_id=google_cloud_conn_id
)
```

GoogleCloudStorageToBigQueryOperator

class airflow.contrib.operators.gcs_to_bq.**GoogleCloudStorageToBigQueryOperator**(*bucket*, *source_objects*, *destination_project_dat schema_fields=N schema_object=I source_format=' compression='NONE', create_disposition= skip_leading_ro write_disposition field_delimiter=' ', max_bad_record quote_character= ignore_unknown_v allow_quoted_new allow_jagged_row max_id_key=Non bigquery_conn_id= google_cloud_st delegate_to=None, schema_update_ src_fmt_configs= external_table=False, time_partitioning *args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Loads files from Google cloud storage into BigQuery.

The schema to be used for the BigQuery table may be specified in one of two ways. You may either directly pass the schema fields in, or you may point the operator to a Google cloud storage object name. The object in Google cloud storage must be a JSON file with the schema fields in it.

Parameters

- **bucket** (*string*) – The bucket to load from. (templated)
- **source_objects** – List of Google cloud storage URIs to load from. (templated) If

source_format is 'DATASTORE_BACKUP', the list must only contain a single URI.

- **destination_project_dataset_table** (*string*) – The dotted (<project>.)<dataset>.<table> BigQuery table to load data into. If <project> is not included, project will be the project defined in the connection json. (templated)
- **schema_fields** (*list*) – If set, the schema field list as defined here: https://cloud.google.com/bigquery/docs/reference/v2/jobs#configuration.load Should not be set when source_format is 'DATASTORE_BACKUP'.
- **schema_object** – If set, a GCS object path pointing to a .json file that contains the schema for the table. (templated)
- **schema_object** – string
- **source_format** (*string*) – File format to export.
- **compression** (*string*) – [Optional] The compression type of the data source. Possible values include GZIP and NONE. The default value is NONE. This setting is ignored for Google Cloud Bigtable, Google Cloud Datastore backups and Avro formats.
- **create_disposition** (*string*) – The create disposition if the table doesn't exist.
- **skip_leading_rows** (*int*) – Number of rows to skip when loading from a CSV.
- **write_disposition** (*string*) – The write disposition if the table already exists.
- **field_delimiter** (*string*) – The delimiter to use when loading from a CSV.
- **max_bad_records** (*int*) – The maximum number of bad records that BigQuery can ignore when running the job.
- **quote_character** (*string*) – The value that is used to quote data sections in a CSV file.
- **ignore_unknown_values** (*bool*) – [Optional] Indicates if BigQuery should allow extra values that are not represented in the table schema. If true, the extra values are ignored. If false, records with extra columns are treated as bad records, and if there are too many bad records, an invalid error is returned in the job result.
- **allow_quoted_newlines** (*boolean*) – Whether to allow quoted newlines (true) or not (false).
- **allow_jagged_rows** (*bool*) – Accept rows that are missing trailing optional columns. The missing values are treated as nulls. If false, records with missing trailing columns are treated as bad records, and if there are too many bad records, an invalid error is returned in the job result. Only applicable to CSV, ignored for other formats.
- **max_id_key** (*string*) – If set, the name of a column in the BigQuery table that's to be loaded. Thsi will be used to select the MAX value from BigQuery after the load occurs. The results will be returned by the execute() command, which in turn gets stored in XCom for future operators to use. This can be helpful with incremental loads–during future executions, you can pick up from the max ID.
- **bigquery_conn_id** (*string*) – Reference to a specific BigQuery hook.
- **google_cloud_storage_conn_id** (*string*) – Reference to a specific Google cloud storage hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **schema_update_options** (*list*) – Allows the schema of the destination table to be updated as a side effect of the load job.

- **src_fmt_configs** (*dict*) – configure optional fields specific to the source format
- **external_table** (*bool*) – Flag to specify if the destination table should be a BigQuery external table. Default Value is False.
- **time_partitioning** (*dict*) – configure optional time partitioning fields i.e. partition by field, type and expiration as per API specifications. Note that 'field' is not available in concurrency with dataset.table$partition.

GoogleCloudStorageToGoogleCloudStorageOperator

class airflow.contrib.operators.gcs_to_gcs.**GoogleCloudStorageToGoogleCloudStorageOperator**(

Bases: *airflow.models.BaseOperator*

Copies objects from a bucket to another, with renaming if requested.

Parameters

- **source_bucket** (*string*) – The source Google cloud storage bucket where the object is. (templated)
- **source_object** (*string*) – The source name of the object to copy in the Google cloud storage bucket. (templated) If wildcards are used in this argument:

  You can use only one wildcard for objects (filenames) within your bucket. The wildcard can appear inside the object name or at the end of the object name. Appending a wildcard to the bucket name is unsupported.
- **destination_bucket** – The destination Google cloud storage bucket

where the object should be. (templated) :type destination_bucket: string :param destination_object: The destination name of the object in the

destination Google cloud storage bucket. (templated) If a wildcard is supplied in the source_object argument, this is the prefix that will be prepended to the final destination objects' paths. Note that the source path's part before the wildcard will be removed; if it needs to be retained it should be appended to destination_object. For example, with prefix `foo/*` and destination_object *'blah/'*, the file `foo/baz` will be copied to `blah/baz`; to retain the prefix write the destination_object as e.g. `blah/foo`, in which case the copied file will be named `blah/foo/baz`.

Parameters **move_object** – When move object is True, the object is moved instead

of copied to the new location. This is the equivalent of a mv command as opposed to a cp command.

Parameters

- **google_cloud_storage_conn_id** (*string*) – The connection ID to use when connecting to Google cloud storage.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

Examples: The following Operator would copy a single file named `sales/sales-2017/january.avro` in the `data` bucket to the file named `copied_sales/2017/january-backup.avro` in the `data_backup` bucket

```
copy_single_file = GoogleCloudStorageToGoogleCloudStorageOperator(
    task_id='copy_single_file',
    source_bucket='data',
    source_object='sales/sales-2017/january.avro',
    destination_bucket='data_backup',
    destination_object='copied_sales/2017/january-backup.avro',
    google_cloud_storage_conn_id=google_cloud_conn_id
)
```

The following Operator would copy all the Avro files from `sales/sales-2017` folder (i.e. with names starting with that prefix) in `data` bucket to the `copied_sales/2017` folder in the `data_backup` bucket.

```
copy_files = GoogleCloudStorageToGoogleCloudStorageOperator(
    task_id='copy_files',
    source_bucket='data',
    source_object='sales/sales-2017/*.avro',
    destination_bucket='data_backup',
    destination_object='copied_sales/2017/',
    google_cloud_storage_conn_id=google_cloud_conn_id
)
```

The following Operator would move all the Avro files from `sales/sales-2017` folder (i.e. with names starting with that prefix) in `data` bucket to the same folder in the `data_backup` bucket, deleting the original files in the process.

```
move_files = GoogleCloudStorageToGoogleCloudStorageOperator(
    task_id='move_files',
    source_bucket='data',
    source_object='sales/sales-2017/*.avro',
    destination_bucket='data_backup',
    move_object=True,
    google_cloud_storage_conn_id=google_cloud_conn_id
)
```

GoogleCloudStorageHook

class airflow.contrib.hooks.gcs_hook.**GoogleCloudStorageHook**(*google_cloud_storage_conn_id='google_clo*
delegate_to=None)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Interact with Google Cloud Storage. This hook uses the Google Cloud Platform connection.

copy(*source_bucket*, *source_object*, *destination_bucket=None*, *destination_object=None*)
Copies an object from a bucket to another, with renaming if requested.

destination_bucket or destination_object can be omitted, in which case source bucket/object is used, but not both.

Parameters

- **source_bucket** (*string*) – The bucket of the object to copy from.
- **source_object** (*string*) – The object to copy.
- **destination_bucket** (*string*) – The destination of the object to copied to. Can be omitted; then the same bucket is used.
- **destination_object** – The (renamed) path of the object if given. Can be omitted; then the same name is used.

create_bucket(*bucket_name*, *storage_class='MULTI_REGIONAL'*, *location='US'*, *project_id=None*, *labels=None*)
Creates a new bucket. Google Cloud Storage uses a flat namespace, so you can't create a bucket with a name that is already in use.

See also:

For more information, see Bucket Naming Guidelines: https://cloud.google.com/storage/docs/bucketnaming.html#requirements

Parameters

- **bucket_name** (*string*) – The name of the bucket.
- **storage_class** (*string*) – This defines how objects in the bucket are stored and determines the SLA and the cost of storage. Values include
  - MULTI_REGIONAL
  - REGIONAL
  - STANDARD
  - NEARLINE
  - COLDLINE.

  If this value is not specified when the bucket is created, it will default to STANDARD.
- **location** (*string*) – The location of the bucket. Object data for objects in the bucket resides in physical storage within this region. Defaults to US.

  See also:

  https://developers.google.com/storage/docs/bucket-locations
- **project_id** (*string*) – The ID of the GCP Project.
- **labels** (*dict*) – User-provided labels, in key/value pairs.

Returns If successful, it returns the `id` of the bucket.

delete(*bucket*, *object*, *generation=None*)
Delete an object if versioning is not enabled for the bucket, or if generation parameter is used.

Parameters

- **bucket** (*string*) – name of the bucket, where the object resides

- **object** (*string*) – name of the object to delete
- **generation** (*string*) – if present, permanently delete the object of this generation

Returns True if succeeded

download(*bucket*, *object*, *filename=None*)
Get a file from Google Cloud Storage.

Parameters

- **bucket** (*string*) – The bucket to fetch from.
- **object** (*string*) – The object to fetch.
- **filename** (*string*) – If set, a local file path where the file should be written to.

exists(*bucket*, *object*)
Checks for the existence of a file in Google Cloud Storage.

Parameters

- **bucket** (*string*) – The Google cloud storage bucket where the object is.
- **object** (*string*) – The name of the object to check in the Google cloud storage bucket.

get_conn()
Returns a Google Cloud Storage service object.

get_crc32c(*bucket*, *object*)
Gets the CRC32c checksum of an object in Google Cloud Storage.

Parameters

- **bucket** (*string*) – The Google cloud storage bucket where the object is.
- **object** (*string*) – The name of the object to check in the Google cloud storage bucket.

get_md5hash(*bucket*, *object*)
Gets the MD5 hash of an object in Google Cloud Storage.

Parameters

- **bucket** (*string*) – The Google cloud storage bucket where the object is.
- **object** (*string*) – The name of the object to check in the Google cloud storage bucket.

get_size(*bucket*, *object*)
Gets the size of a file in Google Cloud Storage.

Parameters

- **bucket** (*string*) – The Google cloud storage bucket where the object is.
- **object** (*string*) – The name of the object to check in the Google cloud storage bucket.

is_updated_after(*bucket*, *object*, *ts*)
Checks if an object is updated in Google Cloud Storage.

Parameters

- **bucket** (*string*) – The Google cloud storage bucket where the object is.
- **object** (*string*) – The name of the object to check in the Google cloud storage bucket.
- **ts** (*datetime*) – The timestamp to check against.

list(*bucket*, *versions=None*, *maxResults=None*, *prefix=None*, *delimiter=None*)
List all objects from the bucket with the give string prefix in name

Parameters

- **bucket** (*string*) – bucket name
- **versions** (*boolean*) – if true, list all versions of the objects
- **maxResults** (*integer*) – max count of items to return in a single page of responses
- **prefix** (*string*) – prefix string which filters objects whose name begin with this prefix
- **delimiter** (*string*) – filters objects based on the delimiter (for e.g '.csv')

Returns a stream of object names matching the filtering criteria

rewrite(*source_bucket*, *source_object*, *destination_bucket*, *destination_object=None*)
Has the same functionality as copy, except that will work on files over 5 TB, as well as when copying between locations and/or storage classes.

destination_object can be omitted, in which case source_object is used.

Parameters

- **source_bucket** (*string*) – The bucket of the object to copy from.
- **source_object** (*string*) – The object to copy.
- **destination_bucket** (*string*) – The destination of the object to copied to.
- **destination_object** – The (renamed) path of the object if given. Can be omitted; then the same name is used.

upload(*bucket*, *object*, *filename*, *mime_type='application/octet-stream'*)
Uploads a local file to Google Cloud Storage.

Parameters

- **bucket** (*string*) – The bucket to upload to.
- **object** (*string*) – The object name to set when uploading the local file.
- **filename** (*string*) – The local file path to the file to be uploaded.
- **mime_type** (*string*) – The MIME type to set when uploading the file.

3.16.5.8 Google Kubernetes Engine

Google Kubernetes Engine Cluster Operators

- *GKEClusterDeleteOperator* : Creates a Kubernetes Cluster in Google Cloud Platform
- *Google Kubernetes Engine Hook* : Deletes a Kubernetes Cluster in Google Cloud Platform

GKEClusterCreateOperator

class airflow.contrib.operators.gcp_container_operator.**GKEClusterCreateOperator**(*project_id*, *location*, *body={}*, *gcp_conn_id='*, *api_version='v*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

GKEClusterDeleteOperator

class airflow.contrib.operators.gcp_container_operator.**GKEClusterDeleteOperator**(*project_id*, *name*, *location*, *gcp_conn_id='*, *api_version='v*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Google Kubernetes Engine Hook

class airflow.contrib.hooks.gcp_container_hook.**GKEClusterHook**(*project_id*, *location*)

Bases: airflow.hooks.base_hook.BaseHook

create_cluster(*cluster*, *retry=<object object>*, *timeout=<object object>*)

Creates a cluster, consisting of the specified number and type of Google Compute Engine instances.

Parameters

- **cluster** (*dict or google.cloud.container_v1.types.Cluster*) – A Cluster protobuf or dict. If dict is provided, it must be of the same form as the protobuf message google.cloud.container_v1.types.Cluster
- **retry** (*google.api_core.retry.Retry*) – A retry object (google.api_core.retry.Retry) used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.

Returns The full url to the new, or existing, cluster

:raises ParseError: On JSON parsing problems when trying to convert dict AirflowException: cluster is not dict type nor Cluster proto type

delete_cluster(*name*, *retry=<object object>*, *timeout=<object object>*)

Deletes the cluster, including the Kubernetes endpoint and all worker nodes. Firewalls and routes that were

configured during cluster creation are also deleted. Other Google Compute Engine resources that might be in use by the cluster (e.g. load balancer resources) will not be deleted if they weren't present at the initial create time.

Parameters

- **name** (*str*) – The name of the cluster to delete
- **retry** (*google.api_core.retry.Retry*) – Retry object used to determine when/if to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.

Returns The full url to the delete operation if successful, else None

get_cluster(*name*, *retry=<object object>*, *timeout=<object object>*)

Gets details of specified cluster :param name: The name of the cluster to retrieve :type name: str :param retry: A retry object used to retry requests. If None is specified,

requests will not be retried.

Parameters **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.

Returns A google.cloud.container_v1.types.Cluster instance

get_operation(*operation_name*)

Fetches the operation from Google Cloud :param operation_name: Name of operation to fetch :type operation_name: str :return: The new, updated operation from Google Cloud

wait_for_operation(*operation*)

Given an operation, continuously fetches the status from Google Cloud until either completion or an error occurring :param operation: The Operation to wait for :type operation: A google.cloud.container_V1.gapic.enums.Operator :return: A new, updated operation fetched from Google Cloud

3.17 Lineage

Note: Lineage support is very experimental and subject to change.

Airflow can help track origins of data, what happens to it and where it moves over time. This can aid having audit trails and data governance, but also debugging of data flows.

Airflow tracks data by means of inlets and outlets of the tasks. Let's work from an example and see how it works.

```
from airflow.operators.bash_operator import BashOperator
from airflow.operators.dummy_operator import DummyOperator
from airflow.lineage.datasets import File
from airflow.models import DAG
from datetime import timedelta

FILE_CATEGORIES = ["CAT1", "CAT2", "CAT3"]

args = {
    'owner': 'airflow',
```

(continues on next page)

(continued from previous page)

```
    'start_date': airflow.utils.dates.days_ago(2)
}

dag = DAG(
    dag_id='example_lineage', default_args=args,
    schedule_interval='0 0 * * *',
    dagrun_timeout=timedelta(minutes=60))

f_final = File("/tmp/final")
run_this_last = DummyOperator(task_id='run_this_last', dag=dag,
    inlets={"auto": True},
    outlets={"datasets": [f_final,]})

f_in = File("/tmp/whole_directory/")
outlets = []
for file in FILE_CATEGORIES:
    f_out = File("/tmp/{}/{{{{ execution_date }}}}".format(file))
    outlets.append(f_out)
run_this = BashOperator(
    task_id='run_me_first', bash_command='echo 1', dag=dag,
    inlets={"datasets": [f_in,]},
    outlets={"datasets": outlets}
    )
run_this.set_downstream(run_this_last)
```

Tasks take the parameters *inlets* and *outlets*. Inlets can be manually defined by a list of dataset *{"datasets": [dataset1, dataset2]}* or can be configured to look for outlets from upstream tasks *{"task_ids": ["task_id1", "task_id2"]}* or can be configured to pick up outlets from direct upstream tasks *{"auto": True}* or a combination of them. Outlets are defined as list of dataset *{"datasets": [dataset1, dataset2]}*. Any fields for the dataset are templated with the context when the task is being executed.

Note: Operators can add inlets and outlets automatically if the operator supports it.

In the example DAG task *run_me_first* is a BashOperator that takes 3 inlets: *CAT1*, *CAT2*, *CAT3*, that are generated from a list. Note that *execution_date* is a templated field and will be rendered when the task is running.

Note: Behind the scenes Airflow prepares the lineage metadata as part of the *pre_execute* method of a task. When the task has finished execution *post_execute* is called and lineage metadata is pushed into XCOM. Thus if you are creating your own operators that override this method make sure to decorate your method with *prepare_lineage* and *apply_lineage* respectively.

3.17.1 Apache Atlas

Airflow can send its lineage metadata to Apache Atlas. You need to enable the *atlas* backend and configure it properly, e.g. in your *airflow.cfg*:

```
[lineage]
backend = airflow.lineage.backend.atlas

[atlas]
username = my_username
```

(continues on next page)

(continued from previous page)

```
password = my_password
host = host
port = 21000
```

Please make sure to have the *atlasclient* package installed.

3.18 FAQ

3.18.1 Why isn't my task getting scheduled?

There are very many reasons why your task might not be getting scheduled. Here are some of the common causes:

- Does your script "compile", can the Airflow engine parse it and find your DAG object. To test this, you can run `airflow list_dags` and confirm that your DAG shows up in the list. You can also run `airflow list_tasks foo_dag_id --tree` and confirm that your task shows up in the list as expected. If you use the CeleryExecutor, you may want to confirm that this works both where the scheduler runs as well as where the worker runs.
- Does the file containing your DAG contain the string "airflow" and "DAG" somewhere in the contents? When searching the DAG directory, Airflow ignores files not containing "airflow" and "DAG" in order to prevent the DagBag parsing from importing all python files collocated with user's DAGs.
- Is your `start_date` set properly? The Airflow scheduler triggers the task soon after the `start_date + scheduler_interval` is passed.
- Is your `schedule_interval` set properly? The default `schedule_interval` is one day (`datetime.timedelta(1)`). You must specify a different `schedule_interval` directly to the DAG object you instantiate, not as a `default_param`, as task instances do not override their parent DAG's `schedule_interval`.
- Is your `start_date` beyond where you can see it in the UI? If you set your `start_date` to some time say 3 months ago, you won't be able to see it in the main view in the UI, but you should be able to see it in the `Menu -> Browse ->Task Instances`.
- Are the dependencies for the task met. The task instances directly upstream from the task need to be in a `success` state. Also, if you have set `depends_on_past=True`, the previous task instance needs to have succeeded (except if it is the first run for that task). Also, if `wait_for_downstream=True`, make sure you understand what it means. You can view how these properties are set from the `Task Instance Details` page for your task.
- Are the DagRuns you need created and active? A DagRun represents a specific execution of an entire DAG and has a state (running, success, failed, ...). The scheduler creates new DagRun as it moves forward, but never goes back in time to create new ones. The scheduler only evaluates `running` DagRuns to see what task instances it can trigger. Note that clearing tasks instances (from the UI or CLI) does set the state of a DagRun back to running. You can bulk view the list of DagRuns and alter states by clicking on the schedule tag for a DAG.
- Is the `concurrency` parameter of your DAG reached? `concurrency` defines how many `running` task instances a DAG is allowed to have, beyond which point things get queued.
- Is the `max_active_runs` parameter of your DAG reached? `max_active_runs` defines how many `running` concurrent instances of a DAG there are allowed to be.

You may also want to read the Scheduler section of the docs and make sure you fully understand how it proceeds.

3.18.2 How do I trigger tasks based on another task's failure?

Check out the `Trigger Rule` section in the Concepts section of the documentation

3.18.3 Why are connection passwords still not encrypted in the metadata db after I installed airflow[crypto]?

Check out the `Connections` section in the Configuration section of the documentation

3.18.4 What's the deal with `start_date`?

`start_date` is partly legacy from the pre-DagRun era, but it is still relevant in many ways. When creating a new DAG, you probably want to set a global `start_date` for your tasks using `default_args`. The first DagRun to be created will be based on the `min(start_date)` for all your task. From that point on, the scheduler creates new DagRuns based on your `schedule_interval` and the corresponding task instances run as your dependencies are met. When introducing new tasks to your DAG, you need to pay special attention to `start_date`, and may want to reactivate inactive DagRuns to get the new task onboarded properly.

We recommend against using dynamic values as `start_date`, especially `datetime.now()` as it can be quite confusing. The task is triggered once the period closes, and in theory an `@hourly` DAG would never get to an hour after now as `now()` moves along.

Previously we also recommended using rounded `start_date` in relation to your `schedule_interval`. This meant an `@hourly` would be at `00:00` minutes:seconds, a `@daily` job at midnight, a `@monthly` job on the first of the month. This is no longer required. Airflow will now auto align the `start_date` and the `schedule_interval`, by using the `start_date` as the moment to start looking.

You can use any sensor or a `TimeDeltaSensor` to delay the execution of tasks within the schedule interval. While `schedule_interval` does allow specifying a `datetime.timedelta` object, we recommend using the macros or cron expressions instead, as it enforces this idea of rounded schedules.

When using `depends_on_past=True` it's important to pay special attention to `start_date` as the past dependency is not enforced only on the specific schedule of the `start_date` specified for the task. It's also important to watch DagRun activity status in time when introducing new `depends_on_past=True`, unless you are planning on running a backfill for the new task(s).

Also important to note is that the tasks `start_date`, in the context of a backfill CLI command, get overridden by the backfill's command `start_date`. This allows for a backfill on tasks that have `depends_on_past=True` to actually start, if that wasn't the case, the backfill just wouldn't start.

3.18.5 How can I create DAGs dynamically?

Airflow looks in your `DAGS_FOLDER` for modules that contain `DAG` objects in their global namespace, and adds the objects it finds in the `DagBag`. Knowing this all we need is a way to dynamically assign variable in the global namespace, which is easily done in python using the `globals()` function for the standard library which behaves like a simple dictionary.

```
for i in range(10):
    dag_id = 'foo_{}'.format(i)
    globals()[dag_id] = DAG(dag_id)
    # or better, call a function that returns a DAG object!
```

3.18.6 What are all the `airflow run` commands in my process list?

There are many layers of `airflow run` commands, meaning it can call itself.

- Basic `airflow run`: fires up an executor, and tell it to run an `airflow run --local` command. if using Celery, this means it puts a command in the queue for it to run remote, on the worker. If using LocalExecutor, that translates into running it in a subprocess pool.
- Local `airflow run --local`: starts an `airflow run --raw` command (described below) as a subprocess and is in charge of emitting heartbeats, listening for external kill signals and ensures some cleanup takes place if the subprocess fails
- Raw `airflow run --raw` runs the actual operator's execute method and performs the actual work

3.18.7 How can my airflow dag run faster?

There are three variables we could control to improve airflow dag performance:

- `parallelism`: This variable controls the number of task instances that the airflow worker can run simultaneously. User could increase the parallelism variable in the `airflow.cfg`.
- `concurrency`: The Airflow scheduler will run no more than `$concurrency` task instances for your DAG at any given time. Concurrency is defined in your Airflow DAG. If you do not set the concurrency on your DAG, the scheduler will use the default value from the `dag_concurrency` entry in your `airflow.cfg`.
- `max_active_runs`: the Airflow scheduler will run no more than `max_active_runs` DagRuns of your DAG at a given time. If you do not set the `max_active_runs` in your DAG, the scheduler will use the default value from the `max_active_runs_per_dag` entry in your `airflow.cfg`.

3.18.8 How can we reduce the airflow UI page load time?

If your dag takes long time to load, you could reduce the value of `default_dag_run_display_number` configuration in `airflow.cfg` to a smaller value. This configurable controls the number of dag run to show in UI with default value 25.

3.18.9 How to fix Exception: Global variable explicit_defaults_for_timestamp needs to be on (1)?

This means `explicit_defaults_for_timestamp` is disabled in your mysql server and you need to enable it by:

1. Set `explicit_defaults_for_timestamp = 1` under the mysqld section in your my.cnf file.
2. Restart the Mysql server.

3.18.10 How to reduce airflow dag scheduling latency in production?

- `max_threads`: Scheduler will spawn multiple threads in parallel to schedule dags. This is controlled by `max_threads` with default value of 2. User should increase this value to a larger value(e.g numbers of cpus where scheduler runs - 1) in production.
- `scheduler_heartbeat_sec`: User should consider to increase `scheduler_heartbeat_sec` config to a higher value(e.g 60 secs) which controls how frequent the airflow scheduler gets the heartbeat and updates the job's entry in database.

3.19 API Reference

3.19.1 Operators

Operators allow for generation of certain types of tasks that become nodes in the DAG when instantiated. All operators derive from `BaseOperator` and inherit many attributes and methods that way. Refer to the *BaseOperator* documentation for more details.

There are 3 main types of operators:

- Operators that performs an **action**, or tell another system to perform an action
- **Transfer** operators move data from one system to another
- **Sensors** are a certain type of operator that will keep running until a certain criterion is met. Examples include a specific file landing in HDFS or S3, a partition appearing in Hive, or a specific time of the day. Sensors are derived from `BaseSensorOperator` and run a poke method at a specified `poke_interval` until it returns `True`.

3.19.1.1 BaseOperator

All operators are derived from `BaseOperator` and acquire much functionality through inheritance. Since this is the core of the engine, it's worth taking the time to understand the parameters of `BaseOperator` to understand the primitive features that can be leveraged in your DAGs.

class airflow.models.**BaseOperator**(*task_id*, *owner='Airflow'*, *email=None*, *email_on_retry=True*, *email_on_failure=True*, *retries=0*, *retry_delay=datetime.timedelta(0, 300)*, *retry_exponential_backoff=False*, *max_retry_delay=None*, *start_date=None*, *end_date=None*, *schedule_interval=None*, *depends_on_past=False*, *wait_for_downstream=False*, *dag=None*, *params=None*, *default_args=None*, *adhoc=False*, *priority_weight=1*, *weight_rule=u'downstream'*, *queue='default'*, *pool=None*, *sla=None*, *execution_timeout=None*, *on_failure_callback=None*, *on_success_callback=None*, *on_retry_callback=None*, *trigger_rule=u'all_success'*, *resources=None*, *run_as_user=None*, *task_concurrency=None*, *executor_config=None*, *inlets=None*, *outlets=None*, **args*, ***kwargs*)

Bases: `airflow.utils.log.logging_mixin.LoggingMixin`

Abstract base class for all operators. Since operators create objects that become nodes in the dag, BaseOperator contains many recursive methods for dag crawling behavior. To derive this class, you are expected to override the constructor as well as the 'execute' method.

Operators derived from this class should perform or trigger certain tasks synchronously (wait for completion). Example of operators could be an operator that runs a Pig job (PigOperator), a sensor operator that waits for a partition to land in Hive (HiveSensorOperator), or one that moves data from Hive to MySQL (Hive2MySqlOperator). Instances of these operators (tasks) target specific operations, running specific scripts, functions or data transfers.

This class is abstract and shouldn't be instantiated. Instantiating a class derived from this one results in the creation of a task object, which ultimately becomes a node in DAG objects. Task dependencies should be set by using the set_upstream and/or set_downstream methods.

Parameters

- **task_id** (*string*) – a unique, meaningful id for the task
- **owner** (*string*) – the owner of the task, using the unix username is recommended
- **retries** (*int*) – the number of retries that should be performed before failing the task
- **retry_delay** (*timedelta*) – delay between retries
- **retry_exponential_backoff** (*bool*) – allow progressive longer waits between retries by using exponential backoff algorithm on retry delay (delay will be converted into seconds)
- **max_retry_delay** (*timedelta*) – maximum delay interval between retries
- **start_date** (*datetime*) – The `start_date` for the task, determines the `execution_date` for the first task instance. The best practice is to have the start_date rounded to your DAG's `schedule_interval`. Daily jobs have their start_date some day at 00:00:00, hourly jobs have their start_date at 00:00 of a specific hour. Note that Airflow simply looks at the latest `execution_date` and adds the `schedule_interval` to determine the next `execution_date`. It is also very important to note that different tasks' dependencies need to line up in time. If task A depends on task B and their start_date are offset in a way that their execution_date don't line up, A's dependencies will never be met. If you are looking to delay a task, for example running a daily task at 2AM, look into the `TimeSensor` and `TimeDeltaSensor`. We advise against using dynamic `start_date` and recommend using fixed ones. Read the FAQ entry about start_date for more information.
- **end_date** (*datetime*) – if specified, the scheduler won't go beyond this date
- **depends_on_past** (*bool*) – when set to true, task instances will run sequentially while relying on the previous task's schedule to succeed. The task instance for the start_date is allowed to run.
- **wait_for_downstream** (*bool*) – when set to true, an instance of task X will wait for tasks immediately downstream of the previous instance of task X to finish successfully before it runs. This is useful if the different instances of a task X alter the same asset, and this asset is used by tasks downstream of task X. Note that depends_on_past is forced to True wherever wait_for_downstream is used.
- **queue** (*str*) – which queue to target when running this job. Not all executors implement queue management, the CeleryExecutor does support targeting specific queues.
- **dag** (DAG) – a reference to the dag the task is attached to (if any)
- **priority_weight** (*int*) – priority weight of this task against other task. This allows the executor to trigger higher priority tasks before others when things get backed up.
- **weight_rule** (*str*) – weighting method used for the effective total priority weight of the task. Options are: `{ downstream | upstream | absolute }` default is `downstream` When set to `downstream` the effective weight of the task is the aggregate sum of all downstream descendants. As a result, upstream tasks will have higher weight and will be scheduled more aggressively when using positive weight values. This is useful when you have multiple dag run instances and desire to have all upstream tasks to complete for all runs before each dag can continue processing downstream tasks. When set to `upstream` the effective weight is the aggregate sum of all upstream ancestors. This is the opposite where downtream tasks have higher weight and will be scheduled more aggressively when using positive weight values. This is useful when you have multiple dag run instances and prefer to have each dag complete before starting upstream tasks of other dags. When set to

absolute, the effective weight is the exact priority_weight specified without additional weighting. You may want to do this when you know exactly what priority weight each task should have. Additionally, when set to absolute, there is bonus effect of significantly speeding up the task creation process as for very large DAGS. Options can be set as string or using the constants defined in the static class airflow.utils.WeightRule

- **pool** (*str*) – the slot pool this task should run in, slot pools are a way to limit concurrency for certain tasks
- **sla** (*datetime.timedelta*) – time by which the job is expected to succeed. Note that this represents the timedelta after the period is closed. For example if you set an SLA of 1 hour, the scheduler would send an email soon after 1:00AM on the 2016-01-02 if the 2016-01-01 instance has not succeeded yet. The scheduler pays special attention for jobs with an SLA and sends alert emails for sla misses. SLA misses are also recorded in the database for future reference. All tasks that share the same SLA time get bundled in a single email, sent soon after that time. SLA notification are sent once and only once for each task instance.
- **execution_timeout** (*datetime.timedelta*) – max time allowed for the execution of this task instance, if it goes beyond it will raise and fail.
- **on_failure_callback** (*callable*) – a function to be called when a task instance of this task fails. a context dictionary is passed as a single parameter to this function. Context contains references to related objects to the task instance and is documented under the macros section of the API.
- **on_retry_callback** – much like the on_failure_callback except that it is executed when retries occur.
- **on_success_callback** (*callable*) – much like the on_failure_callback except that it is executed when the task succeeds.
- **trigger_rule** (*str*) – defines the rule by which dependencies are applied for the task to get triggered. Options are: { all_success | all_failed | all_done | one_success | one_failed | dummy} default is all_success. Options can be set as string or using the constants defined in the static class airflow.utils.TriggerRule
- **resources** (*dict*) – A map of resource parameter names (the argument names of the Resources constructor) to their values.
- **run_as_user** (*str*) – unix username to impersonate while running the task
- **task_concurrency** (*int*) – When set, a task will be able to limit the concurrent runs across execution_dates
- **executor_config** (*dict*) – Additional task-level configuration parameters that are interpreted by a specific executor. Parameters are namespaced by the name of executor. ‘‘example: to run this task in a specific docker container through the KubernetesExecutor MyOperator(. . . ,

  executor_config={ “KubernetesExecutor”:

  {“image”: “myCustomDockerImage”} }

)‘‘

clear (***kwargs*)

Clears the state of task instances associated with the task, following the parameters specified.

dag

Returns the Operator’s DAG if set, otherwise raises an error

deps
Returns the list of dependencies for the operator. These differ from execution context dependencies in that they are specific to tasks and can be extended/overridden by subclasses.

downstream_list
@property: list of tasks directly downstream

execute(*context*)
This is the main method to derive when creating an operator. Context is the same dictionary used as when rendering jinja templates.

Refer to get_template_context for more context.

get_direct_relative_ids(*upstream=False*)
Get the direct relative ids to the current task, upstream or downstream.

get_direct_relatives(*upstream=False*)
Get the direct relatives to the current task, upstream or downstream.

get_flat_relative_ids(*upstream=False*, *found_descendants=None*)
Get a flat list of relatives' ids, either upstream or downstream.

get_flat_relatives(*upstream=False*)
Get a flat list of relatives, either upstream or downstream.

get_task_instances(*session*, *start_date=None*, *end_date=None*)
Get a set of task instance related to this task for a specific date range.

has_dag()
Returns True if the Operator has been assigned to a DAG.

on_kill()
Override this method to cleanup subprocesses when a task instance gets killed. Any use of the threading, subprocess or multiprocessing module within an operator needs to be cleaned up or it will leave ghost processes behind.

post_execute(*context*, **args*, ***kwargs*)
This hook is triggered right after self.execute() is called. It is passed the execution context and any results returned by the operator.

pre_execute(*context*, **args*, ***kwargs*)
This hook is triggered right before self.execute() is called.

prepare_template()
Hook that is triggered after the templated fields get replaced by their content. If you need your operator to alter the content of the file before the template is rendered, it should override this method to do so.

render_template(*attr*, *content*, *context*)
Renders a template either from a file or directly in a field, and returns the rendered result.

render_template_from_field(*attr*, *content*, *context*, *jinja_env*)
Renders a template from a field. If the field is a string, it will simply render the string and return the result. If it is a collection or nested set of collections, it will traverse the structure and render all strings in it.

run(*start_date=None*, *end_date=None*, *ignore_first_depends_on_past=False*, *ignore_ti_state=False*, *mark_success=False*)
Run a set of task instances for a date range.

schedule_interval
The schedule interval of the DAG always wins over individual tasks so that tasks within a DAG always line up. The task still needs a schedule_interval as it may not be attached to a DAG.

set_downstream(*task_or_task_list*)
Set a task or a task list to be directly downstream from the current task.

set_upstream(*task_or_task_list*)
Set a task or a task list to be directly upstream from the current task.

upstream_list
@property: list of tasks directly upstream

xcom_pull(*context*, *task_ids=None*, *dag_id=None*, *key=u'return_value'*, *include_prior_dates=None*)
See TaskInstance.xcom_pull()

xcom_push(*context*, *key*, *value*, *execution_date=None*)
See TaskInstance.xcom_push()

3.19.1.2 BaseSensorOperator

All sensors are derived from `BaseSensorOperator`. All sensors inherit the `timeout` and `poke_interval` on top of the `BaseOperator` attributes.

class airflow.sensors.base_sensor_operator.**BaseSensorOperator**(*poke_interval=60*, *timeout=604800*, *soft_fail=False*, **args*, ***kwargs*)
Bases: *airflow.models.BaseOperator*, airflow.models.SkipMixin

Sensor operators are derived from this class an inherit these attributes.

Sensor operators keep executing at a time interval and succeed when a criteria is met and fail if and when they time out.

Parameters

- **soft_fail** (*bool*) – Set to true to mark the task as SKIPPED on failure
- **poke_interval** (*int*) – Time in seconds that the job should wait in between each tries
- **timeout** (*int*) – Time, in seconds before the task times out and fails.

poke(*context*)
Function that the sensors defined while deriving this class should override.

3.19.1.3 Core Operators

Operators

class airflow.operators.bash_operator.**BashOperator**(*bash_command*, *xcom_push=False*, *env=None*, *output_encoding='utf-8'*, **args*, ***kwargs*)
Bases: *airflow.models.BaseOperator*

Execute a Bash script, command or set of commands.

Parameters

- **bash_command** (*string*) – The command, set of commands or reference to a bash script (must be '.sh') to be executed. (templated)

- **xcom_push** (*bool*) – If xcom_push is True, the last line written to stdout will also be pushed to an XCom when the bash command completes.
- **env** (*dict*) – If env is not None, it must be a mapping that defines the environment variables for the new process; these are used instead of inheriting the current process environment, which is the default behavior. (templated)

execute(*context*)
Execute the bash command in a temporary directory which will be cleaned afterwards

class airflow.operators.python_operator.**BranchPythonOperator**(*python_callable*, *op_args=None*, *op_kwargs=None*, *provide_context=False*, *templates_dict=None*, *templates_exts=None*, **args*, ***kwargs*)

Bases: *airflow.operators.python_operator.PythonOperator*, airflow.models.SkipMixin

Allows a workflow to "branch" or follow a single path following the execution of this task.

It derives the PythonOperator and expects a Python function that returns the task_id to follow. The task_id returned should point to a task directly downstream from {self}. All other "branches" or directly downstream tasks are marked with a state of `skipped` so that these paths can't move forward. The `skipped` states are propageted downstream to allow for the DAG state to fill up and the DAG run's state to be inferred.

Note that using tasks with `depends_on_past=True` downstream from `BranchPythonOperator` is logically unsound as `skipped` status will invariably lead to block tasks that depend on their past successes. `skipped` states propagates where all directly upstream tasks are `skipped`.

class airflow.operators.check_operator.**CheckOperator**(*sql*, *conn_id=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Performs checks against a db. The `CheckOperator` expects a sql query that will return a single row. Each value on that first row is evaluated using python `bool` casting. If any of the values return `False` the check is failed and errors out.

Note that Python bool casting evals the following as `False`:

- `False`
- `0`
- Empty string (`""`)
- Empty list (`[]`)
- Empty dictionary or set (`{}`)

Given a query like `SELECT COUNT(*) FROM foo`, it will fail only if the count `== 0`. You can craft much more complex query that could, for instance, check that the table has the same number of rows as the source table upstream, or that the count of today's partition is greater than yesterday's partition, or that a set of metrics are less than 3 standard deviation for the 7 day average.

This operator can be used as a data quality check in your pipeline, and depending on where you put it in your DAG, you have the choice to stop the critical path, preventing from publishing dubious data, or on the side and receive email alerts without stopping the progress of the DAG.

Note that this is an abstract class and get_db_hook needs to be defined. Whereas a get_db_hook is hook that gets a single record from an external source.

Parameters **sql** (*string*) – the sql to be executed. (templated)

class airflow.operators.docker_operator.**DockerOperator**(*image*, *api_version=None*, *command=None*, *cpus=1.0*, *docker_url='unix://var/run/docker.sock'*, *environment=None*, *force_pull=False*, *mem_limit=None*, *network_mode=None*, *tls_ca_cert=None*, *tls_client_cert=None*, *tls_client_key=None*, *tls_hostname=None*, *tls_ssl_version=None*, *tmp_dir='/tmp/airflow'*, *user=None*, *volumes=None*, *working_dir=None*, *xcom_push=False*, *xcom_all=False*, *docker_conn_id=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Execute a command inside a docker container.

A temporary directory is created on the host and mounted into a container to allow storing files that together exceed the default disk size of 10GB in a container. The path to the mounted directory can be accessed via the environment variable `AIRFLOW_TMP_DIR`.

If a login to a private registry is required prior to pulling the image, a Docker connection needs to be configured in Airflow and the connection ID be provided with the parameter `docker_conn_id`.

Parameters

- **image** (*str*) – Docker image from which to create the container.
- **api_version** (*str*) – Remote API version. Set to `auto` to automatically detect the server's version.
- **command** (*str or list*) – Command to be run in the container. (templated)
- **cpus** (*float*) – Number of CPUs to assign to the container. This value gets multiplied with 1024. See https://docs.docker.com/engine/reference/run/#cpu-share-constraint
- **docker_url** (*str*) – URL of the host running the docker daemon. Default is unix://var/run/docker.sock
- **environment** (*dict*) – Environment variables to set in the container. (templated)
- **force_pull** (*bool*) – Pull the docker image on every run. Default is false.
- **mem_limit** (*float or str*) – Maximum amount of memory the container can use. Either a float value, which represents the limit in bytes, or a string like `128m` or `1g`.
- **network_mode** (*str*) – Network mode for the container.
- **tls_ca_cert** (*str*) – Path to a PEM-encoded certificate authority to secure the docker connection.

- **tls_client_cert** (*str*) – Path to the PEM-encoded certificate used to authenticate docker client.
- **tls_client_key** (*str*) – Path to the PEM-encoded key used to authenticate docker client.
- **tls_hostname** (*str or bool*) – Hostname to match against the docker server certificate or False to disable the check.
- **tls_ssl_version** (*str*) – Version of SSL to use when communicating with docker daemon.
- **tmp_dir** (*str*) – Mount point inside the container to a temporary directory created on the host by the operator. The path is also made available via the environment variable `AIRFLOW_TMP_DIR` inside the container.
- **user** (*int or str*) – Default user inside the docker container.
- **volumes** – List of volumes to mount into the container, e.g. `['/host/path:/container/path', '/host/path2:/container/path2:ro']`.
- **working_dir** (*str*) – Working directory to set on the container (equivalent to the -w switch the docker client)
- **xcom_push** (*bool*) – Does the stdout will be pushed to the next step using XCom. The default is False.
- **xcom_all** (*bool*) – Push all the stdout or just the last line. The default is False (last line).
- **docker_conn_id** (*str*) – ID of the Airflow connection to use

class airflow.operators.dummy_operator.**DummyOperator**(**args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Operator that does literally nothing. It can be used to group tasks in a DAG.

class airflow.operators.email_operator.**EmailOperator**(*to*, *subject*, *html_content*, *files=None*, *cc=None*, *bcc=None*, *mime_subtype='mixed'*, *mime_charset='us_ascii'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Sends an email.

Parameters

- **to** (*list or string (comma or semicolon delimited)*) – list of emails to send the email to. (templated)
- **subject** (*string*) – subject line for the email. (templated)
- **html_content** (*string*) – content of the email, html markup is allowed. (templated)
- **files** (*list*) – file names to attach in email
- **cc** (*list or string (comma or semicolon delimited)*) – list of recipients to be added in CC field
- **bcc** (*list or string (comma or semicolon delimited)*) – list of recipients to be added in BCC field
- **mime_subtype** (*string*) – MIME sub content type
- **mime_charset** (*string*) – character set parameter added to the Content-Type header.

class airflow.operators.generic_transfer.**GenericTransfer**(*sql*, *destination_table*, *source_conn_id*, *destination_conn_id*, *preoperator=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Moves data from a connection to another, assuming that they both provide the required methods in their respective hooks. The source hook needs to expose a *get_records* method, and the destination a *insert_rows* method.

This is meant to be used on small-ish datasets that fit in memory.

Parameters

- **sql** (*str*) – SQL query to execute against the source database. (templated)
- **destination_table** (*str*) – target table. (templated)
- **source_conn_id** (*str*) – source connection
- **destination_conn_id** (*str*) – source connection
- **preoperator** (*str or list of str*) – sql statement or list of statements to be executed prior to loading the data. (templated)

class airflow.operators.check_operator.**IntervalCheckOperator**(*table*, *metrics_thresholds*, *date_filter_column='ds'*, *days_back=-7*, *conn_id=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Checks that the values of metrics given as SQL expressions are within a certain tolerance of the ones from days_back before.

Note that this is an abstract class and get_db_hook needs to be defined. Whereas a get_db_hook is hook that gets a single record from an external source.

Parameters

- **table** (*str*) – the table name
- **days_back** (*int*) – number of days between ds and the ds we want to check against. Defaults to 7 days
- **metrics_threshold** (*dict*) – a dictionary of ratios indexed by metrics

class airflow.operators.latest_only_operator.**LatestOnlyOperator**(*task_id*, *owner='Airflow'*, *email=None*, *email_on_retry=True*, *email_on_failure=True*, *retries=0*, *retry_delay=datetime.timedelta(0, 300)*, *retry_exponential_backoff=False*, *max_retry_delay=None*, *start_date=None*, *end_date=None*, *schedule_interval=None*, *depends_on_past=False*, *wait_for_downstream=False*, *dag=None*, *params=None*, *default_args=None*, *adhoc=False*, *priority_weight=1*, *weight_rule=u'downstream'*, *queue='default'*, *pool=None*, *sla=None*, *execution_timeout=None*, *on_failure_callback=None*, *on_success_callback=None*, *on_retry_callback=None*, *trigger_rule=u'all_success'*, *resources=None*, *run_as_user=None*, *task_concurrency=None*, *executor_config=None*, *inlets=None*, *outlets=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*, airflow.models.SkipMixin

Allows a workflow to skip tasks that are not running during the most recent schedule interval.

If the task is run outside of the latest schedule interval, all directly downstream tasks will be skipped.

class airflow.operators.mssql_operator.**MsSqlOperator**(*sql*, *mssql_conn_id='mssql_default'*, *parameters=None*, *autocommit=False*, *database=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes sql code in a specific Microsoft SQL database

Parameters

- **mssql_conn_id** (*string*) – reference to a specific mssql database
- **sql** (*string or string pointing to a template file with .sql extension. (templated)*) – the sql code to be executed
- **database** (*string*) – name of database which overwrite defined one in connection

class airflow.operators.pig_operator.**PigOperator**(*pig*, *pig_cli_conn_id='pig_cli_default'*, *pigparams_jinja_translate=False*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes pig script.

Parameters

- **pig** (*string*) – the pig latin script to be executed. (templated)
- **pig_cli_conn_id** (*string*) – reference to the Hive database
- **pigparams_jinja_translate** (*boolean*) – when True, pig params-type templating ${var} gets translated into jinja-type templating {{ var }}. Note that you may want to use this along with the `DAG(user_defined_macros=myargs)` parameter. View the DAG object documentation for more details.

class airflow.operators.python_operator.**PythonOperator**(*python_callable*, *op_args=None*, *op_kwargs=None*, *provide_context=False*, *templates_dict=None*, *templates_exts=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes a Python callable

Parameters

- **python_callable** (*python callable*) – A reference to an object that is callable
- **op_kwargs** (*dict*) – a dictionary of keyword arguments that will get unpacked in your function
- **op_args** (*list*) – a list of positional arguments that will get unpacked when calling your callable
- **provide_context** (*bool*) – if set to true, Airflow will pass a set of keyword arguments that can be used in your function. This set of kwargs correspond exactly to what you can use in your jinja templates. For this to work, you need to define ***kwargs* in your function header.
- **templates_dict** (*dict of str*) – a dictionary where the values are templates that will get templated by the Airflow engine sometime between `__init__` and `execute` takes place and are made available in your callable's context after the template has been applied. (templated)
- **templates_exts** (*list(str)*) – a list of file extensions to resolve while processing templated fields, for examples `['.sql', '.hql']`

class airflow.operators.python_operator.**PythonVirtualenvOperator**(*python_callable*, *requirements=None*, *python_version=None*, *use_dill=False*, *system_site_packages=True*, *op_args=None*, *op_kwargs=None*, *string_args=None*, *templates_dict=None*, *templates_exts=None*, **args*, ***kwargs*)

Bases: *airflow.operators.python_operator.PythonOperator*

Allows one to run a function in a virtualenv that is created and destroyed automatically (with certain caveats).

The function must be defined using def, and not be part of a class. All imports must happen inside the function and no variables outside of the scope may be referenced. A global scope variable named virtualenv_string_args will be available (populated by string_args). In addition, one can pass stuff through op_args and op_kwargs, and one can use a return value.

Note that if your virtualenv runs in a different Python major version than Airflow, you cannot use return values, op_args, or op_kwargs. You can use string_args though.

Parameters

- **python_callable** (*function*) – A python function with no references to outside variables, defined with def, which will be run in a virtualenv
- **requirements** (*list(str)*) – A list of requirements as specified in a pip install command
- **python_version** (*str*) – The Python version to run the virtualenv with. Note that both 2 and 2.7 are acceptable forms.
- **use_dill** (*bool*) – Whether to use dill to serialize the args and result (pickle is default). This allow more complex types but requires you to include dill in your requirements.
- **system_site_packages** (*bool*) – Whether to include system_site_packages in your virtualenv. See virtualenv documentation for more information.
- **op_args** – A list of positional arguments to pass to python_callable.
- **op_kwargs** (*dict*) – A dict of keyword arguments to pass to python_callable.
- **string_args** (*list(str)*) – Strings that are present in the global var virtualenv_string_args, available to python_callable at runtime as a list(str). Note that args are split by newline.
- **templates_dict** (*dict of str*) – a dictionary where the values are templates that will get templated by the Airflow engine sometime between `__init__` and `execute` takes place and are made available in your callable's context after the template has been applied
- **templates_exts** (*list(str)*) – a list of file extensions to resolve while processing templated fields, for examples `['.sql', '.hql']`

class airflow.operators.s3_file_transform_operator.**S3FileTransformOperator**(*source_s3_key*, *dest_s3_key*, *transform_script=None*, *select_expression=None*, *source_aws_conn_id=*, *dest_aws_conn_id='aw*, *replace=False*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Copies data from a source S3 location to a temporary location on the local filesystem. Runs a transformation on this file as specified by the transformation script and uploads the output to a destination S3 location.

The locations of the source and the destination files in the local filesystem is provided as an first and second arguments to the transformation script. The transformation script is expected to read the data from source, transform it and write the output to the local destination file. The operator then takes over control and uploads the local destination file to S3.

S3 Select is also available to filter the source contents. Users can omit the transformation script if S3 Select expression is specified.

Parameters

- **source_s3_key** (*str*) – The key to be retrieved from S3. (templated)
- **source_aws_conn_id** (*str*) – source s3 connection
- **dest_s3_key** (*str*) – The key to be written from S3. (templated)
- **dest_aws_conn_id** (*str*) – destination s3 connection
- **replace** (*bool*) – Replace dest S3 key if it already exists
- **transform_script** (*str*) – location of the executable transformation script
- **select_expression** (*str*) – S3 Select expression

class airflow.operators.python_operator.**ShortCircuitOperator**(*python_callable*, *op_args=None*, *op_kwargs=None*, *provide_context=False*, *templates_dict=None*, *templates_exts=None*, **args*, ***kwargs*)

Bases: *airflow.operators.python_operator.PythonOperator*, airflow.models.SkipMixin

Allows a workflow to continue only if a condition is met. Otherwise, the workflow "short-circuits" and downstream tasks are skipped.

The ShortCircuitOperator is derived from the PythonOperator. It evaluates a condition and short-circuits the workflow if the condition is False. Any downstream tasks are marked with a state of "skipped". If the condition is True, downstream tasks proceed as normal.

The condition is determined by the result of *python_callable*.

class airflow.operators.http_operator.**SimpleHttpOperator**(*endpoint*, *method='POST'*, *data=None*, *headers=None*, *response_check=None*, *extra_options=None*, *xcom_push=False*, *http_conn_id='http_default'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Calls an endpoint on an HTTP system to execute an action

Parameters

- **http_conn_id** (*string*) – The connection to run the sensor against
- **endpoint** (*string*) – The relative part of the full url. (templated)
- **method** (*string*) – The HTTP method to use, default = "POST"
- **data** (*For POST/PUT, depends on the content-type parameter, for GET a dictionary of key/value string pairs*) – The data to pass. POST-data in POST/PUT and params in the URL for a GET request. (templated)
- **headers** (*a dictionary of string key/value pairs*) – The HTTP headers to be added to the GET request
- **response_check** (*A lambda or defined function.*) – A check against the 'requests' response object. Returns True for 'pass' and False otherwise.
- **extra_options** (*A dictionary of options, where key is string and value depends on the option that's being modified.*) – Extra options for the 'requests' library, see the 'requests' documentation (options to modify timeout, ssl, etc.)

class airflow.operators.sqlite_operator.**SqliteOperator**(*sql*, *sqlite_conn_id='sqlite_default'*, *parameters=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes sql code in a specific Sqlite database

Parameters

- **sqlite_conn_id** (*string*) – reference to a specific sqlite database
- **sql** (*string or string pointing to a template file. File must have a '.sql' extensions.*) – the sql code to be executed. (templated)

class airflow.operators.subdag_operator.**SubDagOperator**(***kwargs*)

Bases: *airflow.models.BaseOperator*

class airflow.operators.dagrun_operator.**TriggerDagRunOperator**(*trigger_dag_id*, *python_callable=None*, *execution_date=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Triggers a DAG run for a specified `dag_id`

Parameters

- **trigger_dag_id** (*str*) – the dag_id to trigger
- **python_callable** (*python callable*) – a reference to a python function that will be called while passing it the `context` object and a placeholder object `obj` for your callable to fill and return if you want a DagRun created. This `obj` object contains a `run_id` and `payload` attribute that you can modify in your function. The `run_id` should be a unique identifier for that DAG run, and the payload has to be a picklable object that will be made available to your tasks while executing that DAG run. Your function header should look like `def foo(context, dag_run_obj):`
- **execution_date** (*datetime.datetime*) – Execution date for the dag

class airflow.operators.check_operator.**ValueCheckOperator**(*sql*, *pass_value*, *tolerance=None*, *conn_id=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Performs a simple value check using sql code.

Note that this is an abstract class and get_db_hook needs to be defined. Whereas a get_db_hook is hook that gets a single record from an external source.

Parameters **sql** (*string*) – the sql to be executed. (templated)

Sensors

class airflow.sensors.external_task_sensor.**ExternalTaskSensor**(*external_dag_id*, *external_task_id*, *allowed_states=None*, *execution_delta=None*, *execution_date_fn=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a task to complete in a different DAG

Parameters

- **external_dag_id** (*string*) – The dag_id that contains the task you want to wait for
- **external_task_id** (*string*) – The task_id that contains the task you want to wait for
- **allowed_states** (*list*) – list of allowed states, default is `['success']`
- **execution_delta** (*datetime.timedelta*) – time difference with the previous execution to look at, the default is the same execution_date as the current task. For yesterday, use [positive!] datetime.timedelta(days=1). Either execution_delta or execution_date_fn can be passed to ExternalTaskSensor, but not both.
- **execution_date_fn** (*callable*) – function that receives the current execution date and returns the desired execution dates to query. Either execution_delta or execution_date_fn can be passed to ExternalTaskSensor, but not both.

poke(***kwargs*)

Function that the sensors defined while deriving this class should override.

class airflow.sensors.hive_partition_sensor.**HivePartitionSensor**(*table*, *partition="ds='{{ ds }}'"*, *metastore_conn_id='metastore_default'*, *schema='default'*, *poke_interval=180*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a partition to show up in Hive.

Note: Because `partition` supports general logical operators, it can be inefficient. Consider using NamedHivePartitionSensor instead if you don't need the full flexibility of HivePartitionSensor.

Parameters

- **table** (*string*) – The name of the table to wait for, supports the dot notation (my_database.my_table)
- **partition** (*string*) – The partition clause to wait for. This is passed as is to the metastore Thrift client `get_partitions_by_filter` method, and apparently supports SQL like notation as in `ds='2015-01-01' AND type='value'` and comparison operators as in `"ds>=2015-01-01"`
- **metastore_conn_id** (*str*) – reference to the metastore thrift service connection id

poke(*context*)

Function that the sensors defined while deriving this class should override.

class airflow.sensors.http_sensor.**HttpSensor**(*endpoint*, *http_conn_id='http_default'*, *method='GET'*, *request_params=None*, *headers=None*, *response_check=None*, *extra_options=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Executes a HTTP get statement and returns False on failure: 404 not found or response_check function returned False

Parameters

- **http_conn_id** (*string*) – The connection to run the sensor against
- **method** (*string*) – The HTTP request method to use
- **endpoint** (*string*) – The relative part of the full url
- **request_params** (*a dictionary of string key/value pairs*) – The parameters to be added to the GET url
- **headers** (*a dictionary of string key/value pairs*) – The HTTP headers to be added to the GET request
- **response_check** (*A lambda or defined function.*) – A check against the 'requests' response object. Returns True for 'pass' and False otherwise.
- **extra_options** (*A dictionary of options, where key is string and value depends on the option that's being modified.*) – Extra options for the 'requests' library, see the 'requests' documentation (options to modify timeout, ssl, etc.)

poke(*context*)
Function that the sensors defined while deriving this class should override.

class airflow.sensors.metastore_partition_sensor.**MetastorePartitionSensor**(*table*, *partition_name*, *schema='default'*, *mysql_conn_id='metasto*, **args*, ***kwargs*)

Bases: *airflow.sensors.sql_sensor.SqlSensor*

An alternative to the HivePartitionSensor that talk directly to the MySQL db. This was created as a result of observing sub optimal queries generated by the Metastore thrift service when hitting subpartitioned tables. The Thrift service's queries were written in a way that wouldn't leverage the indexes.

Parameters

- **schema** (*str*) – the schema
- **table** (*str*) – the table
- **partition_name** (*str*) – the partition name, as defined in the PARTITIONS table of the Metastore. Order of the fields does matter. Examples: `ds=2016-01-01` or `ds=2016-01-01/sub=foo` for a sub partitioned table
- **mysql_conn_id** (*str*) – a reference to the MySQL conn_id for the metastore

poke(*context*)
Function that the sensors defined while deriving this class should override.

class airflow.sensors.named_hive_partition_sensor.**NamedHivePartitionSensor**(*partition_names*, *metastore_conn_id='metasto*, *poke_interval=180*, *hook=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a set of partitions to show up in Hive.

Parameters

- **partition_names** (*list of strings*) – List of fully qualified names of the partitions to wait for. A fully qualified name is of the form `schema.table/pk1=pv1/pk2=pv2`, for example, default.users/ds=2016-01-01. This is passed as is to the metastore Thrift client `get_partitions_by_name` method. Note that you cannot use logical or comparison operators as in HivePartitionSensor.
- **metastore_conn_id** (*str*) – reference to the metastore thrift service connection id

poke(*context*)
Function that the sensors defined while deriving this class should override.

class airflow.sensors.s3_key_sensor.**S3KeySensor**(*bucket_key*, *bucket_name=None*, *wildcard_match=False*, *aws_conn_id='aws_default'*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a key (a file-like instance on S3) to be present in a S3 bucket. S3 being a key/value it does not support folders. The path is just a key a resource.

Parameters

- **bucket_key** (*str*) – The key being waited on. Supports full s3:// style url or relative path from root level.
- **bucket_name** (*str*) – Name of the S3 bucket
- **wildcard_match** (*bool*) – whether the bucket_key should be interpreted as a Unix wildcard pattern
- **aws_conn_id** (*str*) – a reference to the s3 connection

poke(*context*)

Function that the sensors defined while deriving this class should override.

class airflow.sensors.s3_prefix_sensor.**S3PrefixSensor**(*bucket_name*, *prefix*, *delimiter='/'*, *aws_conn_id='aws_default'*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a prefix to exist. A prefix is the first part of a key, thus enabling checking of constructs similar to glob airfl* or SQL LIKE 'airfl%'. There is the possibility to precise a delimiter to indicate the hierarchy or keys, meaning that the match will stop at that delimiter. Current code accepts sane delimiters, i.e. characters that are NOT special characters in the Python regex engine.

Parameters

- **bucket_name** (*str*) – Name of the S3 bucket
- **prefix** (*str*) – The prefix being waited on. Relative path from bucket root level.
- **delimiter** (*str*) – The delimiter intended to show hierarchy. Defaults to '/'.

poke(*context*)

Function that the sensors defined while deriving this class should override.

class airflow.sensors.sql_sensor.**SqlSensor**(*conn_id*, *sql*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Runs a sql statement until a criteria is met. It will keep trying while sql returns no row, or if the first cell in (0, '0', '').

Parameters

- **conn_id** (*string*) – The connection to run the sensor against
- **sql** – The sql to run. To pass, it needs to return at least one cell that contains a non-zero / empty string value.

poke(*context*)

Function that the sensors defined while deriving this class should override.

class airflow.sensors.time_sensor.**TimeSensor**(*target_time*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits until the specified time of the day.

Parameters **target_time** (*datetime.time*) – time after which the job succeeds

poke(*context*)

Function that the sensors defined while deriving this class should override.

class airflow.sensors.time_delta_sensor.**TimeDeltaSensor**(*delta*, **args*, ***kwargs*)
Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a timedelta after the task's execution_date + schedule_interval. In Airflow, the daily task stamped with `execution_date` 2016-01-01 can only start running on 2016-01-02. The timedelta here represents the time after the execution period has closed.

Parameters **delta** (*datetime.timedelta*) – time length to wait after execution_date before succeeding

poke(*context*)
Function that the sensors defined while deriving this class should override.

class airflow.sensors.web_hdfs_sensor.**WebHdfsSensor**(*filepath*, *webhdfs_conn_id='webhdfs_default'*, **args*, ***kwargs*)
Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a file or folder to land in HDFS

poke(*context*)
Function that the sensors defined while deriving this class should override.

3.19.1.4 Community-contributed Operators

Operators

class airflow.contrib.operators.awsbatch_operator.**AWSBatchOperator**(*job_name*, *job_definition*, *job_queue*, *overrides*, *max_retries=4200*, *aws_conn_id=None*, *region_name=None*, ***kwargs*)
Bases: *airflow.models.BaseOperator*

Execute a job on AWS Batch Service

Parameters

- **job_name** (*str*) – the name for the job that will run on AWS Batch
- **job_definition** (*str*) – the job definition name on AWS Batch
- **job_queue** (*str*) – the queue name on AWS Batch
- **max_retries** (*int*) – exponential backoff retries while waiter is not merged, 4200 = 48 hours
- **aws_conn_id** (*str*) – connection id of AWS credentials / region name. If None, credential boto3 strategy will be used (http://boto3.readthedocs.io/en/latest/guide/configuration.html).
- **region_name** – region name to use in AWS Hook. Override the region_name in connection (if provided)

Param overrides: the same parameter that boto3 will receive on containerOverrides (templated): http://boto3.readthedocs.io/en/latest/reference/services/batch.html#submit_job

Type overrides: dict

class airflow.contrib.operators.bigquery_check_operator.**BigQueryCheckOperator**(*sql*, *bigquery_conn_id='b*, **args*, ***kwargs*)

Bases: *airflow.operators.check_operator.CheckOperator*

Performs checks against BigQuery. The `BigQueryCheckOperator` expects a sql query that will return a single row. Each value on that first row is evaluated using python `bool` casting. If any of the values return `False` the check is failed and errors out.

Note that Python bool casting evals the following as `False`:

- `False`
- `0`
- Empty string (`""`)
- Empty list (`[]`)
- Empty dictionary or set (`{}`)

Given a query like `SELECT COUNT(*) FROM foo`, it will fail only if the count `== 0`. You can craft much more complex query that could, for instance, check that the table has the same number of rows as the source table upstream, or that the count of today's partition is greater than yesterday's partition, or that a set of metrics are less than 3 standard deviation for the 7 day average.

This operator can be used as a data quality check in your pipeline, and depending on where you put it in your DAG, you have the choice to stop the critical path, preventing from publishing dubious data, or on the side and receive email alterts without stopping the progress of the DAG.

Parameters

- **sql** (*string*) – the sql to be executed
- **bigquery_conn_id** (*string*) – reference to the BigQuery database

class airflow.contrib.operators.bigquery_check_operator.**BigQueryValueCheckOperator**(*sql*, *pass_value*, *tolerance=None*, *bigquery_conn*, **args*, ***kwargs*)

Bases: *airflow.operators.check_operator.ValueCheckOperator*

Performs a simple value check using sql code.

Parameters **sql** (*string*) – the sql to be executed

class airflow.contrib.operators.bigquery_check_operator.**BigQueryIntervalCheckOperator**(*table*, *metrics_th*, *date_fi*, *days_b*, *7*, *bigquery_*, **args*, ***kwa*)

Bases: *airflow.operators.check_operator.IntervalCheckOperator*

Checks that the values of metrics given as SQL expressions are within a certain tolerance of the ones from days_back before.

This method constructs a query like so

```
SELECT {metrics_threshold_dict_key} FROM {table}
    WHERE {date_filter_column}=<date>
```

Parameters

- **table** (*str*) – the table name
- **days_back** (*int*) – number of days between ds and the ds we want to check against. Defaults to 7 days
- **metrics_threshold** (*dict*) – a dictionary of ratios indexed by metrics, for example 'COUNT(*)': 1.5 would require a 50 percent or less difference between the current day, and the prior days_back.

class airflow.contrib.operators.bigquery_get_data.**BigQueryGetDataOperator**(*dataset_id*, *table_id*, *max_results='100'*, *selected_fields=None*, *bigquery_conn_id='bigque*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Fetches the data from a BigQuery table (alternatively fetch data for selected columns) and returns data in a python list. The number of elements in the returned list will be equal to the number of rows fetched. Each element in the list will again be a list where element would represent the columns values for that row.

Example Result: `[['Tony', '10'], ['Mike', '20'], ['Steve', '15']]`

Note: If you pass fields to `selected_fields` which are in different order than the order of columns already in BQ table, the data will still be in the order of BQ table. For example if the BQ table has 3 columns as `[A,B,C]` and you pass 'B,A' in the `selected_fields` the data would still be of the form `'A,B'`.

Example:

```
get_data = BigQueryGetDataOperator(
    task_id='get_data_from_bq',
    dataset_id='test_dataset',
    table_id='Transaction_partitions',
    max_results='100',
    selected_fields='DATE',
    bigquery_conn_id='airflow-service-account'
)
```

Parameters

- **dataset_id** – The dataset ID of the requested table. (templated)
- **table_id** (*string*) – The table ID of the requested table. (templated)
- **max_results** (*string*) – The maximum number of records (rows) to be fetched from the table. (templated)
- **selected_fields** (*string*) – List of fields to return (comma-separated). If unspecified, all fields are returned.
- **bigquery_conn_id** (*string*) – reference to a specific BigQuery hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

class airflow.contrib.operators.bigquery_operator.**BigQueryCreateEmptyTableOperator**(*dataset_id*, *table_id*, *project_id=*, *schema_fie*, *gcs_schem*, *time_partit*, *bigquery_con*, *google_clo*, *delegate_to=N*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Creates a new, empty table in the specified BigQuery dataset, optionally with schema.

The schema to be used for the BigQuery table may be specified in one of two ways. You may either directly pass the schema fields in, or you may point the operator to a Google cloud storage object name. The object in Google cloud storage must be a JSON file with the schema fields in it. You can also create a table without schema.

Parameters

- **project_id** (*string*) – The project to create the table into. (templated)
- **dataset_id** (*string*) – The dataset to create the table into. (templated)
- **table_id** (*string*) – The Name of the table to be created. (templated)
- **schema_fields** (*list*) – If set, the schema field list as defined here: https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs#configuration.load.schema

  Example:

```
schema_fields=[{"name": "emp_name", "type": "STRING", "mode":
↪"REQUIRED"},
               {"name": "salary", "type": "INTEGER", "mode":
↪"NULLABLE"}]
```

- **gcs_schema_object** (*string*) – Full path to the JSON file containing schema (templated). For example: `gs://test-bucket/dir1/dir2/employee_schema.json`
- **time_partitioning** (*dict*) – configure optional time partitioning fields i.e. partition by field, type and expiration as per API specifications.

  See also:

  https://cloud.google.com/bigquery/docs/reference/rest/v2/tables#timePartitioning
- **bigquery_conn_id** (*string*) – Reference to a specific BigQuery hook.
- **google_cloud_storage_conn_id** (*string*) – Reference to a specific Google cloud storage hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

Example (with schema JSON in GCS):

```
CreateTable = BigQueryCreateEmptyTableOperator(
    task_id='BigQueryCreateEmptyTableOperator_task',
    dataset_id='ODS',
    table_id='Employees',
    project_id='internal-gcp-project',
    gcs_schema_object='gs://schema-bucket/employee_schema.json',
    bigquery_conn_id='airflow-service-account',
    google_cloud_storage_conn_id='airflow-service-account'
)
```

Corresponding Schema file (`employee_schema.json`):

```
[
  {
    "mode": "NULLABLE",
    "name": "emp_name",
    "type": "STRING"
  },
  {
    "mode": "REQUIRED",
    "name": "salary",
    "type": "INTEGER"
  }
]
```

Example (with schema in the DAG):

```
CreateTable = BigQueryCreateEmptyTableOperator(
    task_id='BigQueryCreateEmptyTableOperator_task',
    dataset_id='ODS',
    table_id='Employees',
    project_id='internal-gcp-project',
    schema_fields=[{"name": "emp_name", "type": "STRING", "mode": "REQUIRED"},
```

(continues on next page)

(continued from previous page)

```
                    {"name": "salary", "type": "INTEGER", "mode": "NULLABLE"}],
    bigquery_conn_id='airflow-service-account',
    google_cloud_storage_conn_id='airflow-service-account'
)
```

class airflow.contrib.operators.bigquery_operator.**BigQueryCreateExternalTableOperator**(*bucket source des- ti- na- tion_p schem schem source com- pres- sion=' skip_le field_a ', max_b quote_ al- low_qu al- low_ja big- query_ google del- e- gate_t src_fm *args, **kwa

Bases: *airflow.models.BaseOperator*

Creates a new external table in the dataset with the data in Google Cloud Storage.

The schema to be used for the BigQuery table may be specified in one of two ways. You may either directly pass the schema fields in, or you may point the operator to a Google cloud storage object name. The object in Google cloud storage must be a JSON file with the schema fields in it.

Parameters

- **bucket** (*string*) – The bucket to point the external table to. (templated)
- **source_objects** – List of Google cloud storage URIs to point table to. (templated) If source_format is 'DATASTORE_BACKUP', the list must only contain a single URI.
- **destination_project_dataset_table** (*string*) – The dotted (<project>.)<dataset>.<table> BigQuery table to load data into (templated). If <project> is not included, project will be the project defined in the connection json.
- **schema_fields** (*list*) – If set, the schema field list as defined here: https://cloud.google.com/bigquery/docs/reference/rest/v2/jobs#configuration.load.schema

Example:

```
schema_fields=[{"name": "emp_name", "type": "STRING", "mode":
↪"REQUIRED"},
               {"name": "salary", "type": "INTEGER", "mode":
↪"NULLABLE"}]
```

Should not be set when source_format is 'DATASTORE_BACKUP'.

- **schema_object** – If set, a GCS object path pointing to a .json file that contains the schema for the table. (templated)
- **schema_object** – string
- **source_format** (*string*) – File format of the data.
- **compression** (*string*) – [Optional] The compression type of the data source. Possible values include GZIP and NONE. The default value is NONE. This setting is ignored for Google Cloud Bigtable, Google Cloud Datastore backups and Avro formats.
- **skip_leading_rows** (*int*) – Number of rows to skip when loading from a CSV.
- **field_delimiter** (*string*) – The delimiter to use for the CSV.
- **max_bad_records** (*int*) – The maximum number of bad records that BigQuery can ignore when running the job.
- **quote_character** (*string*) – The value that is used to quote data sections in a CSV file.
- **allow_quoted_newlines** (*boolean*) – Whether to allow quoted newlines (true) or not (false).
- **allow_jagged_rows** (*bool*) – Accept rows that are missing trailing optional columns. The missing values are treated as nulls. If false, records with missing trailing columns are treated as bad records, and if there are too many bad records, an invalid error is returned in the job result. Only applicable to CSV, ignored for other formats.
- **bigquery_conn_id** (*string*) – Reference to a specific BigQuery hook.
- **google_cloud_storage_conn_id** (*string*) – Reference to a specific Google cloud storage hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **src_fmt_configs** (*dict*) – configure optional fields specific to the source format

class airflow.contrib.operators.bigquery_operator.**BigQueryOperator**(*bql=None*, *sql=None*, *destination_dataset_table=False*, *write_disposition='WRITE_EMPTY'*, *allow_large_results=False*, *flatten_results=False*, *bigquery_conn_id='bigquery_default'*, *delegate_to=None*, *udf_config=False*, *use_legacy_sql=True*, *maximum_billing_tier=None*, *maximum_bytes_billed=None*, *create_disposition='CREATE_IF_NEEDED'*, *schema_update_options=()*, *query_params=None*, *priority='INTERACTIVE'*, *time_partitioning={}*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes BigQuery SQL queries in a specific BigQuery database

Parameters

- **bql** (*Can receive a str representing a sql statement, a list of str (sql statements), or reference to a template file. Template reference are recognized by str ending in '.sql'.*) – (Deprecated. Use *sql* parameter instead) the sql code to be executed (templated)
- **sql** (*Can receive a str representing a sql statement, a list of str (sql statements), or reference to a template file. Template reference are recognized by str ending in '.sql'.*) – the sql code to be executed (templated)
- **destination_dataset_table** (*string*) – A dotted (<project>.|<project>:)<dataset>.<table> that, if set, will store the results of the query. (templated)
- **write_disposition** (*string*) – Specifies the action that occurs if the destination table already exists. (default: 'WRITE_EMPTY')
- **create_disposition** (*string*) – Specifies whether the job is allowed to create new tables. (default: 'CREATE_IF_NEEDED')
- **allow_large_results** (*boolean*) – Whether to allow large results.
- **flatten_results** (*boolean*) – If true and query uses legacy SQL dialect, flattens all nested and repeated fields in the query results. `allow_large_results` must be `true`

if this is set to `false`. For standard SQL queries, this flag is ignored and results are never flattened.

- **bigquery_conn_id** (*string*) – reference to a specific BigQuery hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **udf_config** (*list*) – The User Defined Function configuration for the query. See https://cloud.google.com/bigquery/user-defined-functions for details.
- **use_legacy_sql** (*boolean*) – Whether to use legacy SQL (true) or standard SQL (false).
- **maximum_billing_tier** (*integer*) – Positive integer that serves as a multiplier of the basic price. Defaults to None, in which case it uses the value set in the project.
- **maximum_bytes_billed** (*float*) – Limits the bytes billed for this job. Queries that will have bytes billed beyond this limit will fail (without incurring a charge). If unspecified, this will be set to your project default.
- **schema_update_options** (*tuple*) – Allows the schema of the destination table to be updated as a side effect of the load job.
- **query_params** (*dict*) – a dictionary containing query parameter types and values, passed to BigQuery.
- **priority** (*string*) – Specifies a priority for the query. Possible values include INTERACTIVE and BATCH. The default value is INTERACTIVE.
- **time_partitioning** (*dict*) – configure optional time partitioning fields i.e. partition by field, type and expiration as per API specifications. Note that 'field' is not available in conjunction with dataset.table$partition.

class airflow.contrib.operators.bigquery_table_delete_operator.**BigQueryTableDeleteOperator**

Bases: *airflow.models.BaseOperator*

Deletes BigQuery tables

Parameters

- **deletion_dataset_table** (*string*) – A dotted (<project>.|<project>:)<dataset>.<table> that indicates which table will be deleted. (templated)
- **bigquery_conn_id** (*string*) – reference to a specific BigQuery hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **ignore_if_missing** (*boolean*) – if True, then return success even if the requested table does not exist.

class airflow.contrib.operators.bigquery_to_bigquery.**BigQueryToBigQueryOperator**(*source_project_dataset_tables*, *destination_project_dataset_table*, *write_disposition='WRITE_EMPTY'*, *create_disposition='CREATE_IF_NEEDED'*, *bigquery_conn_id='bigquery_default'*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Copies data from one BigQuery table to another.

See also:

For more details about these parameters: https://cloud.google.com/bigquery/docs/reference/v2/jobs#configuration.copy

Parameters

- **source_project_dataset_tables** (*list|string*) – One or more dotted (project:|project.)<dataset>.<table> BigQuery tables to use as the source data. If <project> is not included, project will be the project defined in the connection json. Use a list if there are multiple source tables. (templated)
- **destination_project_dataset_table** (*string*) – The destination BigQuery table. Format is: (project:|project.)<dataset>.<table> (templated)
- **write_disposition** (*string*) – The write disposition if the table already exists.
- **create_disposition** (*string*) – The create disposition if the table doesn't exist.
- **bigquery_conn_id** (*string*) – reference to a specific BigQuery hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

class airflow.contrib.operators.bigquery_to_gcs.**BigQueryToCloudStorageOperator**(*source_project_d*
destination_cloud_stora
compression='NONE',
export_format='CS
field_delimiter='
',
print_header=Tr
bigquery_conn_id=
delegate_to=None,
**args*,
***kwargs*)

Bases: *airflow.models.BaseOperator*

Transfers a BigQuery table to a Google Cloud Storage bucket.

See also:

For more details about these parameters: https://cloud.google.com/bigquery/docs/reference/v2/jobs

Parameters

- **source_project_dataset_table** (*string*) – The dotted (<project>.|<project>:)<dataset>.<table> BigQuery table to use as the source data. If <project> is not included, project will be the project defined in the connection json. (templated)
- **destination_cloud_storage_uris** (*list*) – The destination Google Cloud Storage URI (e.g. gs://some-bucket/some-file.txt). (templated) Follows convention defined here: https://cloud.google.com/bigquery/exporting-data-from-bigquery#exportingmultiple
- **compression** (*string*) – Type of compression to use.
- **export_format** – File format to export.
- **field_delimiter** (*string*) – The delimiter to use when extracting to a CSV.
- **print_header** (*boolean*) – Whether to print a header for a CSV file extract.
- **bigquery_conn_id** (*string*) – reference to a specific BigQuery hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

class airflow.contrib.operators.databricks_operator.**DatabricksSubmitRunOperator**(*json=None*, *spark_jar_task=*
note-
book_task=Non
new_cluster=N
ex-
ist-
ing_cluster_id=
li-
braries=None,
run_name=Non
time-
out_seconds=N
databricks_con
*polling_period_*
databricks_retr
do_xcom_push=
***kwargs*)

Bases: *airflow.models.BaseOperator*

Submits an Spark job run to Databricks using the api/2.0/jobs/runs/submit API endpoint.

There are two ways to instantiate this operator.

In the first way, you can take the JSON payload that you typically use to call the `api/2.0/jobs/runs/submit` endpoint and pass it directly to our `DatabricksSubmitRunOperator` through the `json` parameter. For example

```
json = {
  'new_cluster': {
    'spark_version': '2.1.0-db3-scala2.11',
    'num_workers': 2
  },
  'notebook_task': {
    'notebook_path': '/Users/airflow@example.com/PrepareData',
  },
}
notebook_run = DatabricksSubmitRunOperator(task_id='notebook_run', json=json)
```

Another way to accomplish the same thing is to use the named parameters of the `DatabricksSubmitRunOperator` directly. Note that there is exactly one named parameter for each top level parameter in the `runs/submit` endpoint. In this method, your code would look like this:

```
new_cluster = {
  'spark_version': '2.1.0-db3-scala2.11',
  'num_workers': 2
}
notebook_task = {
  'notebook_path': '/Users/airflow@example.com/PrepareData',
}
notebook_run = DatabricksSubmitRunOperator(
    task_id='notebook_run',
    new_cluster=new_cluster,
    notebook_task=notebook_task)
```

In the case where both the json parameter **AND** the named parameters are provided, they will be merged together. If there are conflicts during the merge, the named parameters will take precedence and override the top level `json` keys.

Currently the named parameters that `DatabricksSubmitRunOperator` supports are

- `spark_jar_task`
- `notebook_task`
- `new_cluster`
- `existing_cluster_id`
- `libraries`
- `run_name`
- `timeout_seconds`

Parameters

- **`json`** (*`dict`*) – A JSON object containing API parameters which will be passed directly to the `api/2.0/jobs/runs/submit` endpoint. The other named parameters (i.e. `spark_jar_task`, `notebook_task`..) to this operator will be merged with this json dictionary if they are provided. If there are conflicts during the merge, the named parameters will take precedence and override the top level json keys. (templated)

  See also:

  For more information about templating see *Jinja Templating*. https://docs.databricks.com/api/latest/jobs.html#runs-submit

- **`spark_jar_task`** (*`dict`*) – The main class and parameters for the JAR task. Note that the actual JAR is specified in the `libraries`. *EITHER* `spark_jar_task` *OR* `notebook_task` should be specified. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/jobs.html#jobssparkjartask

- **`notebook_task`** (*`dict`*) – The notebook path and parameters for the notebook task. *EITHER* `spark_jar_task` *OR* `notebook_task` should be specified. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/jobs.html#jobsnotebooktask

- **`new_cluster`** (*`dict`*) – Specs for a new cluster on which this task will be run. *EITHER* `new_cluster` *OR* `existing_cluster_id` should be specified. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/jobs.html#jobsclusterspecnewcluster

- **`existing_cluster_id`** (*`string`*) – ID for existing cluster on which to run this task. *EITHER* `new_cluster` *OR* `existing_cluster_id` should be specified. This field will be templated.

- **`libraries`** (*`list of dicts`*) – Libraries which this run will use. This field will be templated.

  See also:

  https://docs.databricks.com/api/latest/libraries.html#managedlibrarieslibrary

- **run_name** (*string*) – The run name used for this task. By default this will be set to the Airflow `task_id`. This `task_id` is a required parameter of the superclass `BaseOperator`. This field will be templated.
- **timeout_seconds** (*int32*) – The timeout for this run. By default a value of 0 is used which means to have no timeout. This field will be templated.
- **databricks_conn_id** (*string*) – The name of the Airflow connection to use. By default and in the common case this will be `databricks_default`. To use token based authentication, provide the key `token` in the extra field for the connection.
- **polling_period_seconds** (*int*) – Controls the rate which we poll for the result of this run. By default the operator will poll every 30 seconds.
- **databricks_retry_limit** (*int*) – Amount of times retry if the Databricks backend is unreachable. Its value must be greater than or equal to 1.
- **do_xcom_push** (*boolean*) – Whether we should push run_id and run_page_url to xcom.

class airflow.contrib.operators.dataflow_operator.**DataFlowJavaOperator**(*jar*, *dataflow_default_options=N* *options=None*, *gcp_conn_id='google_cloud* *delegate_to=None*, *poll_sleep=10*, *job_class=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Java Cloud DataFlow batch job. The parameters of the operation will be passed to the job.

It's a good practice to define dataflow_* parameters in the default_args of the dag like the project, zone and staging location.

```
default_args = {
    'dataflow_default_options': {
        'project': 'my-gcp-project',
        'zone': 'europe-west1-d',
        'stagingLocation': 'gs://my-staging-bucket/staging/'
    }
}
```

You need to pass the path to your dataflow as a file reference with the `jar` parameter, the jar needs to be a self executing jar (see documentation here: https://beam.apache.org/documentation/runners/dataflow/#self-executing-jar). Use `options` to pass on options to your job.

```
t1 = DataFlowOperation(
    task_id='datapflow_example',
    jar='{{var.value.gcp_dataflow_base}}pipeline/build/libs/pipeline-example-1.0.
↪jar',
    options={
        'autoscalingAlgorithm': 'BASIC',
        'maxNumWorkers': '50',
        'start': '{{ds}}',
        'partitionType': 'DAY',
```

(continues on next page)

(continued from previous page)

```
        'labels': {'foo' : 'bar'}
    },
    gcp_conn_id='gcp-airflow-service-account',
    dag=my-dag)
```

Both `jar` and `options` are templated so you can use variables in them.

class airflow.contrib.operators.dataflow_operator.**DataflowTemplateOperator**(*template*, *dataflow_default_options=None*, *parameters=None*, *gcp_conn_id='google_cloud_default'*, *delegate_to=None*, *poll_sleep=10*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Templated Cloud DataFlow batch job. The parameters of the operation will be passed to the job. It's a good practice to define dataflow_* parameters in the default_args of the dag like the project, zone and staging location.

See also:

https://cloud.google.com/dataflow/docs/reference/rest/v1b3/LaunchTemplateParameters https://cloud.google.com/dataflow/docs/reference/rest/v1b3/RuntimeEnvironment

```
default_args = {
    'dataflow_default_options': {
        'project': 'my-gcp-project'
        'zone': 'europe-west1-d',
        'tempLocation': 'gs://my-staging-bucket/staging/'
        }
    }
}
```

You need to pass the path to your dataflow template as a file reference with the `template` parameter. Use `parameters` to pass on parameters to your job. Use `environment` to pass on runtime environment variables to your job.

```
t1 = DataflowTemplateOperator(
    task_id='datapflow_example',
    template='{{var.value.gcp_dataflow_base}}',
    parameters={
        'inputFile': "gs://bucket/input/my_input.txt",
        'outputFile': "gs://bucket/output/my_output.txt"
    },
    gcp_conn_id='gcp-airflow-service-account',
    dag=my-dag)
```

`template`, `dataflow_default_options` and `parameters` are templated so you can use variables in them.

class airflow.contrib.operators.dataflow_operator.**DataFlowPythonOperator**(*py_file*, *py_options=None*, *dataflow_default_options=* *options=None*, *gcp_conn_id='google_cl* *delegate_to=None*, *poll_sleep=10*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

execute(*context*)

Execute the python dataflow job.

class airflow.contrib.operators.dataproc_operator.**DataprocClusterCreateOperator**(*cluster_name*, *project_id*, *num_workers*, *zone*, *network_uri=None*, *subnetwork_uri=None*, *internal_ip_only=N*, *tags=None*, *storage_bucket=No*, *init_actions_ur*, *init_action_tim*, *metadata=None*, *image_version=N*, *properties=None*, *master_machin*, *standard-4'*, *master_disk_size=5*, *worker_machin*, *standard-4'*, *worker_disk_si*, *num_preemptib*, *labels=None*, *region='global'*, *gcp_conn_id='*, *delegate_to=None*, *service_account=*, *service_account_s*, *idle_delete_ttl=*, *auto_delete_tim*, *auto_delete_ttl=*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Create a new cluster on Google Cloud Dataproc. The operator will wait until the creation is successful or an error occurs in the creation process.

The parameters allow to configure the cluster. Please refer to

https://cloud.google.com/dataproc/docs/reference/rest/v1/projects.regions.clusters

for a detailed explanation on the different parameters. Most of the configuration parameters detailed in the link are available as a parameter to this operator.

Parameters

- **cluster_name** (*string*) – The name of the DataProc cluster to create. (templated)
- **project_id** (*string*) – The ID of the google cloud project in which to create the cluster. (templated)
- **num_workers** (*int*) – The # of workers to spin up
- **storage_bucket** (*string*) – The storage bucket to use, setting to None lets dataproc generate a custom one for you
- **init_actions_uris** (*list[string]*) – List of GCS uri's containing dataproc initialization scripts
- **init_action_timeout** (*string*) – Amount of time executable scripts in init_actions_uris has to complete
- **metadata** (*dict*) – dict of key-value google compute engine metadata entries to add to all instances
- **image_version** (*string*) – the version of software inside the Dataproc cluster
- **properties** (*dict*) – dict of properties to set on config files (e.g. spark-defaults.conf), see https://cloud.google.com/dataproc/docs/reference/rest/v1/projects.regions.clusters#SoftwareConfig
- **master_machine_type** (*string*) – Compute engine machine type to use for the master node
- **master_disk_size** (*int*) – Disk size for the master node
- **worker_machine_type** (*string*) – Compute engine machine type to use for the worker nodes
- **worker_disk_size** (*int*) – Disk size for the worker nodes
- **num_preemptible_workers** (*int*) – The # of preemptible worker nodes to spin up
- **labels** (*dict*) – dict of labels to add to the cluster
- **zone** (*string*) – The zone where the cluster will be located. (templated)
- **network_uri** (*string*) – The network uri to be used for machine communication, cannot be specified with subnetwork_uri
- **subnetwork_uri** (*string*) – The subnetwork uri to be used for machine communication, cannot be specified with network_uri
- **internal_ip_only** (*bool*) – If true, all instances in the cluster will only have internal IP addresses. This can only be enabled for subnetwork enabled networks
- **tags** (*list[string]*) – The GCE tags to add to all instances
- **region** – leave as 'global', might become relevant in the future. (templated)
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.

- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **service_account** (*string*) – The service account of the dataproc instances.
- **service_account_scopes** (*list[string]*) – The URIs of service account scopes to be included.
- **idle_delete_ttl** (*int*) – The longest duration that cluster would keep alive while staying idle. Passing this threshold will cause cluster to be auto-deleted. A duration in seconds.
- **auto_delete_time** (*datetime.datetime*) – The time when cluster will be auto-deleted.
- **auto_delete_ttl** (*int*) – The life duration of cluster, the cluster will be auto-deleted at the end of this duration. A duration in seconds. (If auto_delete_time is set this parameter will be ignored)

class airflow.contrib.operators.dataproc_operator.**DataprocClusterScaleOperator**(*cluster_name*, *project_id*, *region='global'*, *gcp_conn_id='g*
delegate_to=None, *num_workers=2*, *num_preemptible*
graceful_decommissio
**args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Scale, up or down, a cluster on Google Cloud Dataproc. The operator will wait until the cluster is re-scaled.

Example:

t1 = DataprocClusterScaleOperator(task_id='dataproc_scale', project_id='my-project', cluster_name='cluster-1', num_workers=10, num_preemptible_workers=10, graceful_decommission_timeout='1h' dag=dag)

See also:

For more detail on about scaling clusters have a look at the reference: https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/scaling-clusters

Parameters

- **cluster_name** (*string*) – The name of the cluster to scale. (templated)
- **project_id** (*string*) – The ID of the google cloud project in which the cluster runs. (templated)
- **region** (*string*) – The region for the dataproc cluster. (templated)
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **num_workers** (*int*) – The new number of workers
- **num_preemptible_workers** (*int*) – The new number of preemptible workers

- **graceful_decommission_timeout** (*string*) – Timeout for graceful YARN decomissioning. Maximum value is 1d
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

class airflow.contrib.operators.dataproc_operator.**DataprocClusterDeleteOperator**(*cluster_name*, *project_id*, *region='global'*, *gcp_conn_id='*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Delete a cluster on Google Cloud Dataproc. The operator will wait until the cluster is destroyed.

Parameters

- **cluster_name** (*string*) – The name of the cluster to create. (templated)
- **project_id** (*string*) – The ID of the google cloud project in which the cluster runs. (templated)
- **region** (*string*) – leave as 'global', might become relevant in the future. (templated)
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

class airflow.contrib.operators.dataproc_operator.**DataProcPigOperator**(*query=None*, *query_uri=None*, *variables=None*, *job_name='{{task.task_id}}_{*, *cluster_name='cluster-1'*, *dataproc_pig_properties=None*, *dataproc_pig_jars=None*, *gcp_conn_id='google_cloud_*, *delegate_to=None*, *region='global'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Pig query Job on a Cloud DataProc cluster. The parameters of the operation will be passed to the cluster.

It's a good practice to define dataproc_* parameters in the default_args of the dag like the cluster name and UDFs.

```
default_args = {
    'cluster_name': 'cluster-1',
    'dataproc_pig_jars': [
        'gs://example/udf/jar/datafu/1.2.0/datafu.jar',
        'gs://example/udf/jar/gpig/1.2/gpig.jar'
    ]
}
```

You can pass a pig script as string or file reference. Use variables to pass on variables for the pig script to be resolved on the cluster or use the parameters to be resolved in the script as template parameters.

Example:

```
t1 = DataProcPigOperator(
        task_id='dataproc_pig',
        query='a_pig_script.pig',
        variables={'out': 'gs://example/output/{{ds}}'},
        dag=dag)
```

See also:

For more detail on about job submission have a look at the reference: https://cloud.google.com/dataproc/reference/rest/v1/projects.regions.jobs

Parameters

- **query** (*string*) – The query or reference to the query file (pg or pig extension). (templated)
- **query_uri** (*string*) – The uri of a pig script on Cloud Storage.
- **variables** (*dict*) – Map of named parameters for the query. (templated)
- **job_name** (*string*) – The job name used in the DataProc cluster. This name by default is the task_id appended with the execution data, but can be templated. The name will always be appended with a random number to avoid name clashes. (templated)
- **cluster_name** (*string*) – The name of the DataProc cluster. (templated)
- **dataproc_pig_properties** (*dict*) – Map for the Pig properties. Ideal to put in default arguments
- **dataproc_pig_jars** (*list*) – URIs to jars provisioned in Cloud Storage (example: for UDFs and libs) and are ideal to put in default arguments.
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **region** (*string*) – The specified region where the dataproc cluster is created.

class airflow.contrib.operators.dataproc_operator.**DataProcHiveOperator**(*query=None*, *query_uri=None*, *variables=None*, *job_name='{{task.task_id}}_*, *cluster_name='cluster-1'*, *dataproc_hive_properties=None*, *dataproc_hive_jars=None*, *gcp_conn_id='google_cloud*, *delegate_to=None*, *region='global'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Hive query Job on a Cloud DataProc cluster.

Parameters

- **query** (*string*) – The query or reference to the query file (q extension).
- **query_uri** (*string*) – The uri of a hive script on Cloud Storage.
- **variables** (*dict*) – Map of named parameters for the query.
- **job_name** (*string*) – The job name used in the DataProc cluster. This name by default is the task_id appended with the execution data, but can be templated. The name will always be appended with a random number to avoid name clashes.
- **cluster_name** (*string*) – The name of the DataProc cluster.
- **dataproc_hive_properties** (*dict*) – Map for the Pig properties. Ideal to put in default arguments
- **dataproc_hive_jars** (*list*) – URIs to jars provisioned in Cloud Storage (example: for UDFs and libs) and are ideal to put in default arguments.
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **region** (*string*) – The specified region where the dataproc cluster is created.

class airflow.contrib.operators.dataproc_operator.**DataProcSparkSqlOperator**(*query=None*, *query_uri=None*, *variables=None*, *job_name='{{task.task*, *cluster_name='cluster-1'*, *dataproc_spark_propertie*, *dataproc_spark_jars=Non*, *gcp_conn_id='google_*, *delegate_to=None*, *region='global'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Spark SQL query Job on a Cloud DataProc cluster.

Parameters

- **query** (*string*) – The query or reference to the query file (q extension). (templated)
- **query_uri** (*string*) – The uri of a spark sql script on Cloud Storage.
- **variables** (*dict*) – Map of named parameters for the query. (templated)
- **job_name** (*string*) – The job name used in the DataProc cluster. This name by default is the task_id appended with the execution data, but can be templated. The name will always be appended with a random number to avoid name clashes. (templated)
- **cluster_name** (*string*) – The name of the DataProc cluster. (templated)
- **dataproc_spark_properties** (*dict*) – Map for the Pig properties. Ideal to put in default arguments
- **dataproc_spark_jars** (*list*) – URIs to jars provisioned in Cloud Storage (example: for UDFs and libs) and are ideal to put in default arguments.
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **region** (*string*) – The specified region where the dataproc cluster is created.

class airflow.contrib.operators.dataproc_operator.**DataProcSparkOperator**(*main_jar=None*, *main_class=None*, *arguments=None*, *archives=None*, *files=None*, *job_name='{{task.task_id}}'*, *cluster_name='cluster-1'*, *dataproc_spark_properties=N*, *dataproc_spark_jars=None*, *gcp_conn_id='google_clou*, *delegate_to=None*, *region='global'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Spark Job on a Cloud DataProc cluster.

Parameters

- **main_jar** (*string*) – URI of the job jar provisioned on Cloud Storage. (use this or the main_class, not both together).
- **main_class** (*string*) – Name of the job class. (use this or the main_jar, not both together).
- **arguments** (*list*) – Arguments for the job. (templated)
- **archives** (*list*) – List of archived files that will be unpacked in the work directory. Should be stored in Cloud Storage.
- **files** (*list*) – List of files to be copied to the working directory
- **job_name** (*string*) – The job name used in the DataProc cluster. This name by default is the task_id appended with the execution data, but can be templated. The name will always be appended with a random number to avoid name clashes. (templated)
- **cluster_name** (*string*) – The name of the DataProc cluster. (templated)
- **dataproc_spark_properties** (*dict*) – Map for the Pig properties. Ideal to put in default arguments
- **dataproc_spark_jars** (*list*) – URIs to jars provisioned in Cloud Storage (example: for UDFs and libs) and are ideal to put in default arguments.
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **region** (*string*) – The specified region where the dataproc cluster is created.

class airflow.contrib.operators.dataproc_operator.**DataProcHadoopOperator**(*main_jar=None*, *main_class=None*, *arguments=None*, *archives=None*, *files=None*, *job_name='{{task.task_id*, *cluster_name='cluster-1'*, *dataproc_hadoop_properties*, *dataproc_hadoop_jars=None*, *gcp_conn_id='google_cloud*, *delegate_to=None*, *region='global'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Hadoop Job on a Cloud DataProc cluster.

Parameters

- **main_jar** (*string*) – URI of the job jar provisioned on Cloud Storage. (use this or the main_class, not both together).
- **main_class** (*string*) – Name of the job class. (use this or the main_jar, not both together).
- **arguments** (*list*) – Arguments for the job. (templated)
- **archives** (*list*) – List of archived files that will be unpacked in the work directory. Should be stored in Cloud Storage.
- **files** (*list*) – List of files to be copied to the working directory
- **job_name** (*string*) – The job name used in the DataProc cluster. This name by default is the task_id appended with the execution data, but can be templated. The name will always be appended with a random number to avoid name clashes. (templated)
- **cluster_name** (*string*) – The name of the DataProc cluster. (templated)
- **dataproc_hadoop_properties** (*dict*) – Map for the Pig properties. Ideal to put in default arguments
- **dataproc_hadoop_jars** (*list*) – URIs to jars provisioned in Cloud Storage (example: for UDFs and libs) and are ideal to put in default arguments.
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **region** (*string*) – The specified region where the dataproc cluster is created.

class airflow.contrib.operators.dataproc_operator.**DataProcPySparkOperator**(*main*, *arguments=None*, *archives=None*, *pyfiles=None*, *files=None*, *job_name='{{task.task_*, *cluster_name='cluster-1'*, *dataproc_pyspark_properti*, *dataproc_pyspark_jars=No*, *gcp_conn_id='google_c*, *delegate_to=None*, *region='global'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a PySpark Job on a Cloud DataProc cluster.

Parameters

- **main** (*string*) – [Required] The Hadoop Compatible Filesystem (HCFS) URI of the main Python file to use as the driver. Must be a .py file.
- **arguments** (*list*) – Arguments for the job. (templated)
- **archives** (*list*) – List of archived files that will be unpacked in the work directory. Should be stored in Cloud Storage.
- **files** (*list*) – List of files to be copied to the working directory
- **pyfiles** (*list*) – List of Python files to pass to the PySpark framework. Supported file types: .py, .egg, and .zip
- **job_name** (*string*) – The job name used in the DataProc cluster. This name by default is the task_id appended with the execution data, but can be templated. The name will always be appended with a random number to avoid name clashes. (templated)
- **cluster_name** (*string*) – The name of the DataProc cluster.
- **dataproc_pyspark_properties** (*dict*) – Map for the Pig properties. Ideal to put in default arguments
- **dataproc_pyspark_jars** (*list*) – URIs to jars provisioned in Cloud Storage (example: for UDFs and libs) and are ideal to put in default arguments.
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **region** (*string*) – The specified region where the dataproc cluster is created.

class airflow.contrib.operators.dataproc_operator.**DataprocWorkflowTemplateBaseOperator**(*proje*
re-
gion=
*gcp_*
del-
e-
*gate_*
**args*
***kw*

Bases: *airflow.models.BaseOperator*

class airflow.contrib.operators.dataproc_operator.**DataprocWorkflowTemplateInstantiateOperat**

Bases: *airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateBaseOperator*

Instantiate a WorkflowTemplate on Google Cloud Dataproc. The operator will wait until the WorkflowTemplate is finished executing.

See also:

Please refer to: https://cloud.google.com/dataproc/docs/reference/rest/v1beta2/projects.regions.workflowTemplates/instantiate

Parameters

- **template_id** (*string*) – The id of the template. (templated)
- **project_id** (*string*) – The ID of the google cloud project in which the template runs
- **region** (*string*) – leave as 'global', might become relevant in the future
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

class airflow.contrib.operators.dataproc_operator.**DataprocWorkflowTemplateInstantiateInline**

Bases: *airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateBaseOperator*

Instantiate a WorkflowTemplate Inline on Google Cloud Dataproc. The operator will wait until the WorkflowTemplate is finished executing.

See also:

Please refer to: https://cloud.google.com/dataproc/docs/reference/rest/v1beta2/projects.regions.workflowTemplates/instantiateInline

Parameters

- **template** (*map*) – The template contents. (templated)
- **project_id** (*string*) – The ID of the google cloud project in which the template runs
- **region** (*string*) – leave as 'global', might become relevant in the future
- **gcp_conn_id** (*string*) – The connection ID to use connecting to Google Cloud Platform.

- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

class airflow.contrib.operators.datastore_export_operator.**DatastoreExportOperator**(*bucket*, *namespace=None*, *datastore_conn_id*, *cloud_storage*, *delegate_to=None*, *entity_filter=None*, *labels=None*, *polling_interval*, *overwrite_existing*, *xcom_push=*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Export entities from Google Cloud Datastore to Cloud Storage

Parameters

- **bucket** (*string*) – name of the cloud storage bucket to backup data
- **namespace** (*str*) – optional namespace path in the specified Cloud Storage bucket to backup data. If this namespace does not exist in GCS, it will be created.
- **datastore_conn_id** (*string*) – the name of the Datastore connection id to use
- **cloud_storage_conn_id** (*string*) – the name of the cloud storage connection id to force-write backup
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **entity_filter** (*dict*) – description of what data from the project is included in the export, refer to https://cloud.google.com/datastore/docs/reference/rest/Shared.Types/EntityFilter
- **labels** (*dict*) – client-assigned labels for cloud storage
- **polling_interval_in_seconds** (*int*) – number of seconds to wait before polling for execution status again
- **overwrite_existing** (*bool*) – if the storage bucket + namespace is not empty, it will be emptied prior to exports. This enables overwriting existing backups.
- **xcom_push** (*bool*) – push operation name to xcom for reference

class airflow.contrib.operators.datastore_import_operator.**DatastoreImportOperator**(*bucket*, *file*, *namespace=None*, *entity_filter=N*, *labels=None*, *datastore_conn_id*, *delegate_to=No*, *polling_inter*, *xcom_push=*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Import entities from Cloud Storage to Google Cloud Datastore

Parameters

- **bucket** (*string*) – container in Cloud Storage to store data
- **file** (*string*) – path of the backup metadata file in the specified Cloud Storage bucket. It should have the extension .overall_export_metadata
- **namespace** (*str*) – optional namespace of the backup metadata file in the specified Cloud Storage bucket.
- **entity_filter** (*dict*) – description of what data from the project is included in the export, refer to https://cloud.google.com/datastore/docs/reference/rest/Shared.Types/EntityFilter
- **labels** (*dict*) – client-assigned labels for cloud storage
- **datastore_conn_id** (*string*) – the name of the connection id to use
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **polling_interval_in_seconds** (*int*) – number of seconds to wait before polling for execution status again
- **xcom_push** (*bool*) – push operation name to xcom for reference

class airflow.contrib.operators.discord_webhook_operator.**DiscordWebhookOperator**(*http_conn_id=*, *webhook_endpoint=*, *message=''*, *username=None*, *avatar_url=No*, *tts=False*, *proxy=None*, **args*, ***kwargs*)

Bases: *airflow.operators.http_operator.SimpleHttpOperator*

This operator allows you to post messages to Discord using incoming webhooks. Takes a Discord connection ID with a default relative webhook endpoint. The default endpoint can be overridden using the webhook_endpoint parameter (https://discordapp.com/developers/docs/resources/webhook).

Each Discord webhook can be pre-configured to use a specific username and avatar_url. You can override these defaults in this operator.

Parameters

- **http_conn_id** (*str*) – Http connection ID with host as "https://discord.com/api/" and default webhook endpoint in the extra field in the form of {"webhook_endpoint": "webhooks/{webhook.id}/{webhook.token}"}
- **webhook_endpoint** (*str*) – Discord webhook endpoint in the form of "webhooks/{webhook.id}/{webhook.token}"
- **message** (*str*) – The message you want to send to your Discord channel (max 2000 characters). (templated)
- **username** (*str*) – Override the default username of the webhook. (templated)
- **avatar_url** (*str*) – Override the default avatar of the webhook
- **tts** (*bool*) – Is a text-to-speech message
- **proxy** (*str*) – Proxy to use to make the Discord webhook call

execute(*context*)

Call the DiscordWebhookHook to post message

class airflow.contrib.operators.ecs_operator.**ECSOperator**(*task_definition*, *cluster*, *overrides*, *aws_conn_id=None*, *region_name=None*, *launch_type='EC2'*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Execute a task on AWS EC2 Container Service

Parameters

- **task_definition** (*str*) – the task definition name on EC2 Container Service
- **cluster** (*str*) – the cluster name on EC2 Container Service
- **aws_conn_id** (*str*) – connection id of AWS credentials / region name. If None, credential boto3 strategy will be used (http://boto3.readthedocs.io/en/latest/guide/configuration.html).
- **region_name** – region name to use in AWS Hook. Override the region_name in connection (if provided)
- **launch_type** – the launch type on which to run your task ('EC2' or 'FARGATE')

Param overrides: the same parameter that boto3 will receive (templated): http://boto3.readthedocs.org/en/latest/reference/services/ecs.html#ECS.Client.run_task

Type overrides: dict

Type launch_type: str

class airflow.contrib.operators.emr_add_steps_operator.**EmrAddStepsOperator**(*job_flow_id*, *aws_conn_id='s3_defa*, *steps=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

An operator that adds steps to an existing EMR job_flow.

Parameters

- **job_flow_id** – id of the JobFlow to add steps to. (templated)
- **aws_conn_id** (*str*) – aws connection to uses
- **steps** (*list*) – boto3 style steps to be added to the jobflow. (templated)

class airflow.contrib.operators.emr_create_job_flow_operator.**EmrCreateJobFlowOperator**(*aws_c*, *emr_c*, *job_flo*, **args*, ***kwa*

Bases: *airflow.models.BaseOperator*

Creates an EMR JobFlow, reading the config from the EMR connection. A dictionary of JobFlow overrides can be passed that override the config from the connection.

Parameters

- **aws_conn_id** (*str*) – aws connection to uses
- **emr_conn_id** (*str*) – emr connection to use
- **job_flow_overrides** – boto3 style arguments to override emr_connection extra. (templated)

class airflow.contrib.operators.emr_terminate_job_flow_operator.**EmrTerminateJobFlowOperator**

Bases: *airflow.models.BaseOperator*

Operator to terminate EMR JobFlows.

Parameters

- **job_flow_id** – id of the JobFlow to terminate. (templated)
- **aws_conn_id** (*str*) – aws connection to uses

class airflow.contrib.operators.file_to_gcs.**FileToGoogleCloudStorageOperator**(*src*, *dst*, *bucket*, *google_cloud_stora*, *mime_type='applic*, *stream'*, *del-*, *e-*, *gate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Uploads a file to Google Cloud Storage

Parameters

- **src** (*string*) – Path to the local file. (templated)
- **dst** (*string*) – Destination path within the specified bucket. (templated)
- **bucket** (*string*) – The bucket to upload to. (templated)
- **google_cloud_storage_conn_id** (*string*) – The Airflow connection ID to upload with
- **mime_type** (*string*) – The mime-type string
- **delegate_to** (*string*) – The account to impersonate, if any

execute(*context*)
Uploads the file to Google cloud storage

class airflow.contrib.operators.gcp_container_operator.**GKEClusterCreateOperator**(*project_id*, *location*, *body={}*, *gcp_conn_id='*, *api_version='v*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

class airflow.contrib.operators.gcp_container_operator.**GKEClusterDeleteOperator**(*project_id*, *name*, *location*, *gcp_conn_id='*, *api_version='v*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

class airflow.contrib.operators.gcs_download_operator.**GoogleCloudStorageDownloadOperator**(*bu*, *ob*, *je*, *fil*, *na*, *st*, *go*, *de*, *e-*, *go*, **a*, ****)

Bases: *airflow.models.BaseOperator*

Downloads a file from Google Cloud Storage.

Parameters

- **bucket** (*string*) – The Google cloud storage bucket where the object is. (templated)

- **object** (*string*) – The name of the object to download in the Google cloud storage bucket. (templated)
- **filename** (*string*) – The file path on the local file system (where the operator is being executed) that the file should be downloaded to. (templated) If no filename passed, the downloaded data will not be stored on the local file system.
- **store_to_xcom_key** (*string*) – If this param is set, the operator will push the contents of the downloaded file to XCom with the key set in this parameter. If not set, the downloaded data will not be pushed to XCom. (templated)
- **google_cloud_storage_conn_id** (*string*) – The connection ID to use when connecting to Google cloud storage.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

class airflow.contrib.operators.gcs_list_operator.**GoogleCloudStorageListOperator**(*bucket*, *prefix=None*, *delimiter=None*, *google_cloud_*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

List all objects from the bucket with the give string prefix and delimiter in name.

This operator returns a python list with the name of objects which can be used by *xcom* in the downstream task.

Parameters

- **bucket** (*string*) – The Google cloud storage bucket to find the objects. (templated)
- **prefix** (*string*) – Prefix string which filters objects whose name begin with this prefix. (templated)
- **delimiter** (*string*) – The delimiter by which you want to filter the objects. (templated) For e.g to lists the CSV files from in a directory in GCS you would use delimiter='.csv'.
- **google_cloud_storage_conn_id** (*string*) – The connection ID to use when connecting to Google cloud storage.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

Example: The following Operator would list all the Avro files from `sales/sales-2017` folder in `data` bucket.

```
GCS_Files = GoogleCloudStorageListOperator(
    task_id='GCS_Files',
    bucket='data',
    prefix='sales/sales-2017/',
    delimiter='.avro',
```

(continues on next page)

(continued from previous page)

```
    google_cloud_storage_conn_id=google_cloud_conn_id
)
```

class airflow.contrib.operators.gcs_operator.**GoogleCloudStorageCreateBucketOperator**(*bucket_na stor- age_class lo- ca- tion='US project_id la- bels=Non google_cl del- e- gate_to=N *args, **kwargs*)

Bases: *airflow.models.BaseOperator*

Creates a new bucket. Google Cloud Storage uses a flat namespace, so you can't create a bucket with a name that is already in use.

See also:

For more information, see Bucket Naming Guidelines: https://cloud.google.com/storage/docs/bucketnaming.html#requirements

Parameters

- **bucket_name** (*string*) – The name of the bucket. (templated)
- **storage_class** (*string*) – This defines how objects in the bucket are stored and determines the SLA and the cost of storage (templated). Values include
  - MULTI_REGIONAL
  - REGIONAL
  - STANDARD
  - NEARLINE
  - COLDLINE.

  If this value is not specified when the bucket is created, it will default to STANDARD.
- **location** (*string*) – The location of the bucket. (templated) Object data for objects in the bucket resides in physical storage within this region. Defaults to US.

  See also:

  https://developers.google.com/storage/docs/bucket-locations
- **project_id** (*string*) – The ID of the GCP Project. (templated)
- **labels** (*dict*) – User-provided labels, in key/value pairs.
- **google_cloud_storage_conn_id** (*string*) – The connection ID to use when connecting to Google cloud storage.

- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

Example: The following Operator would create a new bucket `test-bucket` with `MULTI_REGIONAL` storage class in `EU` region

```
CreateBucket = GoogleCloudStorageCreateBucketOperator(
    task_id='CreateNewBucket',
    bucket_name='test-bucket',
    storage_class='MULTI_REGIONAL',
    location='EU',
    labels={'env': 'dev', 'team': 'airflow'},
    google_cloud_storage_conn_id='airflow-service-account'
)
```

class airflow.contrib.operators.gcs_to_bq.**GoogleCloudStorageToBigQueryOperator**(*bucket*, *source_objects*, *destination_project_dataset_table*, *schema_fields=None*, *schema_object=None*, *source_format='CSV'*, *compression='NONE'*, *create_disposition='CREATE_IF_NEEDED'*, *skip_leading_rows=0*, *write_disposition='WRITE_EMPTY'*, *field_delimiter=','*, *max_bad_records=0*, *quote_character=None*, *ignore_unknown_values=False*, *allow_quoted_newlines=False*, *allow_jagged_rows=False*, *max_id_key=None*, *bigquery_conn_id='bigquery_default'*, *google_cloud_storage_conn_id='google_cloud_default'*, *delegate_to=None*, *schema_update_options=()*, *src_fmt_configs={}*, *external_table=False*, *time_partitioning={}*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Loads files from Google cloud storage into BigQuery.

The schema to be used for the BigQuery table may be specified in one of two ways. You may either directly pass the schema fields in, or you may point the operator to a Google cloud storage object name. The object in Google cloud storage must be a JSON file with the schema fields in it.

Parameters

- **bucket** (*string*) – The bucket to load from. (templated)
- **source_objects** – List of Google cloud storage URIs to load from. (templated) If source_format is 'DATASTORE_BACKUP', the list must only contain a single URI.
- **destination_project_dataset_table** (*string*) – The dotted

(<project>.)<dataset>.<table> BigQuery table to load data into. If <project> is not included, project will be the project defined in the connection json. (templated)

- **schema_fields** (*list*) – If set, the schema field list as defined here: https://cloud.google.com/bigquery/docs/reference/v2/jobs#configuration.load Should not be set when source_format is 'DATASTORE_BACKUP'.
- **schema_object** – If set, a GCS object path pointing to a .json file that contains the schema for the table. (templated)
- **schema_object** – string
- **source_format** (*string*) – File format to export.
- **compression** (*string*) – [Optional] The compression type of the data source. Possible values include GZIP and NONE. The default value is NONE. This setting is ignored for Google Cloud Bigtable, Google Cloud Datastore backups and Avro formats.
- **create_disposition** (*string*) – The create disposition if the table doesn't exist.
- **skip_leading_rows** (*int*) – Number of rows to skip when loading from a CSV.
- **write_disposition** (*string*) – The write disposition if the table already exists.
- **field_delimiter** (*string*) – The delimiter to use when loading from a CSV.
- **max_bad_records** (*int*) – The maximum number of bad records that BigQuery can ignore when running the job.
- **quote_character** (*string*) – The value that is used to quote data sections in a CSV file.
- **ignore_unknown_values** (*bool*) – [Optional] Indicates if BigQuery should allow extra values that are not represented in the table schema. If true, the extra values are ignored. If false, records with extra columns are treated as bad records, and if there are too many bad records, an invalid error is returned in the job result.
- **allow_quoted_newlines** (*boolean*) – Whether to allow quoted newlines (true) or not (false).
- **allow_jagged_rows** (*bool*) – Accept rows that are missing trailing optional columns. The missing values are treated as nulls. If false, records with missing trailing columns are treated as bad records, and if there are too many bad records, an invalid error is returned in the job result. Only applicable to CSV, ignored for other formats.
- **max_id_key** (*string*) – If set, the name of a column in the BigQuery table that's to be loaded. Thsi will be used to select the MAX value from BigQuery after the load occurs. The results will be returned by the execute() command, which in turn gets stored in XCom for future operators to use. This can be helpful with incremental loads–during future executions, you can pick up from the max ID.
- **bigquery_conn_id** (*string*) – Reference to a specific BigQuery hook.
- **google_cloud_storage_conn_id** (*string*) – Reference to a specific Google cloud storage hook.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **schema_update_options** (*list*) – Allows the schema of the destination table to be updated as a side effect of the load job.
- **src_fmt_configs** (*dict*) – configure optional fields specific to the source format

- **external_table** (*bool*) – Flag to specify if the destination table should be a BigQuery external table. Default Value is False.
- **time_partitioning** (*dict*) – configure optional time partitioning fields i.e. partition by field, type and expiration as per API specifications. Note that 'field' is not available in concurrency with dataset.table$partition.

class airflow.contrib.operators.gcs_to_gcs.**GoogleCloudStorageToGoogleCloudStorageOperator**(

Bases: *airflow.models.BaseOperator*

Copies objects from a bucket to another, with renaming if requested.

Parameters

- **source_bucket** (*string*) – The source Google cloud storage bucket where the object is. (templated)
- **source_object** (*string*) – The source name of the object to copy in the Google cloud storage bucket. (templated) If wildcards are used in this argument:

  You can use only one wildcard for objects (filenames) within your bucket. The wildcard can appear inside the object name or at the end of the object name. Appending a wildcard to the bucket name is unsupported.
- **destination_bucket** – The destination Google cloud storage bucket

where the object should be. (templated) :type destination_bucket: string :param destination_object: The destination name of the object in the

destination Google cloud storage bucket. (templated) If a wildcard is supplied in the source_object argument, this is the prefix that will be prepended to the final destination objects' paths. Note that the source path's part before the wildcard will be removed; if it needs to be retained it should be appended to destination_object. For example, with prefix `foo/*` and destination_object *'blah/'*, the file `foo/baz` will be copied to `blah/baz`; to retain the prefix write the destination_object as e.g. `blah/foo`, in which case the copied file will be named `blah/foo/baz`.

Parameters **move_object** – When move object is True, the object is moved instead

of copied to the new location. This is the equivalent of a mv command as opposed to a cp command.

Parameters

- **google_cloud_storage_conn_id** (*string*) – The connection ID to use when connecting to Google cloud storage.

- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

Examples: The following Operator would copy a single file named `sales/sales-2017/january.avro` in the `data` bucket to the file named `copied_sales/2017/january-backup.avro` in the `data_backup` bucket

```
copy_single_file = GoogleCloudStorageToGoogleCloudStorageOperator(
    task_id='copy_single_file',
    source_bucket='data',
    source_object='sales/sales-2017/january.avro',
    destination_bucket='data_backup',
    destination_object='copied_sales/2017/january-backup.avro',
    google_cloud_storage_conn_id=google_cloud_conn_id
)
```

The following Operator would copy all the Avro files from `sales/sales-2017` folder (i.e. with names starting with that prefix) in `data` bucket to the `copied_sales/2017` folder in the `data_backup` bucket.

```
copy_files = GoogleCloudStorageToGoogleCloudStorageOperator(
    task_id='copy_files',
    source_bucket='data',
    source_object='sales/sales-2017/*.avro',
    destination_bucket='data_backup',
    destination_object='copied_sales/2017/',
    google_cloud_storage_conn_id=google_cloud_conn_id
)
```

The following Operator would move all the Avro files from `sales/sales-2017` folder (i.e. with names starting with that prefix) in `data` bucket to the same folder in the `data_backup` bucket, deleting the original files in the process.

```
move_files = GoogleCloudStorageToGoogleCloudStorageOperator(
    task_id='move_files',
    source_bucket='data',
    source_object='sales/sales-2017/*.avro',
    destination_bucket='data_backup',
    move_object=True,
    google_cloud_storage_conn_id=google_cloud_conn_id
)
```

class airflow.contrib.operators.gcs_to_s3.**GoogleCloudStorageToS3Operator**(*bucket*, *prefix=None*, *delimiter=None*, *google_cloud_storage_co*, *delegate_to=None*, *dest_aws_conn_id=None*, *dest_s3_key=None*, *replace=False*, **args*, ***kwargs*)

Bases: *airflow.contrib.operators.gcs_list_operator.GoogleCloudStorageListOperator*

Synchronizes a Google Cloud Storage bucket with an S3 bucket.

Parameters

- **bucket** (*string*) – The Google Cloud Storage bucket to find the objects. (templated)
- **prefix** (*string*) – Prefix string which filters objects whose name begin with this prefix. (templated)
- **delimiter** (*string*) – The delimiter by which you want to filter the objects. (templated) For e.g to lists the CSV files from in a directory in GCS you would use delimiter='.csv'.
- **google_cloud_storage_conn_id** (*string*) – The connection ID to use when connecting to Google Cloud Storage.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **dest_aws_conn_id** (*str*) – The destination S3 connection
- **dest_s3_key** (*str*) – The base S3 key to be used to store the files. (templated)

class airflow.contrib.operators.hipchat_operator.**HipChatAPIOperator**(*token*, *base_url='https://api.hipchat.com*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Base HipChat Operator. All derived HipChat operators reference from HipChat's official REST API documentation at https://www.hipchat.com/docs/apiv2. Before using any HipChat API operators you need to get an authentication token at https://www.hipchat.com/docs/apiv2/auth. In the future additional HipChat operators will be derived from this class as well.

Parameters

- **token** (*str*) – HipChat REST API authentication token
- **base_url** (*str*) – HipChat REST API base url.

prepare_request()

Used by the execute function. Set the request method, url, and body of HipChat's REST API call. Override in child class. Each HipChatAPI child operator is responsible for having a prepare_request method call which sets self.method, self.url, and self.body.

class airflow.contrib.operators.hipchat_operator.**HipChatAPISendRoomNotificationOperator**(*roo*
mes
sag
**ar*
***k*

Bases: *airflow.contrib.operators.hipchat_operator.HipChatAPIOperator*

Send notification to a specific HipChat room. More info: https://www.hipchat.com/docs/apiv2/method/send_room_notification

Parameters

- **room_id** (*str*) – Room in which to send notification on HipChat. (templated)
- **message** (*str*) – The message body. (templated)
- **frm** (*str*) – Label to be shown in addition to sender's name
- **message_format** (*str*) – How the notification is rendered: html or text
- **color** (*str*) – Background color of the msg: yellow, green, red, purple, gray, or random
- **attach_to** (*str*) – The message id to attach this notification to
- **notify** (*bool*) – Whether this message should trigger a user notification
- **card** (*dict*) – HipChat-defined card object

prepare_request()

Used by the execute function. Set the request method, url, and body of HipChat's REST API call. Override in child class. Each HipChatAPI child operator is responsible for having a prepare_request method call which sets self.method, self.url, and self.body.

class airflow.contrib.operators.mlengine_operator.**MLEngineBatchPredictionOperator**(*project_id*,
job_id,
re-
gion,
data_format
in-
put_paths,
out-
put_path,
model_name
ver-
sion_name=
uri=None,
max_worker
run-
time_version
gcp_conn_id
del-
e-
gate_to=No
**args*,
***kwargs*)

Bases: *airflow.models.BaseOperator*

Start a Google Cloud ML Engine prediction job.

NOTE: For model origin, users should consider exactly one from the three options below: 1. Populate 'uri' field only, which should be a GCS location that points to a tensorflow savedModel directory. 2. Populate

'model_name' field only, which refers to an existing model, and the default version of the model will be used. 3. Populate both 'model_name' and 'version_name' fields, which refers to a specific version of a specific model.

In options 2 and 3, both model and version name should contain the minimal identifier. For instance, call

```
MLEngineBatchPredictionOperator(
    ...,
    model_name='my_model',
    version_name='my_version',
    ...)
```

if the desired model version is "projects/my_project/models/my_model/versions/my_version".

See https://cloud.google.com/ml-engine/reference/rest/v1/projects.jobs for further documentation on the parameters.

Parameters

- **project_id** (*string*) – The Google Cloud project name where the prediction job is submitted. (templated)
- **job_id** (*string*) – A unique id for the prediction job on Google Cloud ML Engine. (templated)
- **data_format** (*string*) – The format of the input data. It will default to 'DATA_FORMAT_UNSPECIFIED' if is not provided or is not one of ["TEXT", "TF_RECORD", "TF_RECORD_GZIP"].
- **input_paths** (*list of string*) – A list of GCS paths of input data for batch prediction. Accepting wildcard operator *, but only at the end. (templated)
- **output_path** (*string*) – The GCS path where the prediction results are written to. (templated)
- **region** (*string*) – The Google Compute Engine region to run the prediction job in. (templated)
- **model_name** (*string*) – The Google Cloud ML Engine model to use for prediction. If version_name is not provided, the default version of this model will be used. Should not be None if version_name is provided. Should be None if uri is provided. (templated)
- **version_name** (*string*) – The Google Cloud ML Engine model version to use for prediction. Should be None if uri is provided. (templated)
- **uri** (*string*) – The GCS path of the saved model to use for prediction. Should be None if model_name is provided. It should be a GCS path pointing to a tensorflow SavedModel. (templated)
- **max_worker_count** (*int*) – The maximum number of workers to be used for parallel processing. Defaults to 10 if not specified.
- **runtime_version** (*string*) – The Google Cloud ML Engine runtime version to use for batch prediction.
- **gcp_conn_id** (*string*) – The connection ID used for connection to Google Cloud Platform.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have doamin-wide delegation enabled.

Raises: `ValueError`: if a unique model/version origin cannot be determined.

class airflow.contrib.operators.mlengine_operator.**MLEngineModelOperator**(*project_id*, *model*, *operation='create'*, *gcp_conn_id='google_clou*
delegate_to=None, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Operator for managing a Google Cloud ML Engine model.

Parameters

- **project_id** (*string*) – The Google Cloud project name to which MLEngine model belongs. (templated)
- **model** (*dict*) – A dictionary containing the information about the model. If the *operation* is *create*, then the *model* parameter should contain all the information about this model such as *name*.

  If the *operation* is *get*, the *model* parameter should contain the *name* of the model.
- **operation** – The operation to perform. Available operations are:
  - create: Creates a new model as provided by the *model* parameter.
  - get: Gets a particular model where the name is specified in *model*.
- **gcp_conn_id** (*string*) – The connection ID to use when fetching connection info.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

class airflow.contrib.operators.mlengine_operator.**MLEngineVersionOperator**(*project_id*, *model_name*, *version_name=None*, *version=None*, *operation='create'*, *gcp_conn_id='google_c*
delegate_to=None, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Operator for managing a Google Cloud ML Engine version.

Parameters

- **project_id** (*string*) – The Google Cloud project name to which MLEngine model belongs.
- **model_name** (*string*) – The name of the Google Cloud ML Engine model that the version belongs to. (templated)
- **version_name** (*string*) – A name to use for the version being operated upon. If not None and the *version* argument is None or does not have a value for the *name* key, then this will be populated in the payload for the *name* key. (templated)
- **version** (*dict*) – A dictionary containing the information about the version. If the *operation* is *create*, *version* should contain all the information about this version such as name, and deploymentUrl. If the *operation* is *get* or *delete*, the *version* parameter should contain the *name* of the version. If it is None, the only *operation* possible would be *list*. (templated)
- **operation** (*string*) – The operation to perform. Available operations are:
  - `create`: Creates a new version in the model specified by *model_name*, in which case the *version* parameter should contain all the information to create that version (e.g. *name*, *deploymentUrl*).
  - `get`: Gets full information of a particular version in the model specified by *model_name*. The name of the version should be specified in the *version* parameter.
  - `list`: Lists all available versions of the model specified by *model_name*.
  - `delete`: Deletes the version specified in *version* parameter from the model specified by *model_name*). The name of the version should be specified in the *version* parameter.
- **gcp_conn_id** (*string*) – The connection ID to use when fetching connection info.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

class airflow.contrib.operators.mlengine_operator.**MLEngineTrainingOperator**(*project_id*, *job_id*, *package_uris*, *training_python_module*, *training_args*, *region*, *scale_tier=None*, *runtime_version=None*, *python_version=None*, *job_dir=None*, *gcp_conn_id='google_*, *delegate_to=None*, *mode='PRODUCTION*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Operator for launching a MLEngine training job.

Parameters

- **project_id** (*string*) – The Google Cloud project name within which MLEngine training job should run (templated).
- **job_id** (*string*) – A unique templated id for the submitted Google MLEngine training job. (templated)
- **package_uris** (*string*) – A list of package locations for MLEngine training job, which should include the main training program + any additional dependencies. (templated)
- **training_python_module** (*string*) – The Python module name to run within MLEngine training job after installing 'package_uris' packages. (templated)
- **training_args** (*string*) – A list of templated command line arguments to pass to the MLEngine training program. (templated)
- **region** (*string*) – The Google Compute Engine region to run the MLEngine training job in (templated).
- **scale_tier** (*string*) – Resource tier for MLEngine training job. (templated)
- **runtime_version** (*string*) – The Google Cloud ML runtime version to use for training. (templated)
- **python_version** (*string*) – The version of Python used in training. (templated)
- **job_dir** (*string*) – A Google Cloud Storage path in which to store training outputs and other data needed for training. (templated)
- **gcp_conn_id** (*string*) – The connection ID to use when fetching connection info.
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **mode** (*string*) – Can be one of 'DRY_RUN'/'CLOUD'. In 'DRY_RUN' mode, no real training job will be launched, but the MLEngine training job request will be printed out. In 'CLOUD' mode, a real MLEngine training job creation request will be issued.

class airflow.contrib.operators.pubsub_operator.**PubSubTopicCreateOperator**(*project*, *topic*, *fail_if_exists=False*, *gcp_conn_id='google_c*
del-
e-
gate_to=None, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Create a PubSub topic.

By default, if the topic already exists, this operator will not cause the DAG to fail.

```
with DAG('successful DAG') as dag:
    (
        dag
        >> PubSubTopicCreateOperator(project='my-project',
                                     topic='my_new_topic')
        >> PubSubTopicCreateOperator(project='my-project',
                                     topic='my_new_topic')
    )
```

The operator can be configured to fail if the topic already exists.

```
with DAG('failing DAG') as dag:
    (
        dag
        >> PubSubTopicCreateOperator(project='my-project',
                                     topic='my_new_topic')
        >> PubSubTopicCreateOperator(project='my-project',
                                     topic='my_new_topic',
                                     fail_if_exists=True)
    )
```

Both `project` and `topic` are templated so you can use variables in them.

class airflow.contrib.operators.pubsub_operator.**PubSubTopicDeleteOperator**(*project*, *topic*, *fail_if_not_exists=False*, *gcp_conn_id='google_c*
del-
e-
gate_to=None, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Delete a PubSub topic.

By default, if the topic does not exist, this operator will not cause the DAG to fail.

```
with DAG('successful DAG') as dag:
    (
        dag
        >> PubSubTopicDeleteOperator(project='my-project',
                                     topic='non_existing_topic')
    )
```

The operator can be configured to fail if the topic does not exist.

```
with DAG('failing DAG') as dag:
    (
        dag
        >> PubSubTopicCreateOperator(project='my-project',
                                     topic='non_existing_topic',
                                     fail_if_not_exists=True)
    )
```

Both `project` and `topic` are templated so you can use variables in them.

class airflow.contrib.operators.pubsub_operator.**PubSubSubscriptionCreateOperator**(*topic_project*, *topic*, *subscription=None*, *subscription_project=*, *ack_deadline_*, *fail_if_exists=*, *gcp_conn_id=*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Create a PubSub subscription.

By default, the subscription will be created in `topic_project`. If `subscription_project` is specified and the GCP credentials allow, the Subscription can be created in a different project from its topic.

By default, if the subscription already exists, this operator will not cause the DAG to fail. However, the topic must exist in the project.

```
with DAG('successful DAG') as dag:
    (
        dag
        >> PubSubSubscriptionCreateOperator(
            topic_project='my-project', topic='my-topic',
            subscription='my-subscription')
        >> PubSubSubscriptionCreateOperator(
            topic_project='my-project', topic='my-topic',
            subscription='my-subscription')
    )
```

The operator can be configured to fail if the subscription already exists.

```
with DAG('failing DAG') as dag:
    (
        dag
        >> PubSubSubscriptionCreateOperator(
            topic_project='my-project', topic='my-topic',
            subscription='my-subscription')
        >> PubSubSubscriptionCreateOperator(
            topic_project='my-project', topic='my-topic',
            subscription='my-subscription', fail_if_exists=True)
    )
```

Finally, subscription is not required. If not passed, the operator will generated a universally unique identifier for the subscription's name.

```
with DAG('DAG') as dag:
    (
        dag >> PubSubSubscriptionCreateOperator(
            topic_project='my-project', topic='my-topic')
    )
```

`topic_project`, `topic`, `subscription`, and `subscription` are templated so you can use variables in them.

class airflow.contrib.operators.pubsub_operator.**PubSubSubscriptionDeleteOperator**(*project*, *subscription*, *fail_if_not_ex*, *gcp_conn_id=*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Delete a PubSub subscription.

By default, if the subscription does not exist, this operator will not cause the DAG to fail.

```
with DAG('successful DAG') as dag:
    (
        dag
        >> PubSubSubscriptionDeleteOperator(project='my-project',
                                            subscription='non-existing')
    )
```

The operator can be configured to fail if the subscription already exists.

```
with DAG('failing DAG') as dag:
    (
        dag
        >> PubSubSubscriptionDeleteOperator(
             project='my-project', subscription='non-existing',
             fail_if_not_exists=True)
    )
```

`project`, and `subscription` are templated so you can use variables in them.

class airflow.contrib.operators.pubsub_operator.**PubSubPublishOperator**(*project*, *topic*, *messages*, *gcp_conn_id='google_cloud_*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Publish messages to a PubSub topic.

Each Task publishes all provided messages to the same topic in a single GCP project. If the topic does not exist, this task will fail.

```
from base64 import b64encode as b64e

m1 = {'data': b64e('Hello, World!'),
      'attributes': {'type': 'greeting'}
```

(continues on next page)

(continued from previous page)

```
        }
    m2 = {'data': b64e('Knock, knock')}
    m3 = {'attributes': {'foo': ''}}

    t1 = PubSubPublishOperator(
        project='my-project',topic='my_topic',
        messages=[m1, m2, m3],
        create_topic=True,
        dag=dag)

``project`` , ``topic``, and ``messages`` are templated so you can use
```

variables in them.

class airflow.contrib.operators.s3_list_operator.**S3ListOperator**(*bucket*, *prefix=''*, *delimiter=''*, *aws_conn_id='aws_default'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

List all objects from the bucket with the given string prefix in name.

This operator returns a python list with the name of objects which can be used by *xcom* in the downstream task.

Parameters

- **bucket** (*string*) – The S3 bucket where to find the objects. (templated)
- **prefix** (*string*) – Prefix string to filters the objects whose name begin with such prefix. (templated)
- **delimiter** (*string*) – the delimiter marks key hierarchy. (templated)
- **aws_conn_id** (*string*) – The connection ID to use when connecting to S3 storage.

Example: The following operator would list all the files (excluding subfolders) from the S3 `customers/2018/04/` key in the `data` bucket.

```
s3_file = S3ListOperator(
    task_id='list_3s_files',
    bucket='data',
    prefix='customers/2018/04/',
    delimiter='/',
    aws_conn_id='aws_customers_conn'
)
```

class airflow.contrib.operators.s3_to_gcs_operator.**S3ToGoogleCloudStorageOperator**(*bucket*, *prefix=''*, *delimiter=''*, *aws_conn_id*, *dest_gcs_co*, *dest_gcs=No*, *delegate_to=Non*, *replace=False*, **args*, ***kwargs*)

Bases: *airflow.contrib.operators.s3_list_operator.S3ListOperator*

Synchronizes an S3 key, possibly a prefix, with a Google Cloud Storage destination path.

Parameters

- **bucket** (*string*) – The S3 bucket where to find the objects. (templated)
- **prefix** (*string*) – Prefix string which filters objects whose name begin with such prefix. (templated)
- **delimiter** (*string*) – the delimiter marks key hierarchy. (templated)
- **aws_conn_id** (*string*) – The source S3 connection
- **dest_gcs_conn_id** (*string*) – The destination connection ID to use when connecting to Google Cloud Storage.
- **dest_gcs** (*string*) – The destination Google Cloud Storage bucket and prefix where you want to store the files. (templated)
- **delegate_to** (*string*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.
- **replace** (*bool*) – Whether you want to replace existing destination files or not.

Example: .. code-block:: python

s3_to_gcs_op = S3ToGoogleCloudStorageOperator(task_id='s3_to_gcs_example', bucket='my-s3-bucket', prefix='data/customers-201804', dest_gcs_conn_id='google_cloud_default', dest_gcs='gs://my.gcs.bucket/some/customers/', replace=False, dag=my-dag)

Note that `bucket`, `prefix`, `delimiter` and `dest_gcs` are templated, so you can use variables in them if you wish.

class airflow.contrib.operators.slack_webhook_operator.**SlackWebhookOperator**(*http_conn_id=None*, *webhook_token=None*, *message=''*, *channel=None*, *username=None*, *icon_emoji=None*, *link_names=False*, *proxy=None*, **args*, ***kwargs*)

Bases: *airflow.operators.http_operator.SimpleHttpOperator*

This operator allows you to post messages to Slack using incoming webhooks. Takes both Slack webhook token directly and connection that has Slack webhook token. If both supplied, Slack webhook token will be used.

Each Slack webhook token can be pre-configured to use a specific channel, username and icon. You can override these defaults in this hook.

Parameters

- **conn_id** (*str*) – connection that has Slack webhook token in the extra field
- **webhook_token** (*str*) – Slack webhook token
- **message** (*str*) – The message you want to send on Slack
- **channel** (*str*) – The channel the message should be posted to
- **username** (*str*) – The username to post to slack with
- **icon_emoji** (*str*) – The emoji to use as icon for the user posting to Slack
- **link_names** (*bool*) – Whether or not to find and link channel and usernames in your message
- **proxy** (*str*) – Proxy to use to make the Slack webhook call

execute(*context*)

Call the SparkSqlHook to run the provided sql query

class airflow.contrib.operators.spark_sql_operator.**SparkSqlOperator**(*sql*, *conf=None*, *conn_id='spark_sql_default'*, *total_executor_cores=None*, *executor_cores=None*, *executor_memory=None*, *keytab=None*, *principal=None*, *master='yarn'*, *name='default-name'*, *num_executors=None*, *yarn_queue='default'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Execute Spark SQL query

Parameters

- **sql** (*str*) – The SQL query to execute. (templated)
- **conf** (*str (format: PROP=VALUE)*) – arbitrary Spark configuration property
- **conn_id** (*str*) – connection_id string
- **total_executor_cores** (*int*) – (Standalone & Mesos only) Total cores for all executors (Default: all the available cores on the worker)
- **executor_cores** (*int*) – (Standalone & YARN only) Number of cores per executor (Default: 2)
- **executor_memory** (*str*) – Memory per executor (e.g. 1000M, 2G) (Default: 1G)
- **keytab** (*str*) – Full path to the file that contains the keytab
- **master** (*str*) – spark://host:port, mesos://host:port, yarn, or local
- **name** (*str*) – Name of the job
- **num_executors** (*int*) – Number of executors to launch
- **verbose** (*bool*) – Whether to pass the verbose flag to spark-sql
- **yarn_queue** (*str*) – The YARN queue to submit to (Default: "default")

execute(*context*)

Call the SparkSqlHook to run the provided sql query

class airflow.contrib.operators.sqoop_operator.**SqoopOperator**(*conn_id='sqoop_default'*, *cmd_type='import'*, *table=None*, *query=None*, *target_dir=None*, *append=None*, *file_type='text'*, *columns=None*, *num_mappers=None*, *split_by=None*, *where=None*, *export_dir=None*, *input_null_string=None*, *input_null_non_string=None*, *staging_table=None*, *clear_staging_table=False*, *enclosed_by=None*, *escaped_by=None*, *input_fields_terminated_by=None*, *input_lines_terminated_by=None*, *input_optionally_enclosed_by=None*, *batch=False*, *direct=False*, *driver=None*, *verbose=False*, *relaxed_isolation=False*, *properties=None*, *hcatalog_database=None*, *hcatalog_table=None*, *create_hcatalog_table=False*, *extra_import_options=None*, *extra_export_options=None*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Execute a Sqoop job. Documentation for Apache Sqoop can be found here:

https://sqoop.apache.org/docs/1.4.2/SqoopUserGuide.html.

execute(*context*)

Execute sqoop job

class airflow.contrib.operators.vertica_operator.**VerticaOperator**(*sql*, *vertica_conn_id='vertica_default'*, **args*, ***kwargs*)

Bases: *airflow.models.BaseOperator*

Executes sql code in a specific Vertica database

Parameters

- **vertica_conn_id** (*string*) – reference to a specific Vertica database
- **sql** (*Can receive a str representing a sql statement, a list of str (sql statements), or reference to a template file. Template reference are recognized by str ending in '.sql'*) – the sql code to be executed. (templated)

Sensors

class airflow.contrib.sensors.aws_redshift_cluster_sensor.**AwsRedshiftClusterSensor**(*cluster_ide*
tar-
get_status=
*aws_conn_*
**args*,
***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a Redshift cluster to reach a specific status.

Parameters

- **cluster_identifier** (*str*) – The identifier for the cluster being pinged.
- **target_status** (*str*) – The cluster status desired.

poke(*context*)

Function that the sensors defined while deriving this class should override.

class airflow.contrib.sensors.bash_sensor.**BashSensor**(*bash_command*, *env=None*, *output_encoding='utf-8'*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Executes a bash command/script and returns True if and only if the return code is 0.

Parameters

- **bash_command** (*string*) – The command, set of commands or reference to a bash script (must be '.sh') to be executed.
- **env** (*dict*) – If env is not None, it must be a mapping that defines the environment variables for the new process; these are used instead of inheriting the current process environment, which is the default behavior. (templated)
- **output_encoding** (*string*) – output encoding of bash command.

poke(*context*)

Execute the bash command in a temporary directory which will be cleaned afterwards

class airflow.contrib.sensors.bigquery_sensor.**BigQueryTableSensor**(*project_id*, *dataset_id*, *table_id*, *bigquery_conn_id='bigquery_default_c*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Checks for the existence of a table in Google Bigquery.

param project_id The Google cloud project in which to look for the table. The connection supplied to the hook must provide access to the specified project.

type project_id string

param dataset_id The name of the dataset in which to look for the table. storage bucket.

type dataset_id string

param table_id The name of the table to check the existence of.

type table_id string

param bigquery_conn_id The connection ID to use when connecting to Google BigQuery.

type bigquery_conn_id string

param delegate_to The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

type delegate_to string

poke(*context*)

Function that the sensors defined while deriving this class should override.

class airflow.contrib.sensors.emr_base_sensor.**EmrBaseSensor**(*aws_conn_id='aws_default'*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Contains general sensor behavior for EMR. Subclasses should implement get_emr_response() and state_from_response() methods. Subclasses should also implement NON_TERMINAL_STATES and FAILED_STATE constants.

poke(*context*)

Function that the sensors defined while deriving this class should override.

class airflow.contrib.sensors.emr_job_flow_sensor.**EmrJobFlowSensor**(*job_flow_id*, **args*, ***kwargs*)

Bases: *airflow.contrib.sensors.emr_base_sensor.EmrBaseSensor*

Asks for the state of the JobFlow until it reaches a terminal state. If it fails the sensor errors, failing the task.

Parameters **job_flow_id** (*string*) – job_flow_id to check the state of

class airflow.contrib.sensors.emr_step_sensor.**EmrStepSensor**(*job_flow_id*, *step_id*, **args*, ***kwargs*)

Bases: *airflow.contrib.sensors.emr_base_sensor.EmrBaseSensor*

Asks for the state of the step until it reaches a terminal state. If it fails the sensor errors, failing the task.

Parameters

- **job_flow_id** (*string*) – job_flow_id which contains the step check the state of
- **step_id** (*string*) – step to check the state of

class airflow.contrib.sensors.file_sensor.**FileSensor**(*filepath*, *fs_conn_id='fs_default2'*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a file or folder to land in a filesystem.

If the path given is a directory then this sensor will only return true if any files exist inside it (either directly, or within a subdirectory)

Parameters

- **fs_conn_id** (*string*) – reference to the File (path) connection id
- **filepath** – File or folder name (relative to the base path set within the connection)

poke(*context*)

Function that the sensors defined while deriving this class should override.

class airflow.contrib.sensors.ftp_sensor.**FTPSensor**(*path*, *ftp_conn_id='ftp_default'*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Waits for a file or directory to be present on FTP.

Parameters

- **path** (*str*) – Remote file or directory path
- **ftp_conn_id** (*str*) – The connection to run the sensor against

poke(*context*)

Function that the sensors defined while deriving this class should override.

class airflow.contrib.sensors.ftp_sensor.**FTPSSensor**(*path*, *ftp_conn_id='ftp_default'*, **args*, ***kwargs*)

Bases: *airflow.contrib.sensors.ftp_sensor.FTPSensor*

Waits for a file or directory to be present on FTP over SSL.

class airflow.contrib.sensors.gcs_sensor.**GoogleCloudStorageObjectSensor**(*bucket*, *object*, *google_cloud_conn_id='go*
delegate_to=None, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Checks for the existence of a file in Google Cloud Storage. Create a new GoogleCloudStorageObjectSensor.

param bucket The Google cloud storage bucket where the object is.

type bucket string

param object The name of the object to check in the Google cloud storage bucket.

type object string

param google_cloud_storage_conn_id The connection ID to use when connecting to Google cloud storage.

type google_cloud_storage_conn_id string

param delegate_to The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

type delegate_to string

poke(*context*)
Function that the sensors defined while deriving this class should override.

class airflow.contrib.sensors.gcs_sensor.**GoogleCloudStorageObjectUpdatedSensor**(*bucket*, *object*, *ts_func=<functio ts_function>*, *google_cloud_co del- e- gate_to=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Checks if an object is updated in Google Cloud Storage. Create a new GoogleCloudStorageObjectUpdatedSensor.

param bucket The Google cloud storage bucket where the object is.

type bucket string

param object The name of the object to download in the Google cloud storage bucket.

type object string

param ts_func Callback for defining the update condition. The default callback returns execution_date + schedule_interval. The callback takes the context as parameter.

type ts_func function

param google_cloud_storage_conn_id The connection ID to use when connecting to Google cloud storage.

type google_cloud_storage_conn_id string

param delegate_to The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

type delegate_to string

poke(*context*)
Function that the sensors defined while deriving this class should override.

class airflow.contrib.sensors.gcs_sensor.**GoogleCloudStoragePrefixSensor**(*bucket*, *prefix*, *google_cloud_conn_id='go*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Checks for the existence of a files at prefix in Google Cloud Storage bucket. Create a new GoogleCloudStorageObjectSensor.

param bucket The Google cloud storage bucket where the object is.

type bucket string

param prefix The name of the prefix to check in the Google cloud storage bucket.

type prefix string

param google_cloud_storage_conn_id The connection ID to use when connecting to Google cloud storage.

type google_cloud_storage_conn_id string

param delegate_to The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation enabled.

type delegate_to string

poke(*context*)

Function that the sensors defined while deriving this class should override.

class airflow.contrib.sensors.pubsub_sensor.**PubSubPullSensor**(*project*, *subscription*, *max_messages=5*, *return_immediately=False*, *ack_messages=False*, *gcp_conn_id='google_cloud_default'*, *delegate_to=None*, **args*, ***kwargs*)

Bases: *airflow.sensors.base_sensor_operator.BaseSensorOperator*

Pulls messages from a PubSub subscription and passes them through XCom.

This sensor operator will pull up to `max_messages` messages from the specified PubSub subscription. When the subscription returns messages, the poke method's criteria will be fulfilled and the messages will be returned from the operator and passed through XCom for downstream tasks.

If `ack_messages` is set to True, messages will be immediately acknowledged before being returned, otherwise, downstream tasks will be responsible for acknowledging them.

`project` and `subscription` are templated so you can use variables in them.

execute(*context*)

Overridden to allow messages to be passed

poke(*context*)

Function that the sensors defined while deriving this class should override.

3.19.2 Macros

Here's a list of variables and macros that can be used in templates

3.19.2.1 Default Variables

The Airflow engine passes a few variables by default that are accessible in all templates

Variable	Description
{{ ds }}	the execution date as YYYY-MM-DD
{{ ds_nodash }}	the execution date as YYYYMMDD
{{ prev_ds }}	the previous execution date as YYYY-MM-DD. if {{ ds }} is 2016-01-08 and schedule_interval is @weekly, {{ prev_ds }} will be 2016-01-01.
{{ next_ds }}	the next execution date as YYYY-MM-DD. if {{ ds }} is 2016-01-01 and schedule_interval is @weekly, {{ prev_ds }} will be 2016-01-08.
{{ yesterday_ds }}	yesterday's date as YYYY-MM-DD
{{ yesterday_ds_nodash }}	yesterday's date as YYYYMMDD
{{ tomorrow_ds }}	tomorrow's date as YYYY-MM-DD
{{ tomorrow_ds_nodash }}	tomorrow's date as YYYYMMDD
{{ ts }}	same as execution_date.isoformat()
{{ ts_nodash }}	same as ts without - and :
{{ execution_date }}	the execution_date, (datetime.datetime)
{{ prev_execution_date }}	the previous execution date (if available) (datetime.datetime)
{{ next_execution_date }}	the next execution date (datetime.datetime)
{{ dag }}	the DAG object
{{ task }}	the Task object
{{ macros }}	a reference to the macros package, described below
{{ task_instance }}	the task_instance object
{{ end_date }}	same as {{ ds }}
{{ latest_date }}	same as {{ ds }}
{{ ti }}	same as {{ task_instance }}
{{ params }}	a reference to the user-defined params dictionary which can be overridden by the dictionary passed through trigger_dag -c if you enabled dag_run_conf_overrides_params` in ``airflow.cfg
{{ var.value.my_var }}	global defined variables represented as a dictionary
{{ var.json.my_var.path }}	global defined variables represented as a dictionary with deserialized JSON object, append the path to the key within the JSON object
{{ task_instance_key_str }}	a unique, human-readable key to the task instance formatted {dag_id}_{task_id}_{ds}

Note that you can access the object's attributes and methods with simple dot notation. Here are some examples of what is possible: `{{ task.owner }}`, `{{ task.task_id }}`, `{{ ti.hostname }}`, ... Refer to the models documentation for more information on the objects' attributes and methods.

The `var` template variable allows you to access variables defined in Airflow's UI. You can access them as either plain-text or JSON. If you use JSON, you are also able to walk nested structures, such as dictionaries like: `{{ var.json.my_dict_var.key1 }}`

3.19.2.2 Macros

Macros are a way to expose objects to your templates and live under the `macros` namespace in your templates.

A few commonly used libraries and methods are made available.

Variable	Description
`macros.datetime`	The standard lib's `datetime.datetime`
`macros.timedelta`	The standard lib's `datetime.timedelta`
`macros.dateutil`	A reference to the `dateutil` package
`macros.time`	The standard lib's `time`
`macros.uuid`	The standard lib's `uuid`
`macros.random`	The standard lib's `random`

Some airflow specific macros are also defined:

`airflow.macros.`**`ds_add`**(*ds*, *days*)
Add or subtract days from a YYYY-MM-DD

Parameters

- **ds** (*str*) – anchor date in `YYYY-MM-DD` format to add to
- **days** (*int*) – number of days to add to the ds, you can use negative values

```
>>> ds_add('2015-01-01', 5)
'2015-01-06'
>>> ds_add('2015-01-06', -5)
'2015-01-01'
```

`airflow.macros.`**`ds_format`**(*ds*, *input_format*, *output_format*)
Takes an input string and outputs another string as specified in the output format

Parameters

- **ds** (*str*) – input string which contains a date
- **input_format** (*str*) – input string format. E.g. %Y-%m-%d
- **output_format** (*str*) – output string format E.g. %Y-%m-%d

```
>>> ds_format('2015-01-01', "%Y-%m-%d", "%m-%d-%y")
'01-01-15'
>>> ds_format('1/5/2015', "%m/%d/%Y",  "%Y-%m-%d")
'2015-01-05'
```

`airflow.macros.`**`random`**() → x in the interval [0, 1).

`airflow.macros.hive.`**`closest_ds_partition`**(*table*, *ds*, *before=True*, *schema='default'*, *metastore_conn_id='metastore_default'*)
This function finds the date in a list closest to the target date. An optional parameter can be given to get the closest before or after.

Parameters

- **table** (*str*) – A hive table name
- **ds** (*datetime.date list*) – A datestamp `%Y-%m-%d` e.g. `yyyy-mm-dd`
- **before** (*bool or None*) – closest before (True), after (False) or either side of ds

Returns The closest date

Return type str or None

```
>>> tbl = 'airflow.static_babynames_partitioned'
>>> closest_ds_partition(tbl, '2015-01-02')
'2015-01-01'
```

airflow.macros.hive.**max_partition**(*table*, *schema='default'*, *field=None*, *filter_map=None*, *metastore_conn_id='metastore_default'*)

Gets the max partition for a table.

Parameters

- **schema** (*string*) – The hive schema the table lives in
- **table** (*string*) – The hive table you are interested in, supports the dot notation as in "my_database.my_table", if a dot is found, the schema param is disregarded
- **metastore_conn_id** (*string*) – The hive connection you are interested in. If your default is set you don't need to use this parameter.
- **filter_map** (*map*) – partition_key:partition_value map used for partition filtering, e.g. {'key1': 'value1', 'key2': 'value2'}. Only partitions matching all partition_key:partition_value pairs will be considered as candidates of max partition.
- **field** (*str*) – the field to get the max value from. If there's only one partition field, this will be inferred

```
>>> max_partition('airflow.static_babynames_partitioned')
'2015-01-01'
```

3.19.3 Models

Models are built on top of the SQLAlchemy ORM Base class, and instances are persisted in the database.

class airflow.models.**BaseOperator**(*task_id*, *owner='Airflow'*, *email=None*, *email_on_retry=True*, *email_on_failure=True*, *retries=0*, *retry_delay=datetime.timedelta(0, 300)*, *retry_exponential_backoff=False*, *max_retry_delay=None*, *start_date=None*, *end_date=None*, *schedule_interval=None*, *depends_on_past=False*, *wait_for_downstream=False*, *dag=None*, *params=None*, *default_args=None*, *adhoc=False*, *priority_weight=1*, *weight_rule=u'downstream'*, *queue='default'*, *pool=None*, *sla=None*, *execution_timeout=None*, *on_failure_callback=None*, *on_success_callback=None*, *on_retry_callback=None*, *trigger_rule=u'all_success'*, *resources=None*, *run_as_user=None*, *task_concurrency=None*, *executor_config=None*, *inlets=None*, *outlets=None*, **args*, ***kwargs*)

Bases: airflow.utils.log.logging_mixin.LoggingMixin

Abstract base class for all operators. Since operators create objects that become nodes in the dag, BaseOperator contains many recursive methods for dag crawling behavior. To derive this class, you are expected to override the constructor as well as the 'execute' method.

Operators derived from this class should perform or trigger certain tasks synchronously (wait for completion). Example of operators could be an operator that runs a Pig job (PigOperator), a sensor operator that waits for a partition to land in Hive (HiveSensorOperator), or one that moves data from Hive to MySQL (Hive2MySqlOperator). Instances of these operators (tasks) target specific operations, running specific scripts, functions or data transfers.

This class is abstract and shouldn't be instantiated. Instantiating a class derived from this one results in the creation of a task object, which ultimately becomes a node in DAG objects. Task dependencies should be set by using the set_upstream and/or set_downstream methods.

Parameters

- **task_id** (*string*) – a unique, meaningful id for the task
- **owner** (*string*) – the owner of the task, using the unix username is recommended
- **retries** (*int*) – the number of retries that should be performed before failing the task
- **retry_delay** (*timedelta*) – delay between retries
- **retry_exponential_backoff** (*bool*) – allow progressive longer waits between retries by using exponential backoff algorithm on retry delay (delay will be converted into seconds)
- **max_retry_delay** (*timedelta*) – maximum delay interval between retries
- **start_date** (*datetime*) – The start_date for the task, determines the execution_date for the first task instance. The best practice is to have the start_date rounded to your DAG's schedule_interval. Daily jobs have their start_date some day at 00:00:00, hourly jobs have their start_date at 00:00 of a specific hour. Note that Airflow simply looks at the latest execution_date and adds the schedule_interval to determine the next execution_date. It is also very important to note that different tasks' dependencies need to line up in time. If task A depends on task B and their start_date are offset in a way that their execution_date don't line up, A's dependencies will never be met. If you are looking to delay a task, for example running a daily task at 2AM, look into the TimeSensor and TimeDeltaSensor. We advise against using dynamic start_date and recommend using fixed ones. Read the FAQ entry about start_date for more information.
- **end_date** (*datetime*) – if specified, the scheduler won't go beyond this date
- **depends_on_past** (*bool*) – when set to true, task instances will run sequentially while relying on the previous task's schedule to succeed. The task instance for the start_date is allowed to run.
- **wait_for_downstream** (*bool*) – when set to true, an instance of task X will wait for tasks immediately downstream of the previous instance of task X to finish successfully before it runs. This is useful if the different instances of a task X alter the same asset, and this asset is used by tasks downstream of task X. Note that depends_on_past is forced to True wherever wait_for_downstream is used.
- **queue** (*str*) – which queue to target when running this job. Not all executors implement queue management, the CeleryExecutor does support targeting specific queues.
- **dag** (DAG) – a reference to the dag the task is attached to (if any)

- **priority_weight** (*int*) – priority weight of this task against other task. This allows the executor to trigger higher priority tasks before others when things get backed up.
- **weight_rule** (*str*) – weighting method used for the effective total priority weight of the task. Options are: `{ downstream | upstream | absolute }` default is `downstream` When set to `downstream` the effective weight of the task is the aggregate sum of all downstream descendants. As a result, upstream tasks will have higher weight and will be scheduled more aggressively when using positive weight values. This is useful when you have multiple dag run instances and desire to have all upstream tasks to complete for all runs before each dag can continue processing downstream tasks. When set to `upstream` the effective weight is the aggregate sum of all upstream ancestors. This is the opposite where downtream tasks have higher weight and will be scheduled more aggressively when using positive weight values. This is useful when you have multiple dag run instances and prefer to have each dag complete before starting upstream tasks of other dags. When set to `absolute`, the effective weight is the exact `priority_weight` specified without additional weighting. You may want to do this when you know exactly what priority weight each task should have. Additionally, when set to `absolute`, there is bonus effect of significantly speeding up the task creation process as for very large DAGS. Options can be set as string or using the constants defined in the static class `airflow.utils.WeightRule`
- **pool** (*str*) – the slot pool this task should run in, slot pools are a way to limit concurrency for certain tasks
- **sla** (*datetime.timedelta*) – time by which the job is expected to succeed. Note that this represents the `timedelta` after the period is closed. For example if you set an SLA of 1 hour, the scheduler would send an email soon after 1:00AM on the `2016-01-02` if the `2016-01-01` instance has not succeeded yet. The scheduler pays special attention for jobs with an SLA and sends alert emails for sla misses. SLA misses are also recorded in the database for future reference. All tasks that share the same SLA time get bundled in a single email, sent soon after that time. SLA notification are sent once and only once for each task instance.
- **execution_timeout** (*datetime.timedelta*) – max time allowed for the execution of this task instance, if it goes beyond it will raise and fail.
- **on_failure_callback** (*callable*) – a function to be called when a task instance of this task fails. a context dictionary is passed as a single parameter to this function. Context contains references to related objects to the task instance and is documented under the macros section of the API.
- **on_retry_callback** – much like the `on_failure_callback` except that it is executed when retries occur.
- **on_success_callback** (*callable*) – much like the `on_failure_callback` except that it is executed when the task succeeds.
- **trigger_rule** (*str*) – defines the rule by which dependencies are applied for the task to get triggered. Options are: `{ all_success | all_failed | all_done | one_success | one_failed | dummy}` default is `all_success`. Options can be set as string or using the constants defined in the static class `airflow.utils.TriggerRule`
- **resources** (*dict*) – A map of resource parameter names (the argument names of the Resources constructor) to their values.
- **run_as_user** (*str*) – unix username to impersonate while running the task
- **task_concurrency** (*int*) – When set, a task will be able to limit the concurrent runs across execution_dates

- **executor_config** (*dict*) – Additional task-level configuration parameters that are interpreted by a specific executor. Parameters are namespaced by the name of executor. ‘‘example: to run this task in a specific docker container through the KubernetesExecutor MyOperator(. . . ,

  executor_config={ “KubernetesExecutor”:

  {“image”: “myCustomDockerImage”} }

)‘‘

clear(***kwargs*)
Clears the state of task instances associated with the task, following the parameters specified.

dag
Returns the Operator’s DAG if set, otherwise raises an error

deps
Returns the list of dependencies for the operator. These differ from execution context dependencies in that they are specific to tasks and can be extended/overridden by subclasses.

downstream_list
@property: list of tasks directly downstream

execute(*context*)
This is the main method to derive when creating an operator. Context is the same dictionary used as when rendering jinja templates.

Refer to get_template_context for more context.

get_direct_relative_ids(*upstream=False*)
Get the direct relative ids to the current task, upstream or downstream.

get_direct_relatives(*upstream=False*)
Get the direct relatives to the current task, upstream or downstream.

get_flat_relative_ids(*upstream=False*, *found_descendants=None*)
Get a flat list of relatives’ ids, either upstream or downstream.

get_flat_relatives(*upstream=False*)
Get a flat list of relatives, either upstream or downstream.

get_task_instances(*session*, *start_date=None*, *end_date=None*)
Get a set of task instance related to this task for a specific date range.

has_dag()
Returns True if the Operator has been assigned to a DAG.

on_kill()
Override this method to cleanup subprocesses when a task instance gets killed. Any use of the threading, subprocess or multiprocessing module within an operator needs to be cleaned up or it will leave ghost processes behind.

post_execute(*context*, **args*, ***kwargs*)
This hook is triggered right after self.execute() is called. It is passed the execution context and any results returned by the operator.

pre_execute(*context*, **args*, ***kwargs*)
This hook is triggered right before self.execute() is called.

prepare_template()
Hook that is triggered after the templated fields get replaced by their content. If you need your operator to alter the content of the file before the template is rendered, it should override this method to do so.

render_template(*attr*, *content*, *context*)
Renders a template either from a file or directly in a field, and returns the rendered result.

render_template_from_field(*attr*, *content*, *context*, *jinja_env*)
Renders a template from a field. If the field is a string, it will simply render the string and return the result. If it is a collection or nested set of collections, it will traverse the structure and render all strings in it.

run(*start_date=None*, *end_date=None*, *ignore_first_depends_on_past=False*, *ignore_ti_state=False*, *mark_success=False*)
Run a set of task instances for a date range.

schedule_interval
The schedule interval of the DAG always wins over individual tasks so that tasks within a DAG always line up. The task still needs a schedule_interval as it may not be attached to a DAG.

set_downstream(*task_or_task_list*)
Set a task or a task list to be directly downstream from the current task.

set_upstream(*task_or_task_list*)
Set a task or a task list to be directly upstream from the current task.

upstream_list
@property: list of tasks directly upstream

xcom_pull(*context*, *task_ids=None*, *dag_id=None*, *key=u'return_value'*, *include_prior_dates=None*)
See TaskInstance.xcom_pull()

xcom_push(*context*, *key*, *value*, *execution_date=None*)
See TaskInstance.xcom_push()

class airflow.models.**Chart**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

class airflow.models.**Connection**(*conn_id=None*, *conn_type=None*, *host=None*, *login=None*, *password=None*, *schema=None*, *port=None*, *extra=None*, *uri=None*)
Bases: sqlalchemy.ext.declarative.api.Base, airflow.utils.log.logging_mixin.LoggingMixin

Placeholder to store information about different database instances connection information. The idea here is that scripts use references to database instances (conn_id) instead of hard coding hostname, logins and passwords when using operators or hooks.

extra_dejson
Returns the extra property by deserializing json.

class airflow.models.**DAG**(*dag_id*, *description=u''*, *schedule_interval=datetime.timedelta(1)*, *start_date=None*, *end_date=None*, *full_filepath=None*, *template_searchpath=None*, *user_defined_macros=None*, *user_defined_filters=None*, *default_args=None*, *concurrency=16*, *max_active_runs=16*, *dagrun_timeout=None*, *sla_miss_callback=None*, *default_view=u'tree'*, *orientation='LR'*, *catchup=True*, *on_success_callback=None*, *on_failure_callback=None*, *params=None*)
Bases: airflow.dag.base_dag.BaseDag, airflow.utils.log.logging_mixin.LoggingMixin

A dag (directed acyclic graph) is a collection of tasks with directional dependencies. A dag also has a schedule, a start end an end date (optional). For each schedule, (say daily or hourly), the DAG needs to run each individual tasks as their dependencies are met. Certain tasks have the property of depending on their own past, meaning that they can't run until their previous schedule (and upstream tasks) are completed.

DAGs essentially act as namespaces for tasks. A task_id can only be added once to a DAG.

Parameters

- **dag_id** (*string*) – The id of the DAG
- **description** (*string*) – The description for the DAG to e.g. be shown on the webserver
- **schedule_interval** (*datetime.timedelta or dateutil.relativedelta.relativedelta or str that acts as a cron expression*) – Defines how often that DAG runs, this timedelta object gets added to your latest task instance's execution_date to figure out the next schedule
- **start_date** (*datetime.datetime*) – The timestamp from which the scheduler will attempt to backfill
- **end_date** (*datetime.datetime*) – A date beyond which your DAG won't run, leave to None for open ended scheduling
- **template_searchpath** (*string or list of stings*) – This list of folders (non relative) defines where jinja will look for your templates. Order matters. Note that jinja/airflow includes the path of your DAG file by default
- **user_defined_macros** (*dict*) – a dictionary of macros that will be exposed in your jinja templates. For example, passing `dict(foo='bar')` to this argument allows you to `{{ foo }}` in all jinja templates related to this DAG. Note that you can pass any type of object here.
- **user_defined_filters** (*dict*) – a dictionary of filters that will be exposed in your jinja templates. For example, passing `dict(hello=lambda name: 'Hello %s' % name)` to this argument allows you to `{{ 'world' | hello }}` in all jinja templates related to this DAG.
- **default_args** (*dict*) – A dictionary of default parameters to be used as constructor keyword parameters when initialising operators. Note that operators have the same hook, and precede those defined here, meaning that if your dict contains *'depends_on_past': True* here and *'depends_on_past': False* in the operator's call *default_args*, the actual value will be *False*.
- **params** (*dict*) – a dictionary of DAG level parameters that are made accessible in templates, namespaced under *params*. These params can be overridden at the task level.
- **concurrency** (*int*) – the number of task instances allowed to run concurrently
- **max_active_runs** (*int*) – maximum number of active DAG runs, beyond this number of DAG runs in a running state, the scheduler won't create new active DAG runs
- **dagrun_timeout** (*datetime.timedelta*) – specify how long a DagRun should be up before timing out / failing, so that new DagRuns can be created
- **sla_miss_callback** (*types.FunctionType*) – specify a function to call when reporting SLA timeouts.
- **default_view** (*string*) – Specify DAG default view (tree, graph, duration, gantt, landing_times)
- **orientation** (*string*) – Specify DAG orientation in graph view (LR, TB, RL, BT)
- **catchup** (*bool*) – Perform scheduler catchup (or only run latest)? Defaults to True
- **on_failure_callback** (*callable*) – A function to be called when a DagRun of this dag fails. A context dictionary is passed as a single parameter to this function.

- **on_success_callback** (*callable*) – Much like the on_failure_callback except that it is executed when the dag succeeds.

add_task(*task*)
Add a task to the DAG

Parameters **task** (*task*) – the task you want to add

add_tasks(*tasks*)
Add a list of tasks to the DAG

Parameters **tasks** (*list of tasks*) – a lit of tasks you want to add

clear(***kwargs*)
Clears a set of task instances associated with the current dag for a specified date range.

cli()
Exposes a CLI specific to this DAG

concurrency_reached
Returns a boolean indicating whether the concurrency limit for this DAG has been reached

create_dagrun(***kwargs*)
Creates a dag run from this dag including the tasks associated with this dag. Returns the dag run.

Parameters

- **run_id** (*string*) – defines the the run id for this dag run
- **execution_date** (*datetime*) – the execution date of this dag run
- **state** (*State*) – the state of the dag run
- **start_date** (*datetime*) – the date this dag run should be evaluated
- **external_trigger** (*bool*) – whether this dag run is externally triggered
- **session** (*Session*) – database session

static deactivate_stale_dags(**args*, ***kwargs*)
Deactivate any DAGs that were last touched by the scheduler before the expiration date. These DAGs were likely deleted.

Parameters **expiration_date** (*datetime*) – set inactive DAGs that were touched before this time

Returns None

static deactivate_unknown_dags(**args*, ***kwargs*)
Given a list of known DAGs, deactivate any other DAGs that are marked as active in the ORM

Parameters **active_dag_ids** (*list[unicode]*) – list of DAG IDs that are active

Returns None

filepath
File location of where the dag object is instantiated

folder
Folder location of where the dag object is instantiated

following_schedule(*dttm*)
Calculates the following schedule for this dag in local time

Parameters **dttm** – utc datetime

Returns utc datetime

get_active_runs(***kwargs*)
Returns a list of dag run execution dates currently running

Parameters **session** –

Returns List of execution dates

get_dagrun(***kwargs*)
Returns the dag run for a given execution date if it exists, otherwise none.

Parameters

- **execution_date** – The execution date of the DagRun to find.
- **session** –

Returns The DagRun if found, otherwise None.

get_last_dagrun(***kwargs*)
Returns the last dag run for this dag, None if there was none. Last dag run can be any type of run eg. scheduled or backfilled. Overridden DagRuns are ignored

get_num_active_runs(***kwargs*)
Returns the number of active "running" dag runs

Parameters

- **external_trigger** (*bool*) – True for externally triggered active dag runs
- **session** –

Returns number greater than 0 for active dag runs

static get_num_task_instances(**args*, ***kwargs*)
Returns the number of task instances in the given DAG.

Parameters

- **session** – ORM session
- **dag_id** (*unicode*) – ID of the DAG to get the task concurrency of
- **task_ids** (*list[unicode]*) – A list of valid task IDs for the given DAG
- **states** (*list[state]*) – A list of states to filter by if supplied

Returns The number of running tasks

Return type int

get_run_dates(*start_date*, *end_date=None*)
Returns a list of dates between the interval received as parameter using this dag's schedule interval. Returned dates can be used for execution dates.

Parameters

- **start_date** (*datetime*) – the start date of the interval
- **end_date** (*datetime*) – the end date of the interval, defaults to timezone.utcnow()

Returns a list of dates within the interval following the dag's schedule

Return type list

get_template_env()
Returns a jinja2 Environment while taking into account the DAGs template_searchpath, user_defined_macros and user_defined_filters

handle_callback(***kwargs*)
Triggers the appropriate callback depending on the value of success, namely the on_failure_callback or on_success_callback. This method gets the context of a single TaskInstance part of this DagRun and passes that to the callable along with a 'reason', primarily to differentiate DagRun failures. .. note:

```
The logs end up in $AIRFLOW_HOME/logs/scheduler/latest/PROJECT/DAG_FILE.py.log
```

Parameters

- **dagrun** – DagRun object
- **success** – Flag to specify if failure or success callback should be called
- **reason** – Completion reason
- **session** – Database session

is_paused
Returns a boolean indicating whether this DAG is paused

latest_execution_date
Returns the latest date for which at least one dag run exists

normalize_schedule(*dttm*)
Returns dttm + interval unless dttm is first interval then it returns dttm

previous_schedule(*dttm*)
Calculates the previous schedule for this dag in local time

Parameters **dttm** – utc datetime

Returns utc datetime

run(*start_date=None*, *end_date=None*, *mark_success=False*, *local=False*, *executor=None*, *donot_pickle=False*, *ignore_task_deps=False*, *ignore_first_depends_on_past=False*, *pool=None*, *delay_on_limit_secs=1.0*, *verbose=False*, *conf=None*, *rerun_failed_tasks=False*)
Runs the DAG.

Parameters

- **start_date** (*datetime*) – the start date of the range to run
- **end_date** (*datetime*) – the end date of the range to run
- **mark_success** (*bool*) – True to mark jobs as succeeded without running them
- **local** (*bool*) – True to run the tasks using the LocalExecutor
- **executor** (*BaseExecutor*) – The executor instance to run the tasks
- **donot_pickle** (*bool*) – True to avoid pickling DAG object and send to workers
- **ignore_task_deps** (*bool*) – True to skip upstream tasks
- **ignore_first_depends_on_past** (*bool*) – True to ignore depends_on_past dependencies for the first set of tasks only
- **pool** (*string*) – Resource pool to use
- **delay_on_limit_secs** (*float*) – Time in seconds to wait before next attempt to run dag run when max_active_runs limit has been reached
- **verbose** (*boolean*) – Make logging output more verbose
- **conf** (*dict*) – user defined dictionary passed from CLI

set_dependency(*upstream_task_id*, *downstream_task_id*)
Simple utility method to set dependency between two tasks that already have been added to the DAG using add_task()

sub_dag(*task_regex*, *include_downstream=False*, *include_upstream=True*)
Returns a subset of the current dag as a deep copy of the current dag based on a regex that should match one or many tasks, and includes upstream and downstream neighbours based on the flag passed.

subdags
Returns a list of the subdag objects associated to this DAG

sync_to_db(***kwargs*)
Save attributes about this DAG to the DB. Note that this method can be called for both DAGs and SubDAGs. A SubDag is actually a SubDagOperator.

Parameters

- **dag** (DAG) – the DAG object to save to the DB
- **sync_time** (*datetime*) – The time that the DAG should be marked as sync'ed

Returns None

test_cycle()
Check to see if there are any cycles in the DAG. Returns False if no cycle found, otherwise raises exception.

topological_sort()
Sorts tasks in topographical order, such that a task comes after any of its upstream dependencies.

Heavily inspired by: http://blog.jupo.org/2012/04/06/topological-sorting-acyclic-directed-graphs/

Returns list of tasks in topological order

tree_view()
Shows an ascii tree representation of the DAG

class airflow.models.**DagBag**(*dag_folder=None*, *executor=None*, *include_examples=True*)
Bases: airflow.dag.base_dag.BaseDagBag, airflow.utils.log.logging_mixin.LoggingMixin

A dagbag is a collection of dags, parsed out of a folder tree and has high level configuration settings, like what database to use as a backend and what executor to use to fire off tasks. This makes it easier to run distinct environments for say production and development, tests, or for different teams or security profiles. What would have been system level settings are now dagbag level so that one system can run multiple, independent settings sets.

Parameters

- **dag_folder** (*unicode*) – the folder to scan to find DAGs
- **executor** – the executor to use when executing task instances in this DagBag
- **include_examples** (*bool*) – whether to include the examples that ship with airflow or not
- **has_logged** – an instance boolean that gets flipped from False to True after a file has been skipped. This is to prevent overloading the user with logging messages about skipped files. Therefore only once per DagBag is a file logged being skipped.

bag_dag(*dag*, *parent_dag*, *root_dag*)
Adds the DAG into the bag, recurses into sub dags. Throws AirflowDagCycleException if a cycle is detected in this dag or its subdags

collect_dags(*dag_folder=None*, *only_if_updated=True*)
Given a file path or a folder, this method looks for python modules, imports them and adds them to the dagbag collection.

Note that if a .airflowignore file is found while processing, the directory, it will behaves much like a .gitignore does, ignoring files that match any of the regex patterns specified in the file. **Note**: The patterns in .airflowignore are treated as un-anchored regexes, not shell-like glob patterns.

dagbag_report()
Prints a report around DagBag loading stats

get_dag(*dag_id*)
Gets the DAG out of the dictionary, and refreshes it if expired

kill_zombies(***kwargs*)
Fails tasks that haven't had a heartbeat in too long

process_file(*filepath*, *only_if_updated=True*, *safe_mode=True*)
Given a path to a python module or zip file, this method imports the module and look for dag objects within it.

size()

Returns the amount of dags contained in this dagbag

class airflow.models.**DagModel**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

class airflow.models.**DagPickle**(*dag*)
Bases: sqlalchemy.ext.declarative.api.Base

Dags can originate from different places (user repos, master repo, ...) and also get executed in different places (different executors). This object represents a version of a DAG and becomes a source of truth for a BackfillJob execution. A pickle is a native python serialized object, and in this case gets stored in the database for the duration of the job.

The executors pick up the DagPickle id and read the dag definition from the database.

class airflow.models.**DagRun**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base, airflow.utils.log.logging_mixin.LoggingMixin

DagRun describes an instance of a Dag. It can be created by the scheduler (for regular runs) or by an external trigger

static find(**args*, ***kwargs*)
Returns a set of dag runs for the given search criteria.

Parameters

- **dag_id** (*integer, list*) – the dag_id to find dag runs for
- **run_id** (*string*) – defines the the run id for this dag run
- **execution_date** (*datetime*) – the execution date
- **state** (*State*) – the state of the dag run
- **external_trigger** (*bool*) – whether this dag run is externally triggered
- **no_backfills** – return no backfills (True), return all (False).

Defaults to False :type no_backfills: bool :param session: database session :type session: Session

get_dag()
Returns the Dag associated with this DagRun.

Returns DAG

classmethod get_latest_runs(***kwargs*)
Returns the latest DagRun for each DAG.

get_previous_dagrun(***kwargs*)
The previous DagRun, if there is one

get_previous_scheduled_dagrun(***kwargs*)
The previous, SCHEDULED DagRun, if there is one

static get_run(*session*, *dag_id*, *execution_date*)

Parameters

- **dag_id** (*unicode*) – DAG ID
- **execution_date** (*datetime*) – execution date

Returns DagRun corresponding to the given dag_id and execution date

if one exists. None otherwise. :rtype: DagRun

get_task_instance(***kwargs*)
Returns the task instance specified by task_id for this dag run

Parameters **task_id** – the task id

get_task_instances(***kwargs*)
Returns the task instances for this dag run

refresh_from_db(***kwargs*)
Reloads the current dagrun from the database :param session: database session

update_state(***kwargs*)
Determines the overall state of the DagRun based on the state of its TaskInstances.

Returns State

verify_integrity(***kwargs*)
Verifies the DagRun by checking for removed tasks or tasks that are not in the database yet. It will set state to removed or add the task if required.

class airflow.models.**DagStat**(*dag_id*, *state*, *count=0*, *dirty=False*)
Bases: sqlalchemy.ext.declarative.api.Base

static create(**args*, ***kwargs*)
Creates the missing states the stats table for the dag specified

Parameters

- **dag_id** – dag id of the dag to create stats for
- **session** – database session

Returns

static set_dirty(**args*, ***kwargs*)

Parameters

- **dag_id** – the dag_id to mark dirty
- **session** – database session

Returns

static update(**args*, ***kwargs*)
Updates the stats for dirty/out-of-sync dags

Parameters

- **dag_ids** (*list*) – dag_ids to be updated
- **dirty_only** (*bool*) – only updated for marked dirty, defaults to True
- **session** (*Session*) – db session to use

class airflow.models.**ImportError**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

exception airflow.models.**InvalidFernetToken**
Bases: exceptions.Exception

class airflow.models.**KnownEvent**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

class airflow.models.**KnownEventType**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

class airflow.models.**KubeResourceVersion**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

class airflow.models.**KubeWorkerIdentifier**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

class airflow.models.**Log**(*event*, *task_instance*, *owner=None*, *extra=None*, ***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Used to actively log events to the database

class airflow.models.**NullFernet**
Bases: future.types.newobject.newobject

A "Null" encryptor class that doesn't encrypt or decrypt but that presents a similar interface to Fernet.

The purpose of this is to make the rest of the code not have to know the difference, and to only display the message once, not 20 times when *airflow initdb* is ran.

class airflow.models.**Pool**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

open_slots(***kwargs*)
Returns the number of slots open at the moment

queued_slots(***kwargs*)
Returns the number of slots used at the moment

used_slots(***kwargs*)
Returns the number of slots used at the moment

class airflow.models.**SlaMiss**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

Model that stores a history of the SLA that have been missed. It is used to keep track of SLA failures over time and to avoid double triggering alert emails.

class airflow.models.**TaskFail**(*task*, *execution_date*, *start_date*, *end_date*)
Bases: sqlalchemy.ext.declarative.api.Base

TaskFail tracks the failed run durations of each task instance.

class airflow.models.**TaskInstance**(*task*, *execution_date*, *state=None*)

Bases: sqlalchemy.ext.declarative.api.Base, airflow.utils.log.logging_mixin.LoggingMixin

Task instances store the state of a task instance. This table is the authority and single source of truth around what tasks have run and the state they are in.

The SqlAlchemy model doesn't have a SqlAlchemy foreign key to the task or dag model deliberately to have more control over transactions.

Database transactions on this table should insure double triggers and any confusion around what task instances are or aren't ready to run even while multiple schedulers may be firing task instances.

are_dependencies_met(***kwargs*)

Returns whether or not all the conditions are met for this task instance to be run given the context for the dependencies (e.g. a task instance being force run from the UI will ignore some dependencies).

Parameters

- **dep_context** (*DepContext*) – The execution context that determines the dependencies that should be evaluated.
- **session** (*Session*) – database session
- **verbose** (*boolean*) – whether log details on failed dependencies on info or debug log level

are_dependents_done(***kwargs*)

Checks whether the dependents of this task instance have all succeeded. This is meant to be used by wait_for_downstream.

This is useful when you do not want to start processing the next schedule of a task until the dependents are done. For instance, if the task DROPs and recreates a table.

clear_xcom_data(***kwargs*)

Clears all XCom data from the database for the task instance

command(*mark_success=False*, *ignore_all_deps=False*, *ignore_depends_on_past=False*, *ignore_task_deps=False*, *ignore_ti_state=False*, *local=False*, *pickle_id=None*, *raw=False*, *job_id=None*, *pool=None*, *cfg_path=None*)

Returns a command that can be executed anywhere where airflow is installed. This command is part of the message sent to executors by the orchestrator.

command_as_list(*mark_success=False*, *ignore_all_deps=False*, *ignore_task_deps=False*, *ignore_depends_on_past=False*, *ignore_ti_state=False*, *local=False*, *pickle_id=None*, *raw=False*, *job_id=None*, *pool=None*, *cfg_path=None*)

Returns a command that can be executed anywhere where airflow is installed. This command is part of the message sent to executors by the orchestrator.

current_state(***kwargs*)

Get the very latest state from the database, if a session is passed, we use and looking up the state becomes part of the session, otherwise a new session is used.

error(***kwargs*)

Forces the task instance's state to FAILED in the database.

static generate_command(*dag_id*, *task_id*, *execution_date*, *mark_success=False*, *ignore_all_deps=False*, *ignore_depends_on_past=False*, *ignore_task_deps=False*, *ignore_ti_state=False*, *local=False*, *pickle_id=None*, *file_path=None*, *raw=False*, *job_id=None*, *pool=None*, *cfg_path=None*)

Generates the shell command required to execute this task instance.

Parameters

- **dag_id** (*unicode*) – DAG ID
- **task_id** (*unicode*) – Task ID
- **execution_date** (*datetime*) – Execution date for the task
- **mark_success** (*bool*) – Whether to mark the task as successful
- **ignore_all_deps** (*boolean*) – Ignore all ignorable dependencies. Overrides the other ignore_* parameters.
- **ignore_depends_on_past** (*boolean*) – Ignore depends_on_past parameter of DAGs (e.g. for Backfills)
- **ignore_task_deps** (*boolean*) – Ignore task-specific dependencies such as depends_on_past and trigger rule
- **ignore_ti_state** (*boolean*) – Ignore the task instance's previous failure/success
- **local** (*bool*) – Whether to run the task locally
- **pickle_id** (*unicode*) – If the DAG was serialized to the DB, the ID associated with the pickled DAG
- **file_path** – path to the file containing the DAG definition
- **raw** – raw mode (needs more details)
- **job_id** – job ID (needs more details)
- **pool** (*unicode*) – the Airflow pool that the task should run in
- **cfg_path** (*basestring*) – the Path to the configuration file

Returns shell command that can be used to run the task instance

get_dagrun(***kwargs*)
Returns the DagRun for this TaskInstance

Parameters **session** –

Returns DagRun

init_on_load()
Initialize the attributes that aren't stored in the DB.

init_run_context(*raw=False*)
Sets the log context.

is_eligible_to_retry()
Is task instance is eligible for retry

is_premature
Returns whether a task is in UP_FOR_RETRY state and its retry interval has elapsed.

key
Returns a tuple that identifies the task instance uniquely

next_retry_datetime()
Get datetime of the next retry if the task instance fails. For exponential backoff, retry_delay is used as base and will be converted to seconds.

pool_full(***kwargs*)
Returns a boolean as to whether the slot pool has room for this task to run

previous_ti
The task instance for the task that ran before this task instance

ready_for_retry()
Checks on whether the task instance is in the right state and timeframe to be retried.

refresh_from_db(***kwargs*)
Refreshes the task instance from the database based on the primary key

Parameters **lock_for_update** – if True, indicates that the database should lock the TaskInstance (issuing a FOR UPDATE clause) until the session is committed.

try_number
Return the try number that this task number will be when it is acutally run.

If the TI is currently running, this will match the column in the databse, in all othercases this will be incremenetd

xcom_pull(*task_ids=None*, *dag_id=None*, *key=u'return_value'*, *include_prior_dates=False*)
Pull XComs that optionally meet certain criteria.

The default value for *key* limits the search to XComs that were returned by other tasks (as opposed to those that were pushed manually). To remove this filter, pass key=None (or any desired value).

If a single task_id string is provided, the result is the value of the most recent matching XCom from that task_id. If multiple task_ids are provided, a tuple of matching values is returned. None is returned whenever no matches are found.

Parameters

- **key** (*string*) – A key for the XCom. If provided, only XComs with matching keys will be returned. The default key is 'return_value', also available as a constant XCOM_RETURN_KEY. This key is automatically given to XComs returned by tasks (as opposed to being pushed manually). To remove the filter, pass key=None.
- **task_ids** (*string or iterable of strings (representing task_ids)*) – Only XComs from tasks with matching ids will be pulled. Can pass None to remove the filter.
- **dag_id** (*string*) – If provided, only pulls XComs from this DAG. If None (default), the DAG of the calling task is used.
- **include_prior_dates** (*bool*) – If False, only XComs from the current execution_date are returned. If True, XComs from previous dates are returned as well.

xcom_push(*key*, *value*, *execution_date=None*)
Make an XCom available for tasks to pull.

Parameters

- **key** (*string*) – A key for the XCom
- **value** (*any pickleable object*) – A value for the XCom. The value is pickled and stored in the database.
- **execution_date** (*datetime*) – if provided, the XCom will not be visible until this date. This can be used, for example, to send a message to a task on a future date without it being immediately visible.

class airflow.models.**User**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

class airflow.models.**Variable**(***kwargs*)

Bases: sqlalchemy.ext.declarative.api.Base, airflow.utils.log.logging_mixin.LoggingMixin

classmethod setdefault(*key*, *default*, *deserialize_json=False*)

Like a Python builtin dict object, setdefault returns the current value for a key, and if it isn't there, stores the default value and returns it.

Parameters

- **key** (*String*) – Dict key for this Variable
- **default** – Default value to set and return if the variable

isn't already in the DB :type default: Mixed :param deserialize_json: Store this as a JSON encoded value in the DB

and un-encode it when retrieving a value

Returns Mixed

class airflow.models.**XCom**(***kwargs*)

Bases: sqlalchemy.ext.declarative.api.Base, airflow.utils.log.logging_mixin.LoggingMixin

Base class for XCom objects.

classmethod get_many(***kwargs*)

Retrieve an XCom value, optionally meeting certain criteria TODO: "pickling" has been deprecated and JSON is preferred.

"pickling" will be removed in Airflow 2.0.

classmethod get_one(***kwargs*)

Retrieve an XCom value, optionally meeting certain criteria. TODO: "pickling" has been deprecated and JSON is preferred.

"pickling" will be removed in Airflow 2.0.

Returns XCom value

classmethod set(***kwargs*)

Store an XCom value. TODO: "pickling" has been deprecated and JSON is preferred.

"pickling" will be removed in Airflow 2.0.

Returns None

airflow.models.**clear_task_instances**(*tis*, *session*, *activate_dag_runs=True*, *dag=None*)

Clears a set of task instances, but makes sure the running ones get killed.

Parameters

- **tis** – a list of task instances
- **session** – current session
- **activate_dag_runs** – flag to check for active dag run
- **dag** – DAG object

airflow.models.**get_fernet**()
Deferred load of Fernet key.

This function could fail either because Cryptography is not installed or because the Fernet key is invalid.

Returns Fernet object

Raises AirflowException if there's a problem trying to load Fernet

3.19.4 Hooks

Hooks are interfaces to external platforms and databases, implementing a common interface when possible and acting as building blocks for operators.

class airflow.hooks.dbapi_hook.**DbApiHook**(**args*, ***kwargs*)
Bases: airflow.hooks.base_hook.BaseHook

Abstract base class for sql hooks.

bulk_dump(*table*, *tmp_file*)
Dumps a database table into a tab-delimited file

Parameters

- **table** (*str*) – The name of the source table
- **tmp_file** (*str*) – The path of the target file

bulk_load(*table*, *tmp_file*)
Loads a tab-delimited file into a database table

Parameters

- **table** (*str*) – The name of the target table
- **tmp_file** (*str*) – The path of the file to load into the table

get_autocommit(*conn*)
Get autocommit setting for the provided connection. Return True if conn.autocommit is set to True. Return False if conn.autocommit is not set or set to False or conn does not support autocommit. :param conn: Connection to get autocommit setting from. :type conn: connection object. :return: connection autocommit setting. :rtype bool.

get_conn()
Returns a connection object

get_cursor()
Returns a cursor

get_first(*sql*, *parameters=None*)
Executes the sql and returns the first resulting row.

Parameters

- **sql** (*str or list*) – the sql statement to be executed (str) or a list of sql statements to execute
- **parameters** (*mapping or iterable*) – The parameters to render the SQL query with.

get_pandas_df(*sql*, *parameters=None*)
Executes the sql and returns a pandas dataframe

Parameters

- **sql** (*str or list*) – the sql statement to be executed (str) or a list of sql statements to execute
- **parameters** (*mapping or iterable*) – The parameters to render the SQL query with.

get_records(*sql*, *parameters=None*)
Executes the sql and returns a set of records.

Parameters

- **sql** (*str or list*) – the sql statement to be executed (str) or a list of sql statements to execute
- **parameters** (*mapping or iterable*) – The parameters to render the SQL query with.

insert_rows(*table*, *rows*, *target_fields=None*, *commit_every=1000*, *replace=False*)
A generic way to insert a set of tuples into a table, a new transaction is created every commit_every rows

Parameters

- **table** (*str*) – Name of the target table
- **rows** (*iterable of tuples*) – The rows to insert into the table
- **target_fields** (*iterable of strings*) – The names of the columns to fill in the table
- **commit_every** (*int*) – The maximum number of rows to insert in one transaction. Set to 0 to insert all rows in one transaction.
- **replace** (*bool*) – Whether to replace instead of insert

run(*sql*, *autocommit=False*, *parameters=None*)
Runs a command or a list of commands. Pass a list of sql statements to the sql parameter to get them to execute sequentially

Parameters

- **sql** (*str or list*) – the sql statement to be executed (str) or a list of sql statements to execute
- **autocommit** (*bool*) – What to set the connection's autocommit setting to before executing the query.
- **parameters** (*mapping or iterable*) – The parameters to render the SQL query with.

set_autocommit(*conn*, *autocommit*)
Sets the autocommit flag on the connection

class airflow.hooks.docker_hook.**DockerHook**(*docker_conn_id='docker_default'*, *base_url=None*, *version=None*, *tls=None*)
Bases: airflow.hooks.base_hook.BaseHook, airflow.utils.log.logging_mixin.LoggingMixin

Interact with a private Docker registry.

Parameters **docker_conn_id** (*str*) – ID of the Airflow connection where credentials and extra configuration are stored

class airflow.hooks.http_hook.**HttpHook**(*method='POST'*, *http_conn_id='http_default'*)
Bases: airflow.hooks.base_hook.BaseHook

Interact with HTTP servers. :param http_conn_id: connection that has the base API url i.e https://www.google.com/

and optional authentication credentials. Default headers can also be specified in the Extra field in json format.

Parameters **method** (*str*) – the API method to be called

check_response(*response*)
Checks the status code and raise an AirflowException exception on non 2XX or 3XX status codes :param response: A requests response object :type response: requests.response

get_conn(*headers=None*)
Returns http session for use with requests :param headers: additional headers to be passed through as a dictionary :type headers: dict

run(*endpoint*, *data=None*, *headers=None*, *extra_options=None*)
Performs the request :param endpoint: the endpoint to be called i.e. resource/v1/query? :type endpoint: str :param data: payload to be uploaded or request parameters :type data: dict :param headers: additional headers to be passed through as a dictionary :type headers: dict :param extra_options: additional options to be used when executing the request

i.e. {'check_response': False} to avoid checking raising exceptions on non 2XX or 3XX status codes

run_and_check(*session*, *prepped_request*, *extra_options*)
Grabs extra options like timeout and actually runs the request, checking for the result :param session: the session to be used to execute the request :type session: requests.Session :param prepped_request: the prepared request generated in run() :type prepped_request: session.prepare_request :param extra_options: additional options to be used when executing the request

i.e. {'check_response': False} to avoid checking raising exceptions on non 2XX or 3XX status codes

run_with_advanced_retry(*_retry_args*, **args*, ***kwargs*)
Runs Hook.run() with a Tenacity decorator attached to it. This is useful for connectors which might be disturbed by intermittent issues and should not instantly fail. :param _retry_args: Arguments which define the retry behaviour.

See Tenacity documentation at https://github.com/jd/tenacity

Example: :: hook = HttpHook(http_conn_id='my_conn',method='GET') retry_args = dict(

wait=tenacity.wait_exponential(), stop=tenacity.stop_after_attempt(10), retry=requests.exceptions.ConnectionError

) hook.run_with_advanced_retry(

endpoint='v1/test', _retry_args=retry_args

)

class airflow.hooks.mssql_hook.**MsSqlHook**(**args*, ***kwargs*)
Bases: *airflow.hooks.dbapi_hook.DbApiHook*

Interact with Microsoft SQL Server.

get_conn()
Returns a mssql connection object

set_autocommit(*conn*, *autocommit*)
Sets the autocommit flag on the connection

class airflow.hooks.pig_hook.**PigCliHook**(*pig_cli_conn_id='pig_cli_default'*)
Bases: airflow.hooks.base_hook.BaseHook

Simple wrapper around the pig CLI.

Note that you can also set default pig CLI properties using the pig_properties to be used in your connection as in {"pig_properties": "-Dpig.tmpfilecompression=true"}

run_cli(*pig*, *verbose=True*)
Run an pig script using the pig cli

```
>>> ph = PigCliHook()
>>> result = ph.run_cli("ls /;")
>>> ("hdfs://" in result)
True
```

class airflow.hooks.S3_hook.**S3Hook**(*aws_conn_id='aws_default'*)
Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS S3, using the boto3 library.

check_for_bucket(*bucket_name*)
Check if bucket_name exists.

Parameters **bucket_name** (*str*) – the name of the bucket

check_for_key(*key*, *bucket_name=None*)
Checks if a key exists in a bucket

Parameters

- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which the file is stored

check_for_prefix(*bucket_name*, *prefix*, *delimiter*)
Checks that a prefix exists in a bucket

check_for_wildcard_key(*wildcard_key*, *bucket_name=None*, *delimiter=''*)
Checks that a key matching a wildcard expression exists in a bucket

get_bucket(*bucket_name*)
Returns a boto3.S3.Bucket object

Parameters **bucket_name** (*str*) – the name of the bucket

get_key(*key*, *bucket_name=None*)
Returns a boto3.s3.Object

Parameters

- **key** (*str*) – the path to the key
- **bucket_name** (*str*) – the name of the bucket

get_wildcard_key(*wildcard_key*, *bucket_name=None*, *delimiter=''*)
Returns a boto3.s3.Object object matching the wildcard expression

Parameters

- **wildcard_key** (*str*) – the path to the key
- **bucket_name** (*str*) – the name of the bucket

list_keys(*bucket_name*, *prefix=''*, *delimiter=''*, *page_size=None*, *max_items=None*)
Lists keys in a bucket under prefix and not containing delimiter

Parameters

- **bucket_name** (*str*) – the name of the bucket
- **prefix** (*str*) – a key prefix
- **delimiter** (*str*) – the delimiter marks key hierarchy.
- **page_size** (*int*) – pagination size
- **max_items** (*int*) – maximum items to return

list_prefixes(*bucket_name*, *prefix=''*, *delimiter=''*, *page_size=None*, *max_items=None*)
Lists prefixes in a bucket under prefix

Parameters

- **bucket_name** (*str*) – the name of the bucket
- **prefix** (*str*) – a key prefix
- **delimiter** (*str*) – the delimiter marks key hierarchy.
- **page_size** (*int*) – pagination size
- **max_items** (*int*) – maximum items to return

load_bytes(*bytes_data*, *key*, *bucket_name=None*, *replace=False*, *encrypt=False*)
Loads bytes to S3

This is provided as a convenience to drop a string in S3. It uses the boto infrastructure to ship a file to s3.

Parameters

- **bytes_data** (*bytes*) – bytes to set as content for the key.
- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which to store the file
- **replace** (*bool*) – A flag to decide whether or not to overwrite the key if it already exists
- **encrypt** (*bool*) – If True, the file will be encrypted on the server-side by S3 and will be stored in an encrypted form while at rest in S3.

load_file(*filename*, *key*, *bucket_name=None*, *replace=False*, *encrypt=False*)
Loads a local file to S3

Parameters

- **filename** (*str*) – name of the file to load.
- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which to store the file

- **replace** (*bool*) – A flag to decide whether or not to overwrite the key if it already exists. If replace is False and the key exists, an error will be raised.
- **encrypt** (*bool*) – If True, the file will be encrypted on the server-side by S3 and will be stored in an encrypted form while at rest in S3.

load_string(*string_data*, *key*, *bucket_name=None*, *replace=False*, *encrypt=False*, *encoding='utf-8'*)

Loads a string to S3

This is provided as a convenience to drop a string in S3. It uses the boto infrastructure to ship a file to s3.

Parameters

- **string_data** (*str*) – string to set as content for the key.
- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which to store the file
- **replace** (*bool*) – A flag to decide whether or not to overwrite the key if it already exists
- **encrypt** (*bool*) – If True, the file will be encrypted on the server-side by S3 and will be stored in an encrypted form while at rest in S3.

read_key(*key*, *bucket_name=None*)

Reads a key from S3

Parameters

- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which the file is stored

select_key(*key*, *bucket_name=None*, *expression='SELECT * FROM S3Object'*, *expression_type='SQL'*, *input_serialization={'CSV': {}}*, *output_serialization={'CSV': {}}*)

Reads a key with S3 Select.

Parameters

- **key** (*str*) – S3 key that will point to the file
- **bucket_name** (*str*) – Name of the bucket in which the file is stored
- **expression** (*str*) – S3 Select expression
- **expression_type** (*str*) – S3 Select expression type
- **input_serialization** (*dict*) – S3 Select input data serialization format
- **output_serialization** (*dict*) – S3 Select output data serialization format

Returns retrieved subset of original data by S3 Select

Return type str

See also:

For more details about S3 Select parameters: http://boto3.readthedocs.io/en/latest/reference/services/s3.html#S3.Client.select_object_content

class airflow.hooks.sqlite_hook.**SqliteHook**(**args*, ***kwargs*)

Bases: *airflow.hooks.dbapi_hook.DbApiHook*

Interact with SQLite.

get_conn()
Returns a sqlite connection object

3.19.4.1 Community contributed hooks

class airflow.contrib.hooks.aws_dynamodb_hook.**AwsDynamoDBHook**(*table_keys=None*, *table_name=None*, *region_name=None*, **args*, ***kwargs*)

Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS DynamoDB.

Parameters

- **table_keys** (*list*) – partition key and sort key
- **table_name** (*str*) – target DynamoDB table
- **region_name** (*str*) – aws region name (example: us-east-1)

write_batch_data(*items*)
Write batch items to dynamodb table with provisioned throughout capacity.

class airflow.contrib.hooks.aws_hook.**AwsHook**(*aws_conn_id='aws_default'*)

Bases: airflow.hooks.base_hook.BaseHook

Interact with AWS. This class is a thin wrapper around the boto3 python library.

get_credentials(*region_name=None*)
Get the underlying *botocore.Credentials* object.

This contains the attributes: access_key, secret_key and token.

get_session(*region_name=None*)
Get the underlying boto3.session.

class airflow.contrib.hooks.aws_lambda_hook.**AwsLambdaHook**(*function_name*, *region_name=None*, *log_type='None'*, *qualifier='$LATEST'*, *invocation_type='RequestResponse'*, **args*, ***kwargs*)

Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS Lambda

Parameters

- **function_name** (*str*) – AWS Lambda Function Name
- **region_name** (*str*) – AWS Region Name (example: us-west-2)
- **log_type** (*str*) – Tail Invocation Request
- **qualifier** (*str*) – AWS Lambda Function Version or Alias Name
- **invocation_type** (*str*) – AWS Lambda Invocation Type (RequestResponse, Event etc)

invoke_lambda(*payload*)
Invoke Lambda Function

class airflow.contrib.hooks.bigquery_hook.**BigQueryHook**(*bigquery_conn_id='bigquery_default'*, *delegate_to=None*, *use_legacy_sql=True*)
Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*, *airflow.hooks.dbapi_hook.DbApiHook*, airflow.utils.log.logging_mixin.LoggingMixin

Interact with BigQuery. This hook uses the Google Cloud Platform connection.

get_conn()
Returns a BigQuery PEP 249 connection object.

get_pandas_df(*sql*, *parameters=None*, *dialect=None*)
Returns a Pandas DataFrame for the results produced by a BigQuery query. The DbApiHook method must be overridden because Pandas doesn't support PEP 249 connections, except for SQLite. See:

https://github.com/pydata/pandas/blob/master/pandas/io/sql.py#L447 https://github.com/pydata/pandas/issues/6900

Parameters

- **sql** (*string*) – The BigQuery SQL to execute.
- **parameters** (*mapping or iterable*) – The parameters to render the SQL query with (not used, leave to override superclass method)
- **dialect** (*string in {'legacy', 'standard'}*) – Dialect of BigQuery SQL – legacy SQL or standard SQL defaults to use *self.use_legacy_sql* if not specified

get_service()
Returns a BigQuery service object.

insert_rows(*table*, *rows*, *target_fields=None*, *commit_every=1000*)
Insertion is currently unsupported. Theoretically, you could use BigQuery's streaming API to insert rows into a table, but this hasn't been implemented.

table_exists(*project_id*, *dataset_id*, *table_id*)
Checks for the existence of a table in Google BigQuery.

Parameters

- **project_id** (*string*) – The Google cloud project in which to look for the table. The connection supplied to the hook must provide access to the specified project.
- **dataset_id** (*string*) – The name of the dataset in which to look for the table.
- **table_id** (*string*) – The name of the table to check the existence of.

class airflow.contrib.hooks.databricks_hook.**DatabricksHook**(*databricks_conn_id='databricks_default'*, *timeout_seconds=180*, *retry_limit=3*)
Bases: airflow.hooks.base_hook.BaseHook, airflow.utils.log.logging_mixin.LoggingMixin

Interact with Databricks.

submit_run(*json*)
Utility function to call the api/2.0/jobs/runs/submit endpoint.

Parameters **json** (*dict*) – The data used in the body of the request to the submit endpoint.

Returns the run_id as a string

Return type string

class airflow.contrib.hooks.datastore_hook.**DatastoreHook**(*datastore_conn_id='google_cloud_datastore_def*
delegate_to=None)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Interact with Google Cloud Datastore. This hook uses the Google Cloud Platform connection.

This object is not threads safe. If you want to make multiple requests simultaneously, you will need to create a hook per thread.

allocate_ids(*partialKeys*)

Allocate IDs for incomplete keys. see https://cloud.google.com/datastore/docs/reference/rest/v1/projects/allocateIds

Parameters **partialKeys** – a list of partial keys

Returns a list of full keys.

begin_transaction()

Get a new transaction handle

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/beginTransaction

Returns a transaction handle

commit(*body*)

Commit a transaction, optionally creating, deleting or modifying some entities.

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/commit

Parameters **body** – the body of the commit request

Returns the response body of the commit request

delete_operation(*name*)

Deletes the long-running operation

Parameters **name** – the name of the operation resource

export_to_storage_bucket(*bucket*, *namespace=None*, *entity_filter=None*, *labels=None*)

Export entities from Cloud Datastore to Cloud Storage for backup

get_conn(*version='v1'*)

Returns a Google Cloud Storage service object.

get_operation(*name*)

Gets the latest state of a long-running operation

Parameters **name** – the name of the operation resource

import_from_storage_bucket(*bucket*, *file*, *namespace=None*, *entity_filter=None*, *labels=None*)

Import a backup from Cloud Storage to Cloud Datastore

lookup(*keys*, *read_consistency=None*, *transaction=None*)

Lookup some entities by key

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/lookup

Parameters

- **keys** – the keys to lookup
- **read_consistency** – the read consistency to use. default, strong or eventual. Cannot be used with a transaction.
- **transaction** – the transaction to use, if any.

Returns the response body of the lookup request.

poll_operation_until_done(*name*, *polling_interval_in_seconds*)
Poll backup operation state until it's completed

rollback(*transaction*)
Roll back a transaction

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/rollback

Parameters **transaction** – the transaction to roll back

run_query(*body*)
Run a query for entities.

See also:

https://cloud.google.com/datastore/docs/reference/rest/v1/projects/runQuery

Parameters **body** – the body of the query request

Returns the batch of query results.

class airflow.contrib.hooks.discord_webhook_hook.**DiscordWebhookHook**(*http_conn_id=None*, *webhook_endpoint=None*, *message=''*, *username=None*, *avatar_url=None*, *tts=False*, *proxy=None*, **args*, ***kwargs*)

Bases: *airflow.hooks.http_hook.HttpHook*

This hook allows you to post messages to Discord using incoming webhooks. Takes a Discord connection ID with a default relative webhook endpoint. The default endpoint can be overridden using the webhook_endpoint parameter (https://discordapp.com/developers/docs/resources/webhook).

Each Discord webhook can be pre-configured to use a specific username and avatar_url. You can override these defaults in this hook.

Parameters

- **http_conn_id** (*str*) – Http connection ID with host as "https://discord.com/api/" and default webhook endpoint in the extra field in the form of {"webhook_endpoint": "webhooks/{webhook.id}/{webhook.token}"}

- **webhook_endpoint** (*str*) – Discord webhook endpoint in the form of "webhooks/{webhook.id}/{webhook.token}"
- **message** (*str*) – The message you want to send to your Discord channel (max 2000 characters)
- **username** (*str*) – Override the default username of the webhook
- **avatar_url** (*str*) – Override the default avatar of the webhook
- **tts** (*bool*) – Is a text-to-speech message
- **proxy** (*str*) – Proxy to use to make the Discord webhook call

execute()
Execute the Discord webhook call

class airflow.contrib.hooks.emr_hook.**EmrHook**(*emr_conn_id=None*, **args*, ***kwargs*)
Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS EMR. emr_conn_id is only neccessary for using the create_job_flow method.

create_job_flow(*job_flow_overrides*)
Creates a job flow using the config from the EMR connection. Keys of the json extra hash may have the arguments of the boto3 run_job_flow method. Overrides for this config may be passed as the job_flow_overrides.

class airflow.contrib.hooks.fs_hook.**FSHook**(*conn_id='fs_default'*)
Bases: airflow.hooks.base_hook.BaseHook

Allows for interaction with an file server.

Connection should have a name and a path specified under extra:

example: Conn Id: fs_test Conn Type: File (path) Host, Shchema, Login, Password, Port: empty Extra: {"path": "/tmp"}

class airflow.contrib.hooks.ftp_hook.**FTPHook**(*ftp_conn_id='ftp_default'*)
Bases: airflow.hooks.base_hook.BaseHook, airflow.utils.log.logging_mixin.LoggingMixin

Interact with FTP.

Errors that may occur throughout but should be handled downstream.

close_conn()
Closes the connection. An error will occur if the connection wasn't ever opened.

create_directory(*path*)
Creates a directory on the remote system.

Parameters **path** (*str*) – full path to the remote directory to create

delete_directory(*path*)
Deletes a directory on the remote system.

Parameters **path** (*str*) – full path to the remote directory to delete

delete_file(*path*)
Removes a file on the FTP Server.

Parameters **path** (*str*) – full path to the remote file

describe_directory(*path*)
Returns a dictionary of {filename: {attributes}} for all files on the remote system (where the MLSD command is supported).

Parameters **path** (*str*) – full path to the remote directory

get_conn()
Returns a FTP connection object

list_directory(*path*, *nlst=False*)
Returns a list of files on the remote system.

Parameters **path** (*str*) – full path to the remote directory to list

rename(*from_name*, *to_name*)
Rename a file.

Parameters

- **from_name** – rename file from name
- **to_name** – rename file to name

retrieve_file(*remote_full_path*, *local_full_path_or_buffer*)
Transfers the remote file to a local location.

If local_full_path_or_buffer is a string path, the file will be put at that location; if it is a file-like buffer, the file will be written to the buffer but not closed.

Parameters

- **remote_full_path** (*str*) – full path to the remote file
- **local_full_path_or_buffer** (*str or file-like buffer*) – full path to the local file or a file-like buffer

store_file(*remote_full_path*, *local_full_path_or_buffer*)
Transfers a local file to the remote location.

If local_full_path_or_buffer is a string path, the file will be read from that location; if it is a file-like buffer, the file will be read from the buffer but not closed.

Parameters

- **remote_full_path** (*str*) – full path to the remote file
- **local_full_path_or_buffer** (*str or file-like buffer*) – full path to the local file or a file-like buffer

class airflow.contrib.hooks.ftp_hook.**FTPSHook**(*ftp_conn_id='ftp_default'*)
Bases: *airflow.contrib.hooks.ftp_hook.FTPHook*

get_conn()
Returns a FTPS connection object.

class airflow.contrib.hooks.gcp_api_base_hook.**GoogleCloudBaseHook**(*gcp_conn_id='google_cloud_defaul*
dele-
gate_to=None)
Bases: airflow.hooks.base_hook.BaseHook, airflow.utils.log.logging_mixin.LoggingMixin

A base hook for Google cloud-related hooks. Google cloud has a shared REST API client that is built in the same way no matter which service you use. This class helps construct and authorize the credentials needed to then call apiclient.discovery.build() to actually discover and build a client for a Google cloud service.

The class also contains some miscellaneous helper functions.

All hook derived from this base hook use the 'Google Cloud Platform' connection type. Two ways of authentication are supported:

Default credentials: Only the 'Project Id' is required. You'll need to have set up default credentials, such as by the GOOGLE_APPLICATION_DEFAULT environment variable or from the metadata server on Google Compute Engine.

JSON key file: Specify 'Project Id', 'Key Path' and 'Scope'.

Legacy P12 key files are not supported.

class airflow.contrib.hooks.gcp_container_hook.**GKEClusterHook**(*project_id*, *location*)

Bases: airflow.hooks.base_hook.BaseHook

create_cluster(*cluster*, *retry=<object object>*, *timeout=<object object>*)

Creates a cluster, consisting of the specified number and type of Google Compute Engine instances.

Parameters

- **cluster** (*dict or google.cloud.container_v1.types.Cluster*) – A Cluster protobuf or dict. If dict is provided, it must be of the same form as the protobuf message google.cloud.container_v1.types.Cluster
- **retry** (*google.api_core.retry.Retry*) – A retry object (google.api_core.retry.Retry) used to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.

Returns The full url to the new, or existing, cluster

:raises ParseError: On JSON parsing problems when trying to convert dict AirflowException: cluster is not dict type nor Cluster proto type

delete_cluster(*name*, *retry=<object object>*, *timeout=<object object>*)

Deletes the cluster, including the Kubernetes endpoint and all worker nodes. Firewalls and routes that were configured during cluster creation are also deleted. Other Google Compute Engine resources that might be in use by the cluster (e.g. load balancer resources) will not be deleted if they weren't present at the initial create time.

Parameters

- **name** (*str*) – The name of the cluster to delete
- **retry** (*google.api_core.retry.Retry*) – Retry object used to determine when/if to retry requests. If None is specified, requests will not be retried.
- **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.

Returns The full url to the delete operation if successful, else None

get_cluster(*name*, *retry=<object object>*, *timeout=<object object>*)

Gets details of specified cluster :param name: The name of the cluster to retrieve :type name: str :param retry: A retry object used to retry requests. If None is specified,

requests will not be retried.

Parameters **timeout** (*float*) – The amount of time, in seconds, to wait for the request to complete. Note that if retry is specified, the timeout applies to each individual attempt.

Returns A google.cloud.container_v1.types.Cluster instance

get_operation(*operation_name*)
Fetches the operation from Google Cloud :param operation_name: Name of operation to fetch :type operation_name: str :return: The new, updated operation from Google Cloud

wait_for_operation(*operation*)
Given an operation, continuously fetches the status from Google Cloud until either completion or an error occurring :param operation: The Operation to wait for :type operation: A google.cloud.container_V1.gapic.enums.Operator :return: A new, updated operation fetched from Google Cloud

class airflow.contrib.hooks.gcp_dataflow_hook.**DataFlowHook**(*gcp_conn_id='google_cloud_default'*, *delegate_to=None*, *poll_sleep=10*)
Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

get_conn()
Returns a Google Cloud Storage service object.

class airflow.contrib.hooks.gcp_dataproc_hook.**DataProcHook**(*gcp_conn_id='google_cloud_default'*, *delegate_to=None*, *api_version='v1beta2'*)
Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Hook for Google Cloud Dataproc APIs.

await(*operation*)
Awaits for Google Cloud Dataproc Operation to complete.

get_conn()
Returns a Google Cloud Dataproc service object.

class airflow.contrib.hooks.gcp_mlengine_hook.**MLEngineHook**(*gcp_conn_id='google_cloud_default'*, *delegate_to=None*)
Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

create_job(*project_id*, *job*, *use_existing_job_fn=None*)
Launches a MLEngine job and wait for it to reach a terminal state.

Parameters

- **project_id** (*string*) – The Google Cloud project id within which MLEngine job will be launched.
- **job** (*dict*) – MLEngine Job object that should be provided to the MLEngine API, such as:

```
{
  'jobId': 'my_job_id',
  'trainingInput': {
    'scaleTier': 'STANDARD_1',
    ...
  }
}
```

- **use_existing_job_fn** (*function*) – In case that a MLEngine job with the same job_id already exist, this method (if provided) will decide whether we should use this existing job, continue waiting for it to finish and returning the job object. It should accepts a MLEngine job object, and returns a boolean value indicating whether it is OK to reuse the existing job. If 'use_existing_job_fn' is not provided, we by default reuse the existing MLEngine job.

Returns The MLEngine job object if the job successfully reach a terminal state (which might be FAILED or CANCELLED state).

Return type dict

create_model(*project_id*, *model*)
Create a Model. Blocks until finished.

create_version(*project_id*, *model_name*, *version_spec*)
Creates the Version on Google Cloud ML Engine.

Returns the operation if the version was created successfully and raises an error otherwise.

delete_version(*project_id*, *model_name*, *version_name*)
Deletes the given version of a model. Blocks until finished.

get_conn()
Returns a Google MLEngine service object.

get_model(*project_id*, *model_name*)
Gets a Model. Blocks until finished.

list_versions(*project_id*, *model_name*)
Lists all available versions of a model. Blocks until finished.

set_default_version(*project_id*, *model_name*, *version_name*)
Sets a version to be the default. Blocks until finished.

class airflow.contrib.hooks.gcp_pubsub_hook.**PubSubHook**(*gcp_conn_id='google_cloud_default'*, *delegate_to=None*)
Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Hook for accessing Google Pub/Sub.

The GCP project against which actions are applied is determined by the project embedded in the Connection referenced by gcp_conn_id.

acknowledge(*project*, *subscription*, *ack_ids*)
Pulls up to `max_messages` messages from Pub/Sub subscription.

Parameters

- **project** (*string*) – the GCP project name or ID in which to create the topic
- **subscription** (*string*) – the Pub/Sub subscription name to delete; do not include the 'projects/{project}/topics/' prefix.
- **ack_ids** (*list*) – List of ReceivedMessage ackIds from a previous pull response

create_subscription(*topic_project*, *topic*, *subscription=None*, *subscription_project=None*, *ack_deadline_secs=10*, *fail_if_exists=False*)
Creates a Pub/Sub subscription, if it does not already exist.

Parameters

- **topic_project** (*string*) – the GCP project ID of the topic that the subscription will be bound to.
- **topic** (*string*) – the Pub/Sub topic name that the subscription will be bound to create; do not include the `projects/{project}/subscriptions/` prefix.
- **subscription** (*string*) – the Pub/Sub subscription name. If empty, a random name will be generated using the uuid module
- **subscription_project** (*string*) – the GCP project ID where the subscription will be created. If unspecified, `topic_project` will be used.

- **ack_deadline_secs** (*int*) – Number of seconds that a subscriber has to acknowledge each message pulled from the subscription
- **fail_if_exists** (*bool*) – if set, raise an exception if the topic already exists

Returns subscription name which will be the system-generated value if the `subscription` parameter is not supplied

Return type string

create_topic(*project*, *topic*, *fail_if_exists=False*)
Creates a Pub/Sub topic, if it does not already exist.

Parameters

- **project** (*string*) – the GCP project ID in which to create the topic
- **topic** (*string*) – the Pub/Sub topic name to create; do not include the `projects/{project}/topics/` prefix.
- **fail_if_exists** (*bool*) – if set, raise an exception if the topic already exists

delete_subscription(*project*, *subscription*, *fail_if_not_exists=False*)
Deletes a Pub/Sub subscription, if it exists.

Parameters

- **project** (*string*) – the GCP project ID where the subscription exists
- **subscription** (*string*) – the Pub/Sub subscription name to delete; do not include the `projects/{project}/subscriptions/` prefix.
- **fail_if_not_exists** (*bool*) – if set, raise an exception if the topic does not exist

delete_topic(*project*, *topic*, *fail_if_not_exists=False*)
Deletes a Pub/Sub topic if it exists.

Parameters

- **project** (*string*) – the GCP project ID in which to delete the topic
- **topic** (*string*) – the Pub/Sub topic name to delete; do not include the `projects/{project}/topics/` prefix.
- **fail_if_not_exists** (*bool*) – if set, raise an exception if the topic does not exist

get_conn()
Returns a Pub/Sub service object.

Return type apiclient.discovery.Resource

publish(*project*, *topic*, *messages*)
Publishes messages to a Pub/Sub topic.

Parameters

- **project** (*string*) – the GCP project ID in which to publish
- **topic** (*string*) – the Pub/Sub topic to which to publish; do not include the `projects/{project}/topics/` prefix.
- **messages** (list of PubSub messages; see http://cloud.google.com/pubsub/docs/reference/rest/v1/PubsubMessage) – messages to publish; if the data field in a message is set, it should already be base64 encoded.

pull(*project*, *subscription*, *max_messages*, *return_immediately=False*)
Pulls up to `max_messages` messages from Pub/Sub subscription.

Parameters

- **project** (*string*) – the GCP project ID where the subscription exists
- **subscription** (*string*) – the Pub/Sub subscription name to pull from; do not include the 'projects/{project}/topics/' prefix.
- **max_messages** (*int*) – The maximum number of messages to return from the Pub/Sub API.
- **return_immediately** (*bool*) – If set, the Pub/Sub API will immediately return if no messages are available. Otherwise, the request will block for an undisclosed, but bounded period of time

:return A list of Pub/Sub ReceivedMessage objects each containing an `ackId` property and a `message` property, which includes the base64-encoded message content. See https://cloud.google.com/pubsub/docs/reference/rest/v1/ projects.subscriptions/pull#ReceivedMessage

class airflow.contrib.hooks.gcs_hook.**GoogleCloudStorageHook**(*google_cloud_storage_conn_id='google_clou*
delegate_to=None)

Bases: *airflow.contrib.hooks.gcp_api_base_hook.GoogleCloudBaseHook*

Interact with Google Cloud Storage. This hook uses the Google Cloud Platform connection.

copy(*source_bucket*, *source_object*, *destination_bucket=None*, *destination_object=None*)

Copies an object from a bucket to another, with renaming if requested.

destination_bucket or destination_object can be omitted, in which case source bucket/object is used, but not both.

Parameters

- **source_bucket** (*string*) – The bucket of the object to copy from.
- **source_object** (*string*) – The object to copy.
- **destination_bucket** (*string*) – The destination of the object to copied to. Can be omitted; then the same bucket is used.
- **destination_object** – The (renamed) path of the object if given. Can be omitted; then the same name is used.

create_bucket(*bucket_name*, *storage_class='MULTI_REGIONAL'*, *location='US'*, *project_id=None*, *labels=None*)

Creates a new bucket. Google Cloud Storage uses a flat namespace, so you can't create a bucket with a name that is already in use.

See also:

For more information, see Bucket Naming Guidelines: https://cloud.google.com/storage/docs/bucketnaming.html#requirements

Parameters

- **bucket_name** (*string*) – The name of the bucket.
- **storage_class** (*string*) – This defines how objects in the bucket are stored and determines the SLA and the cost of storage. Values include
  - MULTI_REGIONAL
  - REGIONAL
  - STANDARD

- NEARLINE
- COLDLINE.

If this value is not specified when the bucket is created, it will default to STANDARD.

- **location** (*string*) – The location of the bucket. Object data for objects in the bucket resides in physical storage within this region. Defaults to US.

  See also:

  https://developers.google.com/storage/docs/bucket-locations

- **project_id** (*string*) – The ID of the GCP Project.
- **labels** (*dict*) – User-provided labels, in key/value pairs.

Returns If successful, it returns the `id` of the bucket.

delete(*bucket*, *object*, *generation=None*)
Delete an object if versioning is not enabled for the bucket, or if generation parameter is used.

Parameters

- **bucket** (*string*) – name of the bucket, where the object resides
- **object** (*string*) – name of the object to delete
- **generation** (*string*) – if present, permanently delete the object of this generation

Returns True if succeeded

download(*bucket*, *object*, *filename=None*)
Get a file from Google Cloud Storage.

Parameters

- **bucket** (*string*) – The bucket to fetch from.
- **object** (*string*) – The object to fetch.
- **filename** (*string*) – If set, a local file path where the file should be written to.

exists(*bucket*, *object*)
Checks for the existence of a file in Google Cloud Storage.

Parameters

- **bucket** (*string*) – The Google cloud storage bucket where the object is.
- **object** (*string*) – The name of the object to check in the Google cloud storage bucket.

get_conn()
Returns a Google Cloud Storage service object.

get_crc32c(*bucket*, *object*)
Gets the CRC32c checksum of an object in Google Cloud Storage.

Parameters

- **bucket** (*string*) – The Google cloud storage bucket where the object is.
- **object** (*string*) – The name of the object to check in the Google cloud storage bucket.

get_md5hash(*bucket*, *object*)
Gets the MD5 hash of an object in Google Cloud Storage.

Parameters

- **bucket** (*string*) – The Google cloud storage bucket where the object is.
- **object** (*string*) – The name of the object to check in the Google cloud storage bucket.

get_size(*bucket*, *object*)
Gets the size of a file in Google Cloud Storage.

Parameters

- **bucket** (*string*) – The Google cloud storage bucket where the object is.
- **object** (*string*) – The name of the object to check in the Google cloud storage bucket.

is_updated_after(*bucket*, *object*, *ts*)
Checks if an object is updated in Google Cloud Storage.

Parameters

- **bucket** (*string*) – The Google cloud storage bucket where the object is.
- **object** (*string*) – The name of the object to check in the Google cloud storage bucket.
- **ts** (*datetime*) – The timestamp to check against.

list(*bucket*, *versions=None*, *maxResults=None*, *prefix=None*, *delimiter=None*)
List all objects from the bucket with the give string prefix in name

Parameters

- **bucket** (*string*) – bucket name
- **versions** (*boolean*) – if true, list all versions of the objects
- **maxResults** (*integer*) – max count of items to return in a single page of responses
- **prefix** (*string*) – prefix string which filters objects whose name begin with this prefix
- **delimiter** (*string*) – filters objects based on the delimiter (for e.g '.csv')

Returns a stream of object names matching the filtering criteria

rewrite(*source_bucket*, *source_object*, *destination_bucket*, *destination_object=None*)
Has the same functionality as copy, except that will work on files over 5 TB, as well as when copying between locations and/or storage classes.

destination_object can be omitted, in which case source_object is used.

Parameters

- **source_bucket** (*string*) – The bucket of the object to copy from.
- **source_object** (*string*) – The object to copy.
- **destination_bucket** (*string*) – The destination of the object to copied to.
- **destination_object** – The (renamed) path of the object if given. Can be omitted; then the same name is used.

upload(*bucket*, *object*, *filename*, *mime_type='application/octet-stream'*)
Uploads a local file to Google Cloud Storage.

Parameters

- **bucket** (*string*) – The bucket to upload to.
- **object** (*string*) – The object name to set when uploading the local file.
- **filename** (*string*) – The local file path to the file to be uploaded.

- **mime_type** (*string*) – The MIME type to set when uploading the file.

class airflow.contrib.hooks.redshift_hook.**RedshiftHook**(*aws_conn_id='aws_default'*)
Bases: *airflow.contrib.hooks.aws_hook.AwsHook*

Interact with AWS Redshift, using the boto3 library

cluster_status(*cluster_identifier*)
Return status of a cluster

Parameters **cluster_identifier** (*str*) – unique identifier of a cluster

create_cluster_snapshot(*snapshot_identifier*, *cluster_identifier*)
Creates a snapshot of a cluster

Parameters

- **snapshot_identifier** (*str*) – unique identifier for a snapshot of a cluster
- **cluster_identifier** (*str*) – unique identifier of a cluster

delete_cluster(*cluster_identifier*, *skip_final_cluster_snapshot=True*, *final_cluster_snapshot_identifier=''*)
Delete a cluster and optionally create a snapshot

Parameters

- **cluster_identifier** (*str*) – unique identifier of a cluster
- **skip_final_cluster_snapshot** (*bool*) – determines cluster snapshot creation
- **final_cluster_snapshot_identifier** (*str*) – name of final cluster snapshot

describe_cluster_snapshots(*cluster_identifier*)
Gets a list of snapshots for a cluster

Parameters **cluster_identifier** (*str*) – unique identifier of a cluster

restore_from_cluster_snapshot(*cluster_identifier*, *snapshot_identifier*)
Restores a cluster from its snapshot

Parameters

- **cluster_identifier** (*str*) – unique identifier of a cluster
- **snapshot_identifier** (*str*) – unique identifier for a snapshot of a cluster

class airflow.contrib.hooks.slack_webhook_hook.**SlackWebhookHook**(*http_conn_id=None*, *webhook_token=None*, *message=''*, *channel=None*, *username=None*, *icon_emoji=None*, *link_names=False*, *proxy=None*, **args*, ***kwargs*)
Bases: *airflow.hooks.http_hook.HttpHook*

This hook allows you to post messages to Slack using incoming webhooks. Takes both Slack webhook token directly and connection that has Slack webhook token. If both supplied, Slack webhook token will be used.

Each Slack webhook token can be pre-configured to use a specific channel, username and icon. You can override these defaults in this hook.

Parameters

- **http_conn_id** (*str*) – connection that has Slack webhook token in the extra field
- **webhook_token** (*str*) – Slack webhook token
- **message** (*str*) – The message you want to send on Slack
- **channel** (*str*) – The channel the message should be posted to
- **username** (*str*) – The username to post to slack with
- **icon_emoji** (*str*) – The emoji to use as icon for the user posting to Slack
- **link_names** (*bool*) – Whether or not to find and link channel and usernames in your message
- **proxy** (*str*) – Proxy to use to make the Slack webhook call

execute()

Remote Popen (actually execute the slack webhook call)

Parameters

- **cmd** – command to remotely execute
- **kwargs** – extra arguments to Popen (see subprocess.Popen)

class airflow.contrib.hooks.spark_sql_hook.**SparkSqlHook**(*sql*, *conf=None*, *conn_id='spark_sql_default'*, *total_executor_cores=None*, *executor_cores=None*, *executor_memory=None*, *keytab=None*, *principal=None*, *master='yarn'*, *name='default-name'*, *num_executors=None*, *verbose=True*, *yarn_queue='default'*)

Bases: airflow.hooks.base_hook.BaseHook

This hook is a wrapper around the spark-sql binary. It requires that the "spark-sql" binary is in the PATH. :param sql: The SQL query to execute :type sql: str :param conf: arbitrary Spark configuration property :type conf: str (format: PROP=VALUE) :param conn_id: connection_id string :type conn_id: str :param total_executor_cores: (Standalone & Mesos only) Total cores for all executors

(Default: all the available cores on the worker)

Parameters

- **executor_cores** (*int*) – (Standalone & YARN only) Number of cores per executor (Default: 2)
- **executor_memory** (*str*) – Memory per executor (e.g. 1000M, 2G) (Default: 1G)
- **keytab** (*str*) – Full path to the file that contains the keytab
- **master** (*str*) – spark://host:port, mesos://host:port, yarn, or local
- **name** (*str*) – Name of the job.

- **num_executors** (*int*) – Number of executors to launch
- **verbose** (*bool*) – Whether to pass the verbose flag to spark-sql
- **yarn_queue** (*str*) – The YARN queue to submit to (Default: "default")

run_query(*cmd=''*, ***kwargs*)
Remote Popen (actually execute the Spark-sql query)

Parameters

- **cmd** – command to remotely execute
- **kwargs** – extra arguments to Popen (see subprocess.Popen)

class airflow.contrib.hooks.sqoop_hook.**SqoopHook**(*conn_id='sqoop_default'*, *verbose=False*, *num_mappers=None*, *hcatalog_database=None*, *hcatalog_table=None*, *properties=None*)
Bases: airflow.hooks.base_hook.BaseHook, airflow.utils.log.logging_mixin.LoggingMixin

This hook is a wrapper around the sqoop 1 binary. To be able to use the hook it is required that "sqoop" is in the PATH.

Additional arguments that can be passed via the 'extra' JSON field of the sqoop connection: * job_tracker: Job tracker locall|jobtracker:port. * namenode: Namenode. * lib_jars: Comma separated jar files to include in the classpath. * files: Comma separated files to be copied to the map reduce cluster. * archives: Comma separated archives to be unarchived on the compute

machines.

- password_file: Path to file containing the password.

Parameters

- **conn_id** (*str*) – Reference to the sqoop connection.
- **verbose** (*bool*) – Set sqoop to verbose.
- **num_mappers** (*int*) – Number of map tasks to import in parallel.
- **properties** (*dict*) – Properties to set via the -D argument

Popen(*cmd*, ***kwargs*)
Remote Popen

Parameters

- **cmd** – command to remotely execute
- **kwargs** – extra arguments to Popen (see subprocess.Popen)

Returns handle to subprocess

export_table(*table*, *export_dir*, *input_null_string*, *input_null_non_string*, *staging_table*, *clear_staging_table*, *enclosed_by*, *escaped_by*, *input_fields_terminated_by*, *input_lines_terminated_by*, *input_optionally_enclosed_by*, *batch*, *relaxed_isolation*, *extra_export_options=None*)
Exports Hive table to remote location. Arguments are copies of direct sqoop command line Arguments :param table: Table remote destination :param export_dir: Hive table to export :param input_null_string: The string to be interpreted as null for

string columns

Parameters

- **input_null_non_string** – The string to be interpreted as null for non-string columns
- **staging_table** – The table in which data will be staged before being inserted into the destination table
- **clear_staging_table** – Indicate that any data present in the staging table can be deleted
- **enclosed_by** – Sets a required field enclosing character
- **escaped_by** – Sets the escape character
- **input_fields_terminated_by** – Sets the field separator character
- **input_lines_terminated_by** – Sets the end-of-line character
- **input_optionally_enclosed_by** – Sets a field enclosing character
- **batch** – Use batch mode for underlying statement execution
- **relaxed_isolation** – Transaction isolation to read uncommitted for the mappers
- **extra_export_options** – Extra export options to pass as dict. If a key doesn't have a value, just pass an empty string to it. Don't include prefix of – for sqoop options.

import_query(*query*, *target_dir*, *append=False*, *file_type='text'*, *split_by=None*, *direct=None*, *driver=None*, *extra_import_options=None*)

Imports a specific query from the rdbms to hdfs :param query: Free format query to run :param target_dir: HDFS destination dir :param append: Append data to an existing dataset in HDFS :param file_type: "avro", "sequence", "text" or "parquet"

Imports data to hdfs into the specified format. Defaults to text.

Parameters

- **split_by** – Column of the table used to split work units
- **direct** – Use direct import fast path
- **driver** – Manually specify JDBC driver class to use
- **extra_import_options** – Extra import options to pass as dict. If a key doesn't have a value, just pass an empty string to it. Don't include prefix of – for sqoop options.

import_table(*table*, *target_dir=None*, *append=False*, *file_type='text'*, *columns=None*, *split_by=None*, *where=None*, *direct=False*, *driver=None*, *extra_import_options=None*)

Imports table from remote location to target dir. Arguments are copies of direct sqoop command line arguments :param table: Table to read :param target_dir: HDFS destination dir :param append: Append data to an existing dataset in HDFS :param file_type: "avro", "sequence", "text" or "parquet".

Imports data to into the specified format. Defaults to text.

Parameters

- **columns** – <col,col,col...> Columns to import from table
- **split_by** – Column of the table used to split work units
- **where** – WHERE clause to use during import
- **direct** – Use direct connector if exists for the database

- **driver** – Manually specify JDBC driver class to use
- **extra_import_options** – Extra import options to pass as dict. If a key doesn't have a value, just pass an empty string to it. Don't include prefix of – for sqoop options.

class airflow.contrib.hooks.vertica_hook.**VerticaHook**(**args*, ***kwargs*)
Bases: *airflow.hooks.dbapi_hook.DbApiHook*

Interact with Vertica.

get_conn()
Returns verticaql connection object

3.19.5 Executors

Executors are the mechanism by which task instances get run.

class airflow.executors.local_executor.**LocalExecutor**(*parallelism=32*)
Bases: airflow.executors.base_executor.BaseExecutor

LocalExecutor executes tasks locally in parallel. It uses the multiprocessing Python library and queues to parallelize the execution of tasks.

end()
This method is called when the caller is done submitting job and is wants to wait synchronously for the job submitted previously to be all done.

execute_async(*key*, *command*, *queue=None*, *executor_config=None*)
This method will execute the command asynchronously.

start()
Executors may need to get things started. For example LocalExecutor starts N workers.

sync()
Sync will get called periodically by the heartbeat method. Executors should override this to perform gather statuses.

class airflow.executors.sequential_executor.**SequentialExecutor**
Bases: airflow.executors.base_executor.BaseExecutor

This executor will only run one task instance at a time, can be used for debugging. It is also the only executor that can be used with sqlite since sqlite doesn't support multiple connections.

Since we want airflow to work out of the box, it defaults to this SequentialExecutor alongside sqlite as you first install it.

end()
This method is called when the caller is done submitting job and is wants to wait synchronously for the job submitted previously to be all done.

execute_async(*key*, *command*, *queue=None*, *executor_config=None*)
This method will execute the command asynchronously.

sync()
Sync will get called periodically by the heartbeat method. Executors should override this to perform gather statuses.

3.19.5.1 Community-contributed executors

class airflow.contrib.executors.mesos_executor.**MesosExecutor**(*parallelism=32*)
Bases: airflow.executors.base_executor.BaseExecutor, airflow.www.utils.LoginMixin

MesosExecutor allows distributing the execution of task instances to multiple mesos workers.

Apache Mesos is a distributed systems kernel which abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively. See http://mesos.apache.org/

end()
This method is called when the caller is done submitting job and is wants to wait synchronously for the job submitted previously to be all done.

execute_async(*key*, *command*, *queue=None*, *executor_config=None*)
This method will execute the command asynchronously.

start()
Executors may need to get things started. For example LocalExecutor starts N workers.

sync()
Sync will get called periodically by the heartbeat method. Executors should override this to perform gather statuses.

Python Module Index

a

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X