
Airflow Documentation

Release

Maxime Beauchemin

May 18, 2017

Contents

1	Principles	3
2	Beyond the Horizon	5
3	Content	7
3.1	Installation	7
3.1.1	Getting Airflow	7
3.1.2	Extra Packages	7
3.2	Quick Start	9
3.2.1	What's Next?	9
3.3	Tutorial	9
3.3.1	Example Pipeline definition	9
3.3.2	It's a DAG definition file	11
3.3.3	Importing Modules	11
3.3.4	Default Arguments	11
3.3.5	Instantiate a DAG	12
3.3.6	Tasks	12
3.3.7	Templating with Jinja	12
3.3.8	Setting up Dependencies	13
3.3.9	Recap	13
3.3.10	Testing	14
3.3.10.1	Running the Script	14
3.3.10.2	Command Line Metadata Validation	15
3.3.10.3	Testing	15
3.3.10.4	Backfill	15
3.3.11	What's Next?	16
3.4	Configuration	16
3.4.1	Setting Configuration Options	16
3.4.2	Setting up a Backend	17
3.4.3	Connections	17
3.4.4	Scaling Out with Celery	18
3.4.5	Logs	19
3.4.6	Scaling Out on Mesos (community contributed)	19
3.4.7	Integration with systemd	20
3.4.8	Integration with upstart	20
3.5	UI / Screenshots	20
3.5.1	DAGs View	20

3.5.2	Tree View	21
3.5.3	Graph View	22
3.5.4	Gantt Chart	22
3.5.5	Task Duration	23
3.5.6	Code View	24
3.5.7	Task Instance Context Menu	25
3.6	Concepts	26
3.6.1	Core Ideas	26
3.6.1.1	DAGs	26
3.6.1.2	Operators	27
3.6.1.3	Tasks	29
3.6.1.4	Task Instances	29
3.6.1.5	Workflows	29
3.6.2	Additional Functionality	30
3.6.2.1	Hooks	30
3.6.2.2	Pools	30
3.6.2.3	Connections	30
3.6.2.4	Queues	31
3.6.2.5	XComs	31
3.6.2.6	Variables	32
3.6.2.7	Branching	32
3.6.2.8	SubDAGs	32
3.6.2.9	SLAs	35
3.6.2.10	Trigger Rules	35
3.6.2.11	Zombies & Undeads	35
3.6.2.12	Cluster Policy	35
3.6.2.13	Task Documentation & Notes	36
3.6.2.14	Jinja Templating	36
3.7	Data Profiling	37
3.7.1	Adhoc Queries	37
3.7.2	Charts	38
3.7.2.1	Chart Screenshot	39
3.7.2.2	Chart Form Screenshot	40
3.8	Command Line Interface	40
3.9	Scheduling & Triggers	40
3.9.1	DAG Runs	41
3.9.2	External Triggers	41
3.9.3	To Keep in Mind	41
3.10	Plugins	42
3.10.1	What for?	42
3.10.2	Why build on top of Airflow?	42
3.10.3	Interface	43
3.10.4	Example	43
3.11	Security	44
3.11.1	Web Authentication	44
3.11.1.1	Password	44
3.11.1.2	LDAP	45
3.11.1.3	Roll your own	45
3.11.2	Multi-tenancy	46
3.11.3	Kerberos	46
3.11.3.1	Limitations	46
3.11.3.2	Enabling kerberos	46
3.11.3.3	Using kerberos authentication	47
3.11.3.4	GitHub Enterprise (GHE) Authentication	47

3.11.3.5	Setting up GHE Authentication	48
3.12	FAQ	48
3.13	API Reference	49
3.13.1	Operators	49
3.13.1.1	BaseOperator	50
3.13.1.2	BaseSensorOperator	51
3.13.1.3	Operator API	52
3.13.1.4	Community-contributed Operators	57
3.13.2	Macros	57
3.13.2.1	Default Variables	57
3.13.2.2	Macros	58
3.13.3	Models	60
3.13.4	Hooks	68
3.13.4.1	Community contributed hooks	70
3.13.5	Executors	72
3.13.5.1	Community-contributed executors	73
Python Module Index		75



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.

3.1 Installation

3.1.1 Getting Airflow

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

```
pip install airflow
```

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

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

3.1.2 Extra Packages

The `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-pack-age	install command	enables
all	<code>pip install airflow[all]</code>	All Airflow features known to man
all_dbs	<code>pip install airflow[all_dbs]</code>	All databases integrations
async	<code>pip install airflow[async]</code>	Async worker classes for gunicorn
devel	<code>pip install airflow[devel]</code>	Minimum dev tools requirements
de-vel_hadoop	<code>pip install airflow[devel_hadoop]</code>	Airflow + dependencies on the Hadoop stack
celery	<code>pip install airflow[celery]</code>	CeleryExecutor
crypto	<code>pip install airflow[crypto]</code>	Encrypt connection passwords in metadata db
druid	<code>pip install airflow[druid]</code>	Druid.io related operators & hooks
gcp_api	<code>pip install airflow[gcp_api]</code>	Google Cloud Platform hooks and operators (using google-api-python-client)
gcloud	<code>pip install airflow[gcloud]</code>	Google Cloud Platform hooks (using gcloud; see <code>/airflow/contrib/hooks/gcloud/readme.md</code>)
jdbc	<code>pip install airflow[jdbc]</code>	JDBC hooks and operators
hdfs	<code>pip install airflow[hdfs]</code>	HDFS hooks and operators
hive	<code>pip install airflow[hive]</code>	All Hive related operators
kerberos	<code>pip install airflow[kerberos]</code>	kerberos integration for kerberized hadoop
ldap	<code>pip install airflow[ldap]</code>	ldap authentication for users
mssql	<code>pip install airflow[mssql]</code>	Microsoft SQL operators and hook, support as an Airflow backend
mysql	<code>pip install airflow[mysql]</code>	MySQL operators and hook, support as an Airflow backend
pass-word	<code>pip install airflow[password]</code>	Password Authentication for users
postgres	<code>pip install airflow[postgres]</code>	Postgres operators and hook, support as an Airflow backend
qds	<code>pip install airflow[qds]</code>	Enable QDS (qubole data services) support
rab-bitmq	<code>pip install airflow[rabbitmq]</code>	Rabbitmq support as a Celery backend
s3	<code>pip install airflow[s3]</code>	S3KeySensor, S3PrefixSensor
samba	<code>pip install airflow[samba]</code>	Hive2SambaOperator
slack	<code>pip install airflow[slack]</code>	SlackAPIPostOperator
vertica	<code>pip install airflow[vertica]</code>	Vertica hook support as an Airflow backend
cloudant	<code>pip install airflow[cloudant]</code>	Cloudant hook

3.2 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 airflow

# initialize the database
airflow initdb

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

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
```

3.2.1 What’s Next?

From this point, you can head to the [Tutorial](#) section for further examples or the configuration section if you’re ready to get your hands dirty.

3.3 Tutorial

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

3.3.1 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/airbnb/airflow/blob/master/airflow/example_dags/tutorial.py
"""
from airflow import DAG
from airflow.operators import BashOperator
from datetime import datetime, timedelta

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 6, 1),
    'email': ['airflow@airflow.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(
    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)
```


3.3.2 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.

3.3.3 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 import BashOperator
```

3.3.4 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@airflow.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 `:py:class: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.

3.3.5 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))
```

3.3.6 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.

3.3.7 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 }}`.

```
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.

For more information on the variables and macros that can be referenced in templates, make sure to read through the [Macros](#) section

3.3.8 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.

3.3.9 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 import BashOperator
from datetime import datetime, timedelta
```

```

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 6, 1),
    'email': ['airflow@airflow.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',
    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)

```

3.3.10 Testing

3.3.10.1 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.

3.3.10.2 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
```

3.3.10.3 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.

3.3.10.4 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
```

3.3.11 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!

3.4 Configuration

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

3.4.1 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 `celery_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

3.4.2 Setting up a 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**.

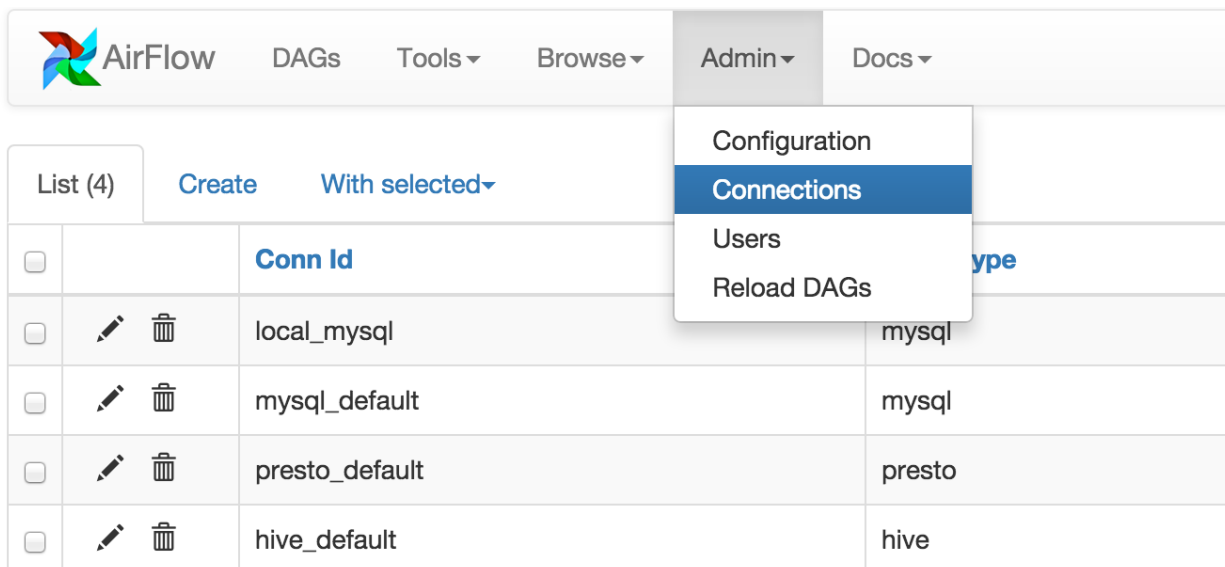
Note: If you decide to use **Postgres**, we recommend using the `psycopg2` driver and specifying it in your `SqlAlchemy` connection string. Also note that since `SqlAlchemy` does not expose a way to target a specific schema in the `Postgres` connection URI, you may want to set a default schema for your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

3.4.3 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.



The screenshot shows the Airflow web interface. At the top, there is a navigation bar with the AirFlow logo and several tabs: DAGs, Tools, Browse, Admin, and Docs. The Admin tab is currently selected, and a dropdown menu is open, showing options: Configuration, Connections (highlighted in blue), Users, and Reload DAGs. Below the navigation bar, there is a table of connections. The table has columns for a checkbox, a list of actions (edit and delete icons), a 'Conn Id' column, and a 'type' column. The table contains four rows of connections: local_mysql, mysql_default, presto_default, and hive_default.

<input type="checkbox"/>		Conn Id	type
<input type="checkbox"/>		local_mysql	mysql
<input type="checkbox"/>		mysql_default	mysql
<input type="checkbox"/>		presto_default	presto
<input type="checkbox"/>		hive_default	hive

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.

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. Please see the [Concepts](#) documentation for more information on environment variables and connections.

3.4.4 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 mean. 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.

3.4.5 Logs

Users can specify a logs folder in `airflow.cfg`. 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. At this time, Amazon S3 and Google Cloud Storage are supported. To enable this feature, `airflow.cfg` must be configured as in this example:

```
[core]
# Airflow can store logs remotely in AWS S3 or Google Cloud Storage. Users
# must supply a remote location URL (starting with either 's3://...' or
# 'gs://...') 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
```

Remote logging uses an existing Airflow connection to read/write logs. If you don’t have a connection properly setup, this will fail. In the above example, Airflow will try to use `S3Hook('MyS3Conn')`.

In the Airflow Web UI, local logs take precedence 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.

3.4.6 Scaling Out on Mesos (community contributed)

`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 machine 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 mesos slave machines 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](#).

3.4.7 Integration 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`.

3.4.8 Integration 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
```

3.5 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.

3.5.1 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.

 AirFlow

DAGs
















Tools ▾

Browse ▾

Admin ▾

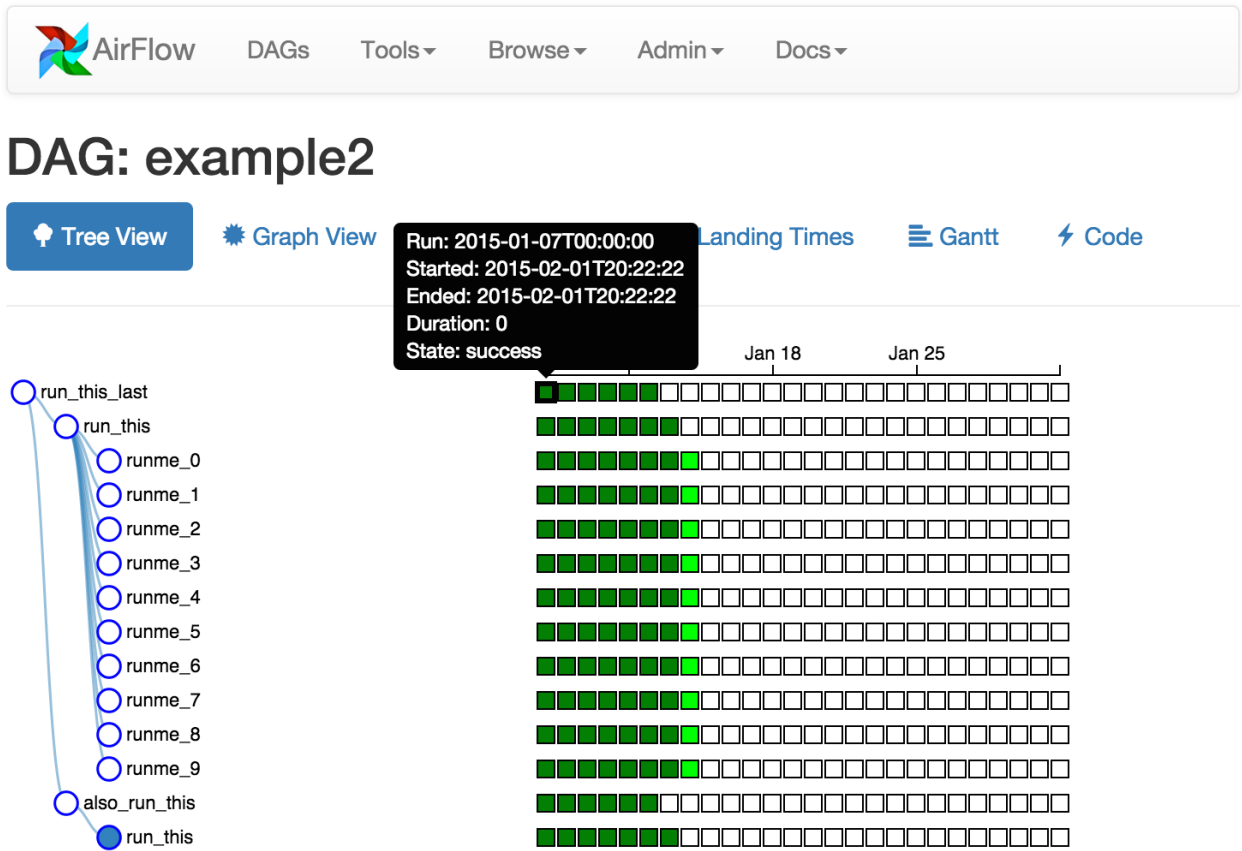
Docs ▾

DAGs

DAG	Filepath	Owner	Task by State	Links
example1	example_dags/example1.py	airflow	<div><div>80</div><div>1</div><div>0</div></div>	    
example2	example_dags/example2.py	airflow	<div><div>128</div><div>10</div><div>0</div></div>	    
example3	example_dags/example3.py	airflow	<div><div>138</div><div>5</div><div>0</div></div>	    

3.5.2 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.



DAG: example2

Tree View

Graph View

Landing Times

Gantt

Code

Run: 2015-01-07T00:00:00

Started: 2015-02-01T20:22:22

Ended: 2015-02-01T20:22:22

Duration: 0

State: success

Jan 18

Jan 25

run_this_last

run_this

runme_0

runme_1

runme_2

runme_3

runme_4

runme_5

runme_6

runme_7

runme_8

runme_9

also_run_this

run_this

runme_0

runme_1

runme_2

runme_3

runme_4

runme_5

runme_6

runme_7

runme_8

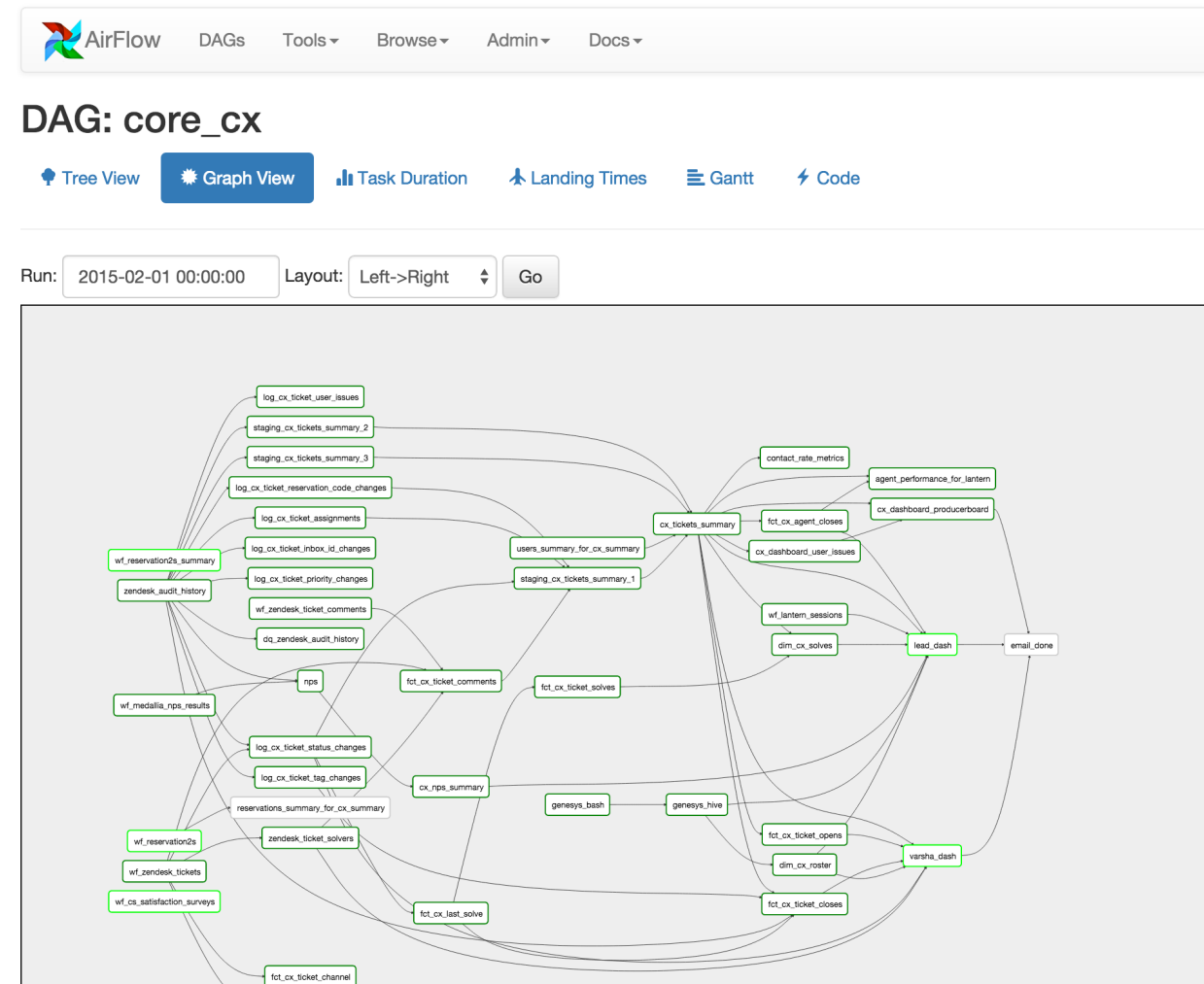
runme_9

3.5. UI / Screenshots

21

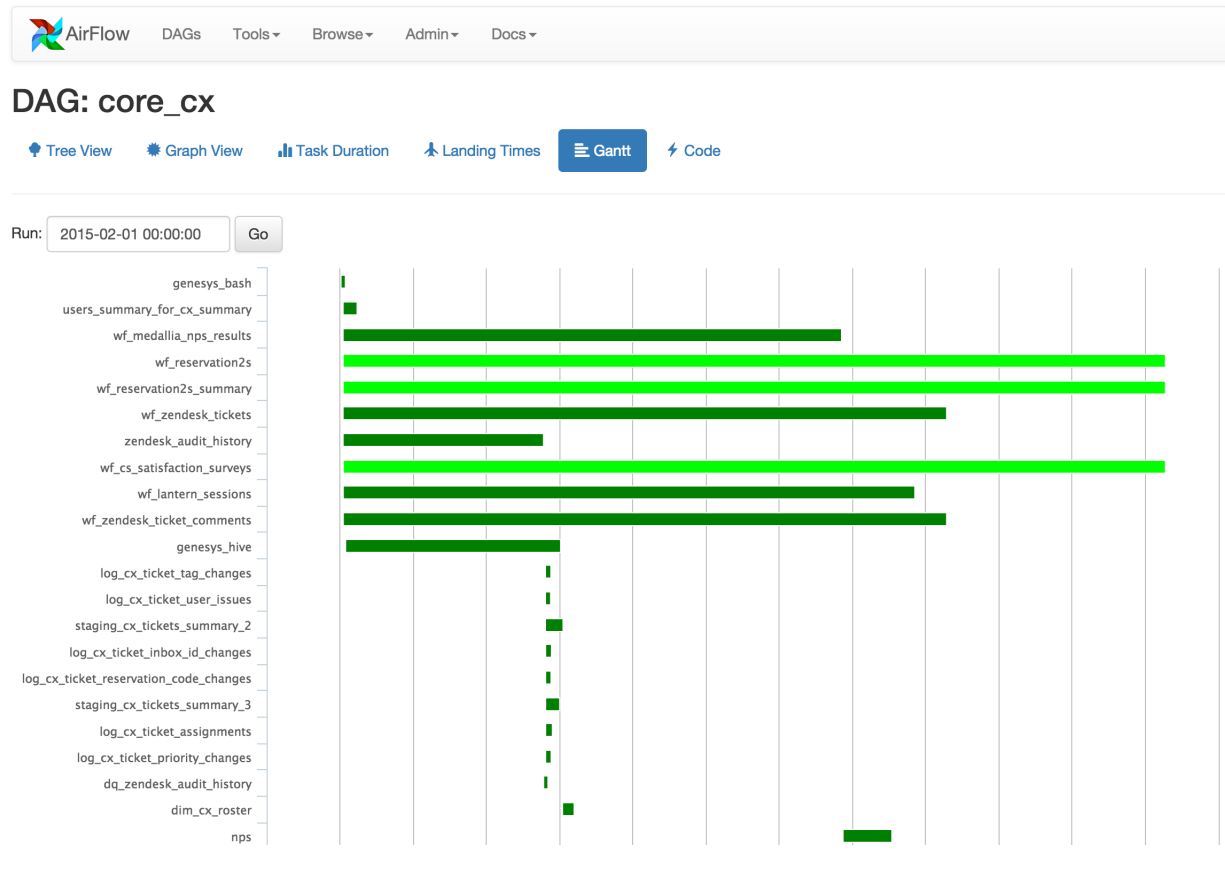
3.5.3 Graph View

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



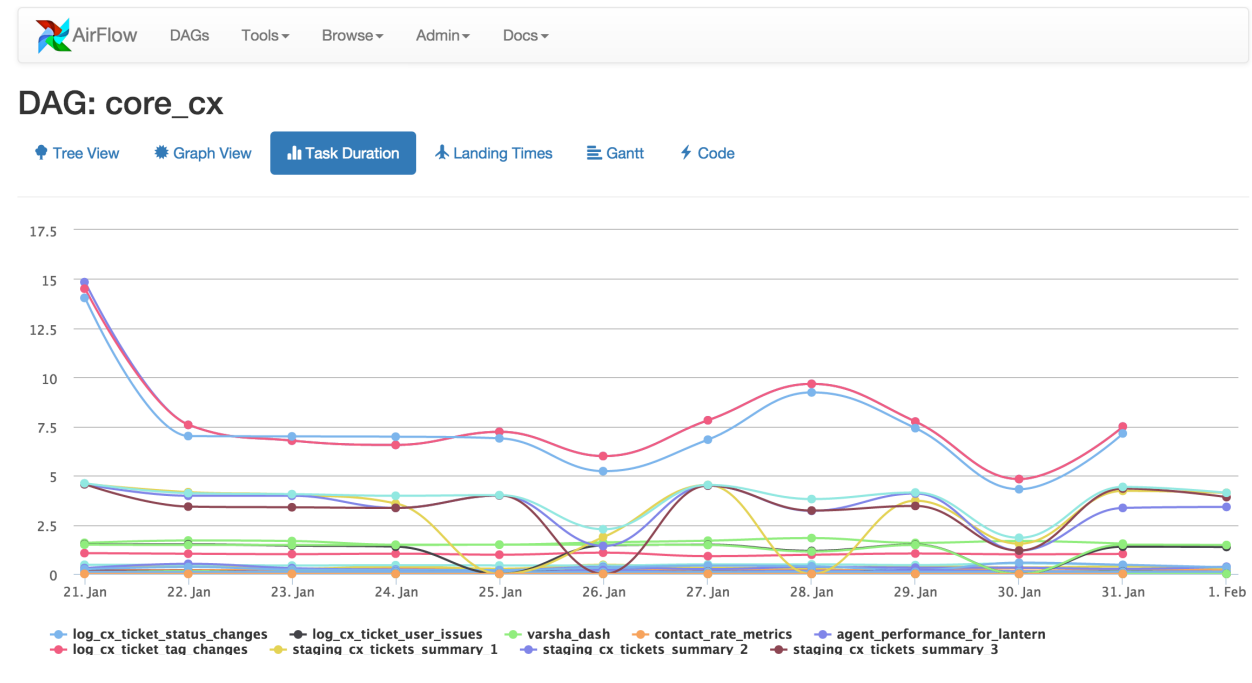
3.5.4 Gantt Chart

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.



3.5.5 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.



3.5.6 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.



AirFlow

DAGs

Tools ▾

Browse ▾

Admin ▾

Docs ▾

DAG: example1

🌳 Tree View

⚙️ Graph View

📊 Task Duration

🕒 Landing Times

≡ Gantt

⚡ Code

example_dags/example1.py

```
from airflow.operators import BashOperator, DummyOperator
from airflow.models import DAG
from datetime import datetime

args = {
    'owner': 'airflow',
    'start_date': datetime(2015, 1, 1),
}

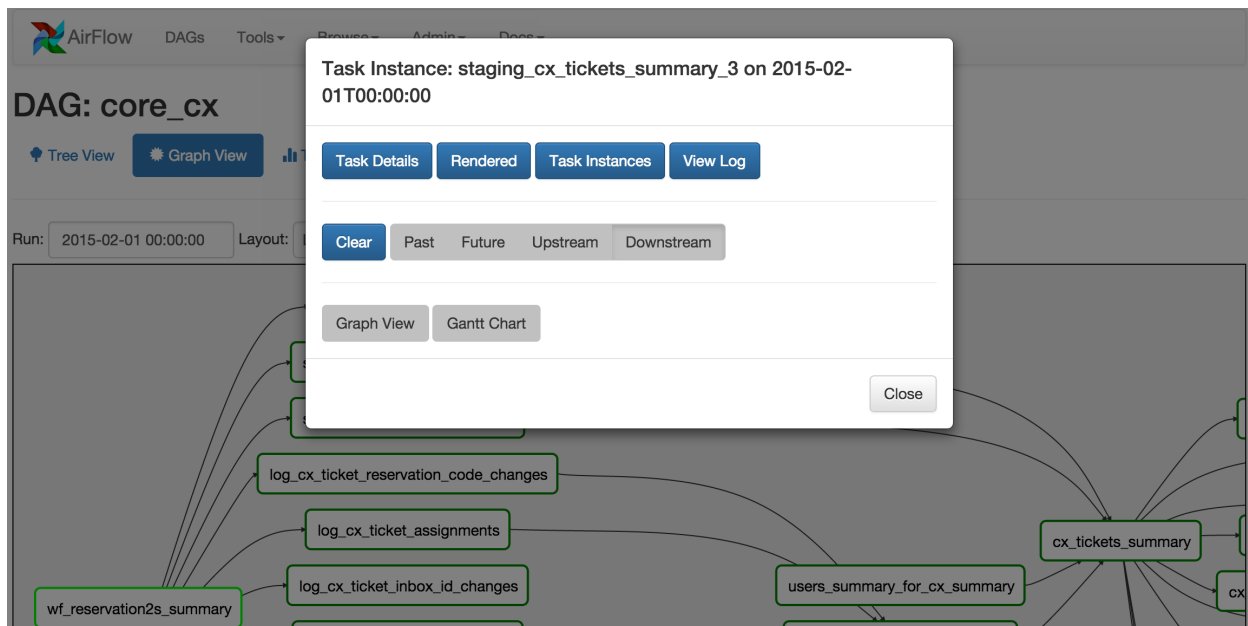
dag = DAG(dag_id='example1')

cmd = 'ls -l'
run_this_last = DummyOperator(
    task_id='run_this_last',
    default_args=args)
dag.add_task(run_this_last)

run_this = BashOperator(
    task_id='run_after_loop', bash_command='echo 1',
    default_args=args)
dag.add_task(run_this)
run_this.set_downstream(run_this_last)
for i in range(9):
    i = str(i)
    task = BashOperator(
```

3.5.7 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.



3.6 Concepts

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

3.6.1 Core Ideas

3.6.1.1 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.

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=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
```

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
```

3.6.1.2 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

- `SqlOperator` - 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)

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 seconds. 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:
    (
        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!"))
    )
```

3.6.1.3 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.

3.6.1.4 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.

3.6.1.5 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.

3.6.2 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.

3.6.2.1 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.

3.6.2.2 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.

3.6.2.3 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.

Airflow also has the ability to reference connections via environment variables from the operating system. The environment variable needs to be prefixed with `AIRFLOW_CONN_` to be considered a connection. 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`. Airflow assumes the value returned from the environment variable to be in a URI format (e.g. `postgres://user:password@localhost:5432/master`).

3.6.2.4 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).

3.6.2.5 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
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.

3.6.2.6 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. 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.

3.6.2.7 Branching

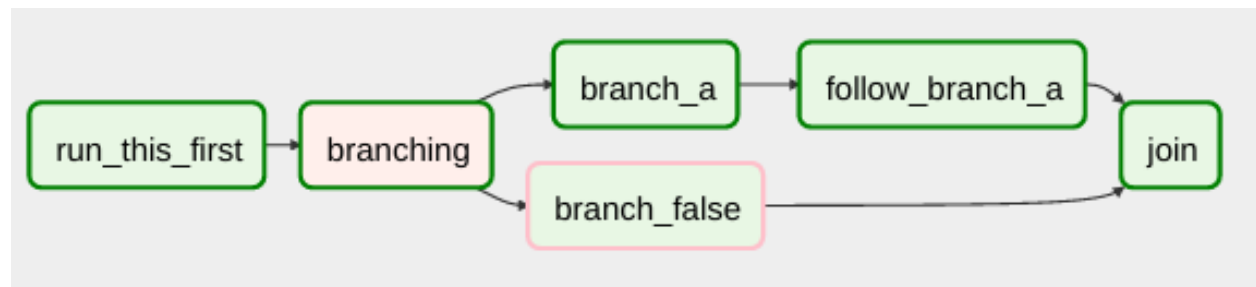
Sometimes you need a workflow to branch, or only go down a certain path based on an arbitrary condition which is typically related to something that happened in an upstream task. One way to do this is by using the BranchPythonOperator.

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.

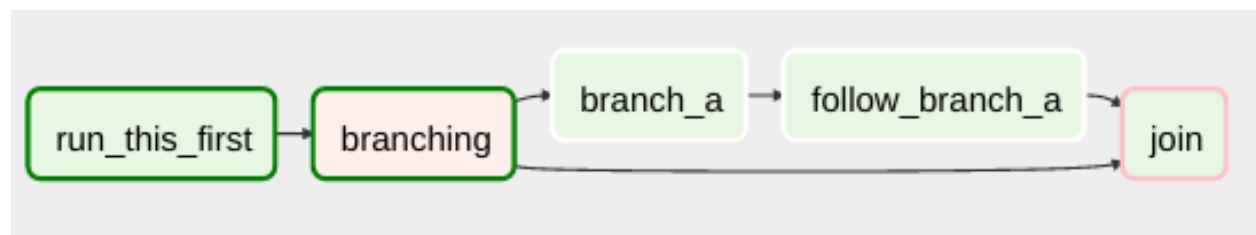
Note that using tasks with depends_on_past=True downstream from BranchPythonOperator is logically unsound as skipped status will invariably lead to block tasks that depend on their past successes. skipped states propagates where all directly upstream tasks are skipped.

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

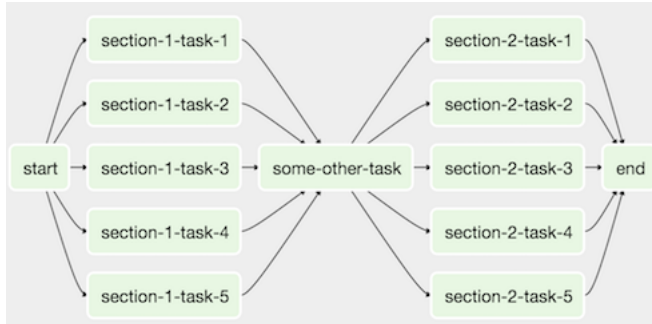


3.6.2.8 SubDAGs

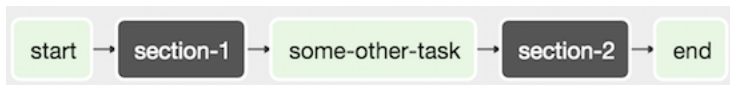
SubDAGs are perfect for repeating patterns. Defining a function that returns a DAG object is a nice design pattern when using Airflow.

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 import DummyOperator

# 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 import SubDagOperator
from dags.subdag import sub_dag

PARENT_DAG_NAME = 'parent_dag'
CHILD_DAG_NAME = 'child_dag'
```

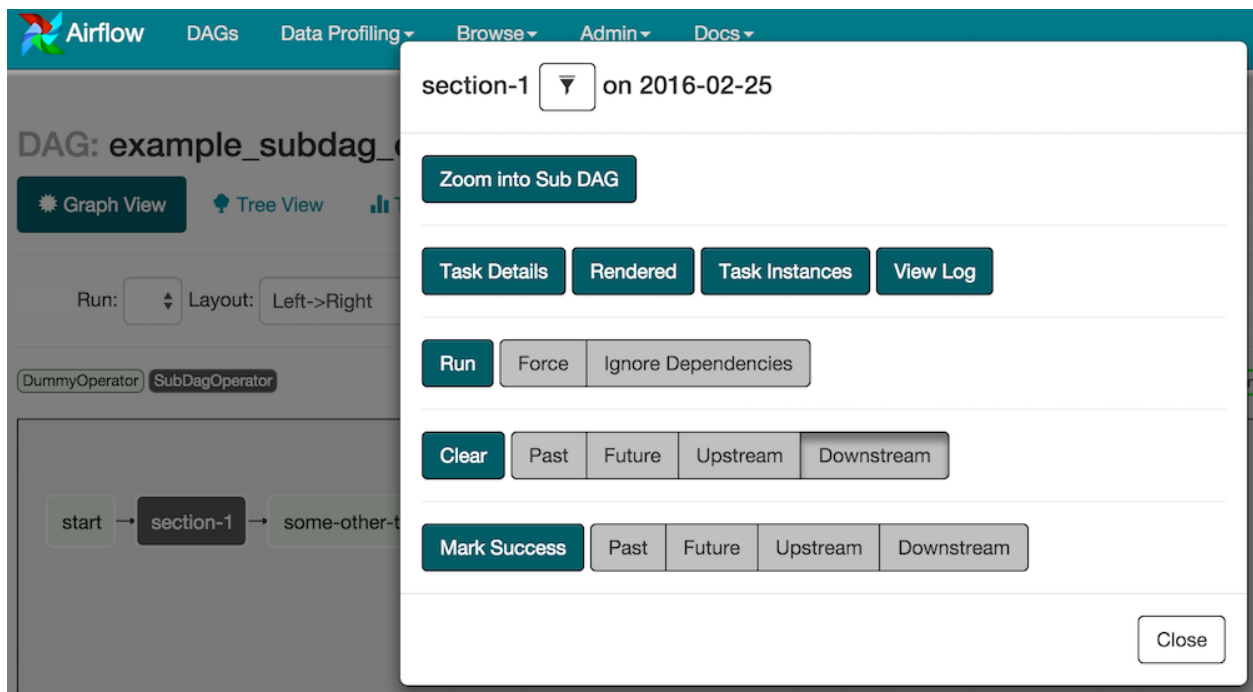
```

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.

3.6.2.9 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.

3.6.2.10 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.

3.6.2.11 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.

3.6.2.12 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)
```

3.6.2.13 Task Documentation & Notes

It's possible to add documentation or notes to your task objects that become visible in the “Task Details” view in the web interface. 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

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.

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

This content will get rendered as markdown in the “Task Details” page.

3.6.2.14 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 ',
    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.

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.

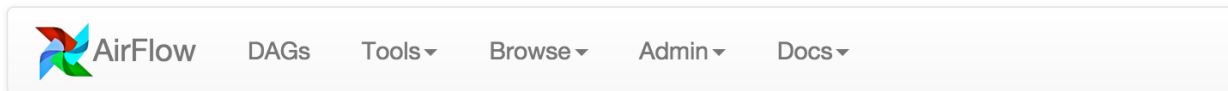
Note: packaged dags cannot contain dynamic libraries (eg. `libz.so`) these need to be available on the system if a module needs those. In other words only pure python modules can be packaged.

3.7 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.

3.7.1 Adhoc Queries

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



Ad Hoc Query

airflow_db

1 `SELECT * FROM task_instance LIMIT 1000`

Show entries Search:

task_id	dag_id	execution_date	start_date	end_date	duration	state
agent_performance_for_lantern	core_cx	2014-11-22 00:00:00	2014-11-23 22:50:51	2014-11-23 22:54:54	243	success
agent_performance_for_lantern	core_cx	2014-11-23 00:00:00	2014-11-24 23:04:53	2014-11-24 23:08:58	245	success
agent_performance_for_lantern	core_cx	2014-11-24 00:00:00	2014-11-26 00:25:46	2014-11-26 00:29:27	220	success
agent_performance_for_lantern	core_cx	2014-11-25 00:00:00	2014-11-29 00:05:02	2014-11-29 00:09:07	244	success
agent_performance_for_lantern	core_cx	2014-11-26 00:00:00	2014-11-29 01:46:23	2014-11-29 02:05:50	1167	success
agent_performance_for_lantern	core_cx	2014-11-27 00:00:00	2014-11-29 18:06:04	2014-11-29 18:10:04	239	success
agent_performance_for_lantern	core_cx	2014-11-28 00:00:00	2014-11-29 18:20:12	2014-11-29 18:23:45	212	success
agent_performance_for_lantern	core_cx	2014-11-29 00:00:00	2014-12-01 05:46:37	2014-12-01 05:50:32	234	success

3.7.2 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 connectons, 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.

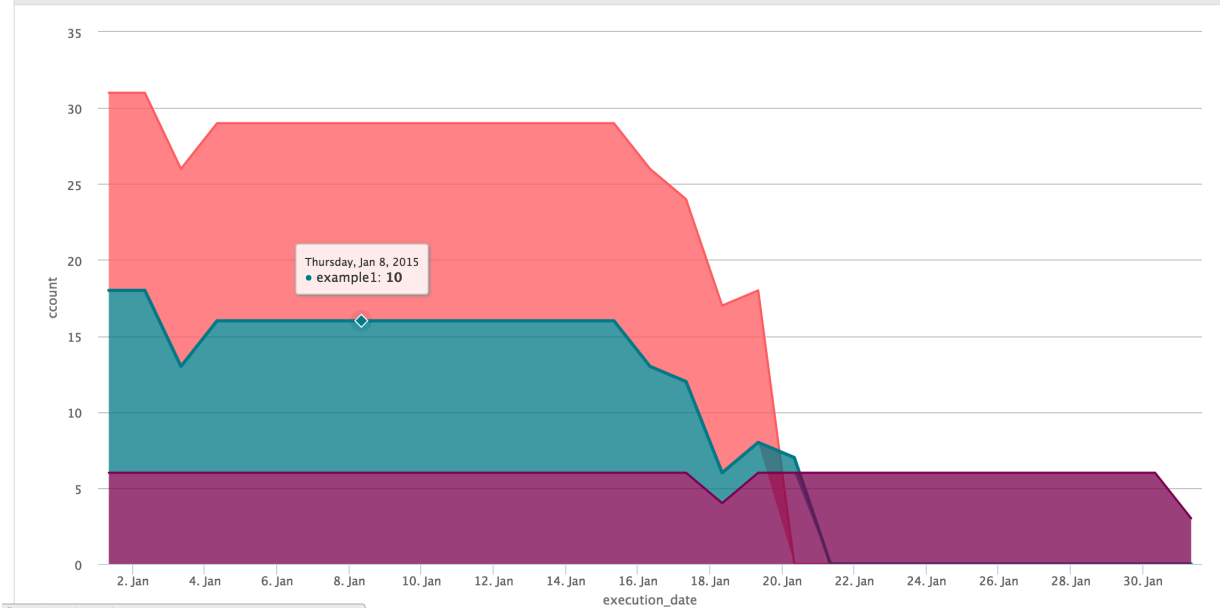
3.7.2.1 Chart Screenshot

Tasks

SQL

```
SELECT dag_id, execution_date, count(*) as ccount
FROM task_instance
GROUP BY dag_id, execution_date
```

Chart



3.7.2.2 Chart Form Screenshot

Label

Can include {{ templated_fields }} and {{ macros }}

Owner

The chart's owner, mostly used for reference and filtering in the list view.

Source Database

Chart Type

Line Chart

The type of chart to be displayed

Show Datatable

☐

Whether to display an interactive data table under the chart.

X Is Date

☒

Whether the X axis should be casted as a date field. Expect most intelligible date formats to get casted prop

Y Log Scale

☐

Whether to use a log scale for the Y axis.

Display the SQL Statement

☒

Whether to display the SQL statement as a collapsible section in the chart page.

Chart Height

600

Height of the chart, in pixels.

SQL Layout

SELECT series, x, y FROM ...

Defines the layout of the SQL that the application should expect. Depending on the tables you are sourcing from, it may make more sense t

SQL

1

SELECT series, x, y FROM table

3.8 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.

3.9 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.

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
```

3.9.1 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	Run once a year at midnight of January 1	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.

3.9.2 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.

3.9.3 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.

- 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`)
- 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.

3.10 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**, **macros**, **executors** and web **views** get integrated to Airflow's main collections and become available for use.

3.10.1 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
- ...

3.10.2 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 it's deployment logistics
- Basic charting capabilities, underlying libraries and abstractions

3.10.3 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 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 = []
```

3.10.4 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.executors.base_executor import BaseExecutor

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

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

# Will show up under airflow.executors.PluginExecutor
class PluginExecutor(BaseExecutor):
    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 intergrate 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='http://pythonhosted.org/airflow/')

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

```

3.11 Security

3.11.1 Web Authentication

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 create your own.

3.11.1.1 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()
```

3.11.1.2 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]
uri = ldaps://<your.ldap.server>:<port>
user_filter = objectClass=*
user_name_attr = uid # in case of Active Directory you would use sAMAccountName
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
bind_user = cn=Manager,dc=example,dc=com
bind_password = insecure
basedn = dc=example,dc=com
cacert = /etc/ca/ldap_ca.crt
search_scope = LEVEL # Set this to SUBTREE if using Active Directory, and not_
↳specifying an Organizational Unit
```

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.

3.11.1.3 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
```

3.11.2 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` as true in your `airflow.cfg`. With this, when a user authenticates and logs into webserver, it will see only the dags which it is owner of. A `super_user`, will be able to see all the dags although. This makes the web UI a multi-tenant UI, where a user will only be able to see dags created by itself.

3.11.3 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.

3.11.3.1 Limitations

Please note that at this time not all hooks have been adjusted to make use of this functionality yet. 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.

3.11.3.2 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.

3.11.3.3 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
```

3.11.3.4 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.

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.

```
[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 = example_team_1, example_team_2
```

3.11.3.5 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

3.12 FAQ

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.
- Is your `start_date` set properly? The Airflow scheduler triggers the task soon after the `start_date + scheduler_interval` is passed.
- Is your `start_date` beyond where you can see it in the UI? If you set your it 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 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.

How do I trigger tasks based on another task's failure?

Check out the `Trigger Rule` section in the Concepts section of the documentation

Why are connection passwords still not encrypted in the metadata db after I installed airflow[crypto]?

- Verify that the `fernet_key` defined in `$AIRFLOW_HOME/airflow.cfg` is a valid Fernet key. It must be a base64-encoded 32-byte key. You need to restart the webserver after you update the key
- 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

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 to get 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.

We also recommend using rounded `start_date` in relation to your `schedule_interval`. This means an `@hourly` would be at `00:00` minutes:seconds, a `@daily` job at midnight, a `@monthly` job on the first of the month. You can use any sensor or a `TimeDeltaSensor` to delay the execution of tasks within that period. 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 it wasn't the case, the backfill just wouldn't start.

3.13 API Reference

3.13.1 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`.

3.13.1.1 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, re-
                                  tries=0, retry_delay=datetime.timedelta(0, 300),
                                  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, queue='default', pool=None, sla=None,
                                  execution_timeout=None, on_failure_callback=None,
                                  on_success_callback=None, on_retry_callback=None, trig-
                                  ger_rule=u'all_success', *args, **kwargs)
```

Abstract base class for all operators. Since operators create objects that become node 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 task should perform or trigger certain tasks synchronously (wait for completion). Example of operators could be an operator the 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.

Note that this class is derived from SQLAlchemy's Base class, which allows us to push metadata regarding tasks to the database. Deriving this classes needs to implement the polymorphic specificities documented in SQLAlchemy. This should become clear while reading the code for other operators.

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
- **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.
- **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` excepts that it is executed when retries occur.
- **on_success_callback** (*callable*) – much like the `on_failure_callback` excepts 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`

3.13.1.2 BaseSensorOperator

All sensors are derived from `BaseSensorOperator`. All sensors inherit the `timeout` and `poke_interval` on top of the `BaseOperator` attributes.

```
class airflow.operators.sensors.BaseSensorOperator (poke_interval=60, timeout=604800,
                                                    soft_fail=False, *args, **kwargs)
```

Sensor operators are derived from this class and 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.

3.13.1.3 Operator API

```
class airflow.operators.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.
- **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.BranchPythonOperator (python_callable, op_args=None,
                                              op_kwargs=None, provide_context=False,
                                              templates_dict=None, templates_exts=None,
                                              *args, **kwargs)
```

Bases: [python_operator.PythonOperator](#)

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 propagated 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.TriggerDagRunOperator (trigger_dag_id, python_callable, *args,
                                              **kwargs)
```

Bases: [airflow.models.BaseOperator](#)

Triggers a DAG run for a specified `dag_id` if a criteria is met

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):`

class `airflow.operators.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.EmailOperator` (*to, subject, html_content, files=None, *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
- **subject** (*string*) – subject line for the email (templated)
- **html_content** (*string*) – content of the email (templated), html markup is allowed
- **files** (*list*) – file names to attach in email

class `airflow.operators.ExternalTaskSensor` (*external_dag_id, external_task_id, allowed_states=None, execution_delta=None, *args, **kwargs*)

Bases: `sensors.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)`

class `airflow.operators.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
- **destination_table** (*str*) – target table
- **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

class airflow.operators.**HdfsSensor** (*filepath, hdfs_conn_id='hdfs_default', *args, **kwargs*)

Bases: *sensors.BaseSensorOperator*

Waits for a file or folder to land in HDFS

class airflow.operators.**HivePartitionSensor** (*table, partition="ds='{ ds }'", metastore_conn_id='metastore_default', schema='default', poke_interval=180, *args, **kwargs*)

Bases: *sensors.BaseSensorOperator*

Waits for a partition to show up in Hive

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 *> <* sings as in “*ds>=2015-01-01*”
- **metastore_conn_id** (*str*) – reference to the metastore thrift service connection id

class airflow.operators.**SimpleHttpOperator** (*endpoint, method='POST', data=None, headers=None, response_check=None, extra_options=None, 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
- **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.
- **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.**HttpSensor** (*endpoint, http_conn_id='http_default', params=None, headers=None, response_check=None, extra_options=None, *args, **kwargs*)

Bases: *sensors.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
- **endpoint** (*string*) – The relative part of the full url
- **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.)

```
class airflow.operators.MetastorePartitionSensor (table, partition_name, schema='default',
                                                  mysql_conn_id='metastore_mysql',
                                                  *args, **kwargs)
```

Bases: `sensors.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

```
class airflow.operators.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
- **templates_exts** – a list of file extensions to resolve while processing templated fields, for examples `['.sql', '.hql']`

```
class airflow.operators.S3KeySensor(bucket_key, bucket_name=None, wildcard_match=False,
                                   s3_conn_id='s3_default', *args, **kwargs)
```

Bases: `sensors.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
- **s3_conn_id** (*str*) – a reference to the s3 connection

```
class airflow.operators.ShortCircuitOperator(python_callable, op_args=None,
                                             op_kwargs=None, provide_context=False,
                                             templates_dict=None, templates_exts=None,
                                             *args, **kwargs)
```

Bases: `python_operator.PythonOperator`

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.SqlSensor(conn_id, sql, *args, **kwargs)
```

Bases: `sensors.BaseSensorOperator`

Runs a sql statement until a criteria is met. It will keep trying until 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.

```
class airflow.operators.TimeSensor(target_time, *args, **kwargs)
```

Bases: `sensors.BaseSensorOperator`

Waits until the specified time of the day.

Parameters **target_time** (*datetime.time*) – time after which the job succeeds

```
class airflow.operators.WebHdfsSensor (filepath, webhdfs_conn_id='webhdfs_default', *args,
                                     **kwargs)
```

Bases: `sensors.BaseSensorOperator`

Waits for a file or folder to land in HDFS

3.13.1.4 Community-contributed Operators

```
class airflow.contrib.operators.SSHExecuteOperator (ssh_hook, bash_command,
                                                    xcom_push=False, env=None,
                                                    *args, **kwargs)
```

Bases: `airflow.models.BaseOperator`

Execute a Bash script, command or set of commands at remote host.

Parameters

- **ssh_hook** – A SSHHook that indicates the remote host you want to run the script
- **ssh_hook** – SSHHook
- **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.

3.13.2 Macros

Here's a list of variables and macros that can be used in templates

3.13.2.1 Default Variables

The Airflow engine passes a few variables by default that are accessible in all templates

Variable	Description
<code>{{ ds }}</code>	the execution date as YYYY-MM-DD
<code>{{ ds_nodash }}</code>	the execution date as YYYYMMDD
<code>{{ yesterday_ds }}</code>	yesterday's date as YYYY-MM-DD
<code>{{ yesterday_ds_nodash }}</code>	yesterday's date as YYYYMMDD
<code>{{ tomorrow_ds }}</code>	tomorrow's date as YYYY-MM-DD
<code>{{ tomorrow_ds_nodash }}</code>	tomorrow's date as YYYYMMDD
<code>{{ ts }}</code>	same as <code>execution_date.isoformat()</code>
<code>{{ ts_nodash }}</code>	same as <code>ts</code> without <code>-</code> and <code>:</code>
<code>{{ execution_date }}</code>	the <code>execution_date</code> , (<code>datetime.datetime</code>)
<code>{{ dag }}</code>	the DAG object
<code>{{ task }}</code>	the Task object
<code>{{ macros }}</code>	a reference to the macros package, described below
<code>{{ task_instance }}</code>	the <code>task_instance</code> object
<code>{{ end_date }}</code>	same as <code>{{ ds }}</code>
<code>{{ latest_date }}</code>	same as <code>{{ ds }}</code>
<code>{{ ti }}</code>	same as <code>{{ task_instance }}</code>
<code>{{ params }}</code>	a reference to the user-defined params dictionary
<code>{{ task_instance_key_str }}</code>	a unique, human-readable key to the task instance formatted as <code>{dag_id}_{task_id}_{ds}</code>
<code>conf</code>	the full configuration object located at <code>airflow.configuration.conf</code> which represents the content of your <code>airflow.cfg</code>
<code>run_id</code>	the <code>run_id</code> of the current DAG run
<code>dag_run</code>	a reference to the <code>DagRun</code> object
<code>test_mode</code>	whether the task instance was called using the CLI's test subcommand

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.

3.13.2.2 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
<code>macros.datetime</code>	The standard lib's <code>datetime.datetime</code>
<code>macros.timedelta</code>	The standard lib's <code>datetime.timedelta</code>
<code>macros.dateutil</code>	A reference to the <code>dateutil</code> package
<code>macros.time</code>	The standard lib's <code>time</code>
<code>macros.uuid</code>	The standard lib's <code>uuid</code>
<code>macros.random</code>	The standard lib's <code>random</code>

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.integrate_plugins()`

Integrate plugins to the context

`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 timestamp %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=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
- **hive_conn_id** (*string*) – The hive connection you are interested in. If your default is set you don’t need to use this parameter.
- **filter** (*string*) – filter on a subset of partition as in *sub_part=’specific_value’*
- **field** – 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'
```

3.13.3 Models

Models are built on top of the SQLAlchemy ORM Base class, and instances are persisted in the database.

```
class airflow.models.DAG(dag_id,      schedule_interval=datetime.timedelta(1),      start_date=None,
                        end_date=None,      full_filepath=None,      template_searchpath=None,
                        user_defined_macros=None,      default_args=None,      concurrency=16,
                        max_active_runs=16,      dagrun_timeout=None,      sla_miss_callback=None,
                        params=None)
```

Bases: `airflow.utils.logging.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
- **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 strings*) – 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.
- **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.

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 **task** (*list of tasks*) – a list of tasks you want to add

cli ()

Exposes a CLI specific to this DAG

concurrency_reached

Returns a boolean indicating whether the concurrency limit for this DAG has been reached

crawl_for_tasks (*objects*)

Typically called at the end of a script by passing `globals()` as a parameter. This allows to not explicitly add every single task to the dag explicitly.

filepath

File location of where the dag object is instantiated

folder

Folder location of where the dag object is instantiated

get_active_runs ()

Maintains and returns the currently active runs as a list of dates.

A run is considered a SUCCESS if all of its root tasks either succeeded or were skipped.

A run is considered a FAILURE if any of its root tasks failed OR if it is deadlocked, meaning no tasks can run.

get_template_env ()

Returns a `jinja2` Environment while taking into account the DAGs `template_searchpath` and `user_defined_macros`

is_paused

Returns a boolean indicating whether this DAG is paused

latest_execution_date

Returns the latest date for which at least one task instance exists

run (*start_date=None, end_date=None, mark_success=False, include_adhoc=False, local=False, executor=None, do_not_pickle=False, ignore_dependencies=False, ignore_first_depends_on_past=False, pool=None*)
Runs the DAG.

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

tree_view ()

Shows an ascii tree representation of the DAG

```
class airflow.models.BaseOperator(task_id, owner='Airflow', email=None,
                                  email_on_retry=True, email_on_failure=True, re-
                                  tries=0, retry_delay=datetime.timedelta(0, 300),
                                  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, queue='default', pool=None, sla=None,
                                  execution_timeout=None, on_failure_callback=None,
                                  on_success_callback=None, on_retry_callback=None, trig-
                                  ger_rule=u'all_success', *args, **kwargs)
```

Bases: `future.types.newobject.newobject`

Abstract base class for all operators. Since operators create objects that become node 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 task should perform or trigger certain tasks synchronously (wait for completion). Example of operators could be an operator the 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.

Note that this class is derived from SQLAlchemy's Base class, which allows us to push metadata regarding tasks to the database. Deriving this classes needs to implement the polymorphic specificities documented in SQLAlchemy. This should become clear while reading the code for other operators.

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
- **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.
- **`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` excepts that it is executed when retries occur.
- **`on_success_callback`** (*callable*) – much like the `on_failure_callback` excepts 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`

`clear` (*start_date=None, end_date=None, upstream=False, downstream=False*)

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

detect_downstream_cycle (*task=None*)

When invoked, this routine will raise an exception if a cycle is detected downstream from self. It is invoked when tasks are added to the DAG to detect cycles.

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_relatives (*upstream=False*)

Get the direct relatives to the current task, upstream or downstream.

get_flat_relatives (*upstream=False, l=None*)

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*)

This is triggered right after `self.execute`, it's mostly a hook for people deriving operators.

pre_execute (*context*)

This is triggered right before `self.execute`, it's mostly a hook for people deriving operators.

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_dependencies=False, ignore_first_depends_on_past=False, force=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 task to be directly downstream from the current task.

set_upstream (*task_or_task_list*)

Set a task, or a task task to be directly upstream from the current task.

upstream_list

@property: list of tasks directly upstream

xcom_pull (*context, task_ids, 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.**TaskInstance** (*task, execution_date, state=None*)

Bases: sqlalchemy.ext.declarative.api.Base

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 (**args, **kwargs*)

Returns a boolean on whether the upstream tasks are in a SUCCESS state and considers depends_on_past and the previous run's state.

Parameters

- **flag_upstream_failed** (*boolean*) – This is a hack to generate the upstream_failed state creation while checking to see whether the task instance is runnable. It was the shortest path to add the feature
- **ignore_depends_on_past** (*boolean*) – if True, ignores depends_on_past dependencies. Defaults to False.
- **verbose** (*boolean*) – verbose provides more logging in the case where the task instance is evaluated as a check right before being executed. In the case of the scheduler evaluating the dependencies, this logging would be way too verbose.

are_dependents_done (**args, **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 (**args, **kwargs*)

Clears all XCom data from the database for the task instance

command (*mark_success=False, ignore_dependencies=False, ignore_depends_on_past=False, force=False, local=False, pickle_id=None, raw=False, job_id=None, pool=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 (**args, **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 (**args, **kwargs*)

Forces the task instance's state to FAILED in the database.

evaluate_trigger_rule (**args, **kwargs*)

Returns a boolean on whether the current task can be scheduled for execution based on its trigger_rule.

Parameters

- **flag_upstream_failed** (*boolean*) – This is a hack to generate the upstream_failed state creation while checking to see whether the task instance is runnable. It was the shortest path to add the feature
- **successes** (*boolean*) – Number of successful upstream tasks
- **skipped** (*boolean*) – Number of skipped upstream tasks
- **failed** (*boolean*) – Number of failed upstream tasks
- **upstream_failed** (*boolean*) – Number of upstream_failed upstream tasks
- **done** (*boolean*) – Number of completed upstream tasks

is_premature ()

Returns whether a task is in UP_FOR_RETRY state and its retry interval has elapsed.

is_queueable (*include_queued=False*, *ignore_depends_on_past=False*,
flag_upstream_failed=False)

Returns a boolean on whether the task instance has met all dependencies and is ready to run. It considers the task's state, the state of its dependencies, depends_on_past and makes sure the execution isn't in the future. It doesn't take into account whether the pool has a slot for it to run.

Parameters

- **include_queued** (*boolean*) – If True, tasks that have already been queued are included. Defaults to False.
- **ignore_depends_on_past** (*boolean*) – if True, ignores depends_on_past dependencies. Defaults to False.
- **flag_upstream_failed** (*boolean*) – This is a hack to generate the upstream_failed state creation while checking to see whether the task instance is runnable. It was the shortest path to add the feature

is_runnable (*include_queued=False*, *ignore_depends_on_past=False*, *flag_upstream_failed=False*)

Returns whether a task is ready to run AND there's room in the queue.

Parameters

- **include_queued** (*boolean*) – If True, tasks that are already QUEUED are considered "runnable". Defaults to False.
- **ignore_depends_on_past** (*boolean*) – if True, ignores depends_on_past dependencies. Defaults to False.

key

Returns a tuple that identifies the task instance uniquely

pool_full (**args*, ***kwargs*)

Returns a boolean as to whether the slot pool has room for this task to run

ready_for_retry ()

Checks on whether the task instance is in the right state and timeframe to be retried.

refresh_from_db (**args*, ***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.

run (**args*, ***kwargs*)

Runs the task instance.

xcom_pull (*task_ids*, *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.DagBag` (*dag_folder=None*, *executor=<airflow.executors.sequential_executor.SequentialExecutor object>*, *include_examples=True*, *sync_to_db=False*)

Bases: `airflow.utils.logging.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** (*str*) – 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
- **sync_to_db** (*bool*) – whether to sync the properties of the DAGs to the metadata DB while finding them, typically should be done by the scheduler job only

bag_dag (*dag, parent_dag, root_dag*)

Adds the DAG into the bag, recurses into sub dags.

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.

get_dag (*dag_id*)

Gets the DAG out of the dictionary, and refreshes it if expired

kill_zombies (**args, **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.**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

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

3.13.4 Hooks

class airflow.hooks.**DbApiHook** (**args, **kwargs*)

Bases: airflow.hooks.base_hook.BaseHook

Abstract base class for sql hooks.

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_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*)

A generic way to insert a set of tuples into a table, the whole set of inserts is treated as one transaction

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.

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.

class airflow.hooks.**HttpHook** (*method='POST', http_conn_id='http_default'*)

Bases: airflow.hooks.base_hook.BaseHook

Interact with HTTP servers.

get_conn (*headers*)

Returns http session for use with requests

run (*endpoint*, *data=None*, *headers=None*, *extra_options=None*)

Performs the request

run_and_check (*session*, *prepped_request*, *extra_options*)

Grabs extra options like timeout and actually runs the request, checking for the result

class `airflow.hooks.SQLiteHook` (**args*, ***kwargs*)

Bases: `airflow.hooks.dbapi_hook.DbApiHook`

Interact with SQLite.

get_conn ()

Returns a sqlite connection object

3.13.4.1 Community contributed hooks

class `airflow.contrib.hooks.GoogleCloudStorageHook` (*scope='https://www.googleapis.com/auth/devstorage.read_only'*,
google_cloud_storage_conn_id='google_cloud_storage_default',
delegate_to=None)

Bases: `airflow.contrib.hooks.gc_base_hook.GoogleCloudBaseHook`

Interact with Google Cloud Storage. Connections must be defined with an extras JSON field containing:

```
{ "project": "<google project ID>", "service_account": "<google service account email>", "key_path": "<p12 key path>"
}
```

If you have used `gcloud auth` to authenticate on the machine that's running Airflow, you can exclude the `service_account` and `key_path` parameters.

download (*bucket*, *object*, *filename=False*)

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.

get_conn ()

Returns a Google Cloud Storage service object.

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.FTPHook` (*ftp_conn_id='ftp_default'*)

Bases: `airflow.hooks.base_hook.BaseHook`

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 wasnt 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

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** – 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.SSHHook(*conn_id='ssh_default'*)

Bases: airflow.hooks.base_hook.BaseHook

Light-weight remote execution library and utilities.

Using this hook (which is just a convenience wrapper for subprocess), is created to let you stream data from a remotely stored file.

As a bonus, *SSHHook* also provides a really cool feature that let's you set up ssh tunnels super easily using a python context manager (there is an example in the integration part of unittests).

Parameters

- **key_file** (*str*) – Typically the SSHHook uses the keys that are used by the user airflow is running under. This sets the behavior to use another file instead.
- **connect_timeout** (*int*) – sets the connection timeout for this connection.
- **no_host_key_check** (*bool*) – whether to check to host key. If True host keys will not be checked, but are also not stored in the current users's known_hosts file.
- **tty** (*bool*) – allocate a tty.
- **sshpass** (*bool*) – Use to non-interactively perform password authentication by using sshpass.

Popen (*cmd*, ***kwargs*)

Remote Popen

Parameters

- **cmd** – command to remotely execute
- **kwargs** – extra arguments to Popen (see subprocess.Popen)

Returns handle to subprocess

check_output (*cmd*)

Executes a remote command and returns the stdout a remote process. Simplified version of Popen when you only want the output as a string and detect any errors.

Parameters **cmd** – command to remotely execute

Returns stdout

tunnel (**args*, ***kws*)

Creates a tunnel between two hosts. Like ssh -L <LOCAL_PORT>:host:<REMOTE_PORT>. Remember to close() the returned “tunnel” object in order to clean up after yourself when you are done with the tunnel.

Parameters

- **local_port** (*int*) –
- **remote_port** (*int*) –
- **remote_host** (*str*) –

Returns

3.13.5 Executors

Executors are the mechanism by which task instances get run.

class airflow.executors.**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.

class airflow.executors.**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.

3.13.5.1 Community-contributed executors

a

- `airflow.contrib.executors`, [73](#)
- `airflow.contrib.hooks`, [70](#)
- `airflow.contrib.operators`, [57](#)
- `airflow.executors`, [72](#)
- `airflow.hooks`, [68](#)
- `airflow.macros`, [59](#)
- `airflow.macros.hive`, [59](#)
- `airflow.models`, [60](#)
- `airflow.operators`, [52](#)

A

[add_task\(\)](#) (airflow.models.DAG method), 61
[add_tasks\(\)](#) (airflow.models.DAG method), 61
[airflow.contrib.executors](#) (module), 73
[airflow.contrib.hooks](#) (module), 70
[airflow.contrib.operators](#) (module), 57
[airflow.executors](#) (module), 72
[airflow.hooks](#) (module), 68
[airflow.macros](#) (module), 59
[airflow.macros.hive](#) (module), 59
[airflow.models](#) (module), 60
[airflow.operators](#) (module), 52
[are_dependencies_met\(\)](#) (airflow.models.TaskInstance method), 65
[are_dependents_done\(\)](#) (airflow.models.TaskInstance method), 65

B

[bag_dag\(\)](#) (airflow.models.DagBag method), 67
[BaseOperator](#) (class in airflow.models), 50, 62
[BaseSensorOperator](#) (class in airflow.operators.sensors), 51
[BashOperator](#) (class in airflow.operators), 52
[BranchPythonOperator](#) (class in airflow.operators), 52
[bulk_load\(\)](#) (airflow.hooks.DbApiHook method), 68

C

[check_output\(\)](#) (airflow.contrib.hooks.SSHHook method), 72
[clear\(\)](#) (airflow.models.BaseOperator method), 63
[clear_xcom_data\(\)](#) (airflow.models.TaskInstance method), 65
[cli\(\)](#) (airflow.models.DAG method), 61
[close_conn\(\)](#) (airflow.contrib.hooks.FTPHook method), 70
[closest_ds_partition\(\)](#) (in module airflow.macros.hive), 59
[collect_dags\(\)](#) (airflow.models.DagBag method), 68
[command\(\)](#) (airflow.models.TaskInstance method), 65
[concurrency_reached](#) (airflow.models.DAG attribute), 61

[Connection](#) (class in airflow.models), 68
[crawl_for_tasks\(\)](#) (airflow.models.DAG method), 61
[create_directory\(\)](#) (airflow.contrib.hooks.FTPHook method), 71
[current_state\(\)](#) (airflow.models.TaskInstance method), 65

D

[dag](#) (airflow.models.BaseOperator attribute), 63
[DAG](#) (class in airflow.models), 60
[DagBag](#) (class in airflow.models), 67
[DbApiHook](#) (class in airflow.hooks), 68
[delete_directory\(\)](#) (airflow.contrib.hooks.FTPHook method), 71
[delete_file\(\)](#) (airflow.contrib.hooks.FTPHook method), 71
[describe_directory\(\)](#) (airflow.contrib.hooks.FTPHook method), 71
[detect_downstream_cycle\(\)](#) (airflow.models.BaseOperator method), 64
[download\(\)](#) (airflow.contrib.hooks.GoogleCloudStorageHook method), 70
[downstream_list](#) (airflow.models.BaseOperator attribute), 64
[ds_add\(\)](#) (in module airflow.macros), 59
[ds_format\(\)](#) (in module airflow.macros), 59
[DummyOperator](#) (class in airflow.operators), 53

E

[EmailOperator](#) (class in airflow.operators), 53
[error\(\)](#) (airflow.models.TaskInstance method), 65
[evaluate_trigger_rule\(\)](#) (airflow.models.TaskInstance method), 65
[execute\(\)](#) (airflow.models.BaseOperator method), 64
[execute\(\)](#) (airflow.operators.BashOperator method), 52
[ExternalTaskSensor](#) (class in airflow.operators), 53
[extra_dejson](#) (airflow.models.Connection attribute), 68

F

[filepath](#) (airflow.models.DAG attribute), 61
[folder](#) (airflow.models.DAG attribute), 61

FTPHook (class in airflow.contrib.hooks), 70

G

GenericTransfer (class in airflow.operators), 53

get_active_runs() (airflow.models.DAG method), 61

get_conn() (airflow.contrib.hooks.FTPHook method), 71

get_conn() (airflow.contrib.hooks.GoogleCloudStorageHook method), 70

get_conn() (airflow.hooks.DbApiHook method), 68

get_conn() (airflow.hooks.HttpHook method), 69

get_conn() (airflow.hooks.SQLiteHook method), 70

get_cursor() (airflow.hooks.DbApiHook method), 68

get_dag() (airflow.models.DagBag method), 68

get_direct_relatives() (airflow.models.BaseOperator method), 64

get_first() (airflow.hooks.DbApiHook method), 68

get_flat_relatives() (airflow.models.BaseOperator method), 64

get_pandas_df() (airflow.hooks.DbApiHook method), 69

get_records() (airflow.hooks.DbApiHook method), 69

get_task_instances() (airflow.models.BaseOperator method), 64

get_template_env() (airflow.models.DAG method), 61

GoogleCloudStorageHook (class in airflow.contrib.hooks), 70

H

has_dag() (airflow.models.BaseOperator method), 64

HdfsSensor (class in airflow.operators), 54

HivePartitionSensor (class in airflow.operators), 54

HttpHook (class in airflow.hooks), 69

HttpSensor (class in airflow.operators), 54

I

insert_rows() (airflow.hooks.DbApiHook method), 69

integrate_plugins() (in module airflow.macros), 59

is_paused (airflow.models.DAG attribute), 61

is_premature() (airflow.models.TaskInstance method), 66

is_queueable() (airflow.models.TaskInstance method), 66

is_runnable() (airflow.models.TaskInstance method), 66

K

key (airflow.models.TaskInstance attribute), 66

kill_zombies() (airflow.models.DagBag method), 68

L

latest_execution_date (airflow.models.DAG attribute), 61

list_directory() (airflow.contrib.hooks.FTPHook method), 71

LocalExecutor (class in airflow.executors), 72

M

max_partition() (in module airflow.macros.hive), 59

MetastorePartitionSensor (class in airflow.operators), 55

O

on_kill() (airflow.models.BaseOperator method), 64

P

pool_full() (airflow.models.TaskInstance method), 66

Popen() (airflow.contrib.hooks.SSHHook method), 72

post_execute() (airflow.models.BaseOperator method), 64

pre_execute() (airflow.models.BaseOperator method), 64

prepare_template() (airflow.models.BaseOperator method), 64

process_file() (airflow.models.DagBag method), 68

PythonOperator (class in airflow.operators), 55

R

random() (in module airflow.macros), 59

ready_for_retry() (airflow.models.TaskInstance method), 66

refresh_from_db() (airflow.models.TaskInstance method), 66

render_template() (airflow.models.BaseOperator method), 64

render_template_from_field() (airflow.models.BaseOperator method), 64

retrieve_file() (airflow.contrib.hooks.FTPHook method), 71

run() (airflow.hooks.DbApiHook method), 69

run() (airflow.hooks.HttpHook method), 69

run() (airflow.models.BaseOperator method), 64

run() (airflow.models.DAG method), 61

run() (airflow.models.TaskInstance method), 66

run_and_check() (airflow.hooks.HttpHook method), 70

S

S3KeySensor (class in airflow.operators), 56

schedule_interval (airflow.models.BaseOperator attribute), 64

SequentialExecutor (class in airflow.executors), 72

set_dependency() (airflow.models.DAG method), 61

set_downstream() (airflow.models.BaseOperator method), 64

set_upstream() (airflow.models.BaseOperator method), 64

ShortCircuitOperator (class in airflow.operators), 56

SimpleHttpOperator (class in airflow.operators), 54

size() (airflow.models.DagBag method), 68

SQLiteHook (class in airflow.hooks), 70

SqlSensor (class in airflow.operators), 56

SSHExecuteOperator (class in airflow.contrib.operators), 57

SSHHook (class in airflow.contrib.hooks), 71

store_file() (airflow.contrib.hooks.FTPHook method), 71

`sub_dag()` (airflow.models.DAG method), [62](#)
`subdags` (airflow.models.DAG attribute), [62](#)

T

`TaskInstance` (class in airflow.models), [65](#)
`TimeSensor` (class in airflow.operators), [56](#)
`tree_view()` (airflow.models.DAG method), [62](#)
`TriggerDagRunOperator` (class in airflow.operators), [52](#)
`tunnel()` (airflow.contrib.hooks.SSHHook method), [72](#)

U

`upload()` (airflow.contrib.hooks.GoogleCloudStorageHook method), [70](#)
`upstream_list` (airflow.models.BaseOperator attribute), [64](#)

W

`WebHdfsSensor` (class in airflow.operators), [56](#)

X

`xcom_pull()` (airflow.models.BaseOperator method), [65](#)
`xcom_pull()` (airflow.models.TaskInstance method), [66](#)
`xcom_push()` (airflow.models.BaseOperator method), [65](#)
`xcom_push()` (airflow.models.TaskInstance method), [67](#)