****

**Airflow Installation and Configuration procedure.**

**Technology Solutions**

**Document Version**

**Version History**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Date** | **Author** | **Role** | **Comments** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Approvals**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Date Approved** | **Author** | **Role** | **Comments** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Reviewed by**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date** | **Name** | **Role** | **E Mail** | **Comments** |
|  |  |  |  |  |
|  |  |  |  |  |

# References/Associated Documentation

|  |  |  |
| --- | --- | --- |
| **Document Name** | **Version #** | **Source** |
|  |  |  |

# Table of Contents

[References/Associated Documentation 2](#_Toc37935753)

[Table of Contents 3](#_Toc37935754)

[1. Objective 4](#_Toc37935755)

[2. Assumptions 4](#_Toc37935756)

[3. Prerequisites 4](#_Toc37935757)

[3.1 Operating System 4](#_Toc37935761)

[3.2 Hardware Requirement 4](#_Toc37935762)

[4. Installation 5](#_Toc37935763)

[5. Configurations 5](#_Toc37935764)

[6. Airflow additional packages installation 7](#_Toc37935765)

[7. Initiating Airflow Database 8](#_Toc37935766)

[8. Instantiating DAG 8](#_Toc37935767)

[9. Creating Task 8](#_Toc37935768)

[10. Adding DAG and Tasks documentation 9](#_Toc37935769)

[11. Basic DAG 11](#_Toc37935770)

[12. Testing 14](#_Toc37935771)

[13. DAG View 16](#_Toc37935772)

[Tree View 17](#_Toc37935773)

[Graph View 17](#_Toc37935774)

[Variable View 18](#_Toc37935775)

[Gantt Chart 19](#_Toc37935776)

[Code View 21](#_Toc37935777)

[Task Instance Context Menu 21](#_Toc37935778)

# Objective

[Provide Detail Objective for the Document]

**Example:**

The Objective of the document is to provide Airflow Installation details and Configuration of instances.

# Assumptions

[Provide the Assumptions for Software & Airflow Installation]

# Prerequisites

The Prerequisites for Hadoop Installation are as follows.



## Operating System

[Provide Operating System Details with the latest patch sets]

**Example**:

|  |  |
| --- | --- |
| Operation System | Version |
| Linux | X64 |
| Windows | X64 |

## Hardware Requirement

[Enter the Server Details CPU, Memory, SWAP Space]

**Example:**

**Hardware**

|  |  |
| --- | --- |
| **Components** | **Value** |
| CPU/Cores/Nodes | 3 |
| RAM (Memory in GB) |  |
| Disk Space |  |
| Swap Space |  |
| Kernel Bit |  |
| /tmp directory |  |

# Installation

To install Airflow you need to install Anconda.

If Anoconda is not installed in your system, then –

Go this link –

<https://www.anaconda.com/distribution/>

# Configurations

Using Anaconda , either use pip/conda prompt for the following airflow installation

# airflow needs a home, ~/airflow is the default,

# but you can lay foundation somewhere else if you prefer

# (optional)

export AIRFLOW\_HOME=~/airflow

# install from pypi using pip

pip install apache-airflow

# initialize the database

airflow initdb

# start the web server, default port is 8080

airflow webserver -p 8080

# start the scheduler

airflow scheduler

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

Upon running these commands, Airflow will create the $AIRFLOW\_HOME folder and lay an

“airflow.cfg” file with defaults that get you going fast. You can inspect the file either in $AIRFLOW\_HOME/airflow.cfg, or through the UI in the

Admin->Configuration menu. The PID file for the webserver will be stored in

$AIRFLOW\_HOME/airflow-webserver.pid or in /run/airflow/webserver.pid if started by systemd.

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

pip install apache-airflow

# Airflow additional packages installation

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

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

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

# Initiating Airflow Database

Airflow requires a database to be initiated before you can run tasksAfter configuration, you will need to initialize the database before you can run tasks

airflow initdb

# Instantiating 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

Sample DAG

dag = DAG(

'tutorial',

default\_args=default\_args,

description='A simple tutorial DAG',

schedule\_interval=timedelta(days=1),

)

# Creating Task

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',

depends\_on\_past=False,

bash\_command='sleep 5',

retries=3,

dag=dag,

)

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:

* Explicitly passed arguments
* Values that exist in the default\_args dictionary
* 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.

# Adding DAG and Tasks documentation

add documentation for DAG or each single task. DAG documentation only support markdown so far and task documentation support plain text, markdown, reStructuredText, json, yaml

airflow/example\_dags/tutorial.py

dag.doc\_md = \_\_doc\_\_

t1.doc\_md = """\

#### Task Documentation

You can document your task using the attributes `doc\_md` (markdown),

`doc` (plain text), `doc\_rst`, `doc\_json`, `doc\_yaml` which gets

rendered in the UI's Task Instance Details page.

![img](http://montcs.bloomu.edu/~bobmon/Semesters/2012-01/491/import%20soul.png)

"""

Setting up Dependencies

We have tasks t1, t2 and t3 that do not depend on each other. Here’s a few ways you can define dependencies between them:

t1.set\_downstream(t2)

# This means that t2 will depend on t1

# running successfully to run.

# It is equivalent to:

t2.set\_upstream(t1)

# The bit shift operator can also be

# used to chain operations:

t1 >> t2

# And the upstream dependency with the

# bit shift operator:

t2 << t1

# Chaining multiple dependencies becomes

# concise with the bit shift operator:

t1 >> t2 >> t3

# A list of tasks can also be set as

# dependencies. These operations

# all have the same effect:

t1.set\_downstream([t2, t3])

t1 >> [t2, t3]

[t2, t3] << t1

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.

# Basic DAG

from datetime import timedelta

# The DAG object; we'll need this to instantiate a DAG

from airflow import DAG

# Operators; we need this to operate!

from airflow.operators.bash\_operator import BashOperator

from airflow.utils.dates import days\_ago

# These args will get passed on to each operator

# You can override them on a per-task basis during operator initialization

default\_args = {

'owner': 'airflow',

'depends\_on\_past': False,

'start\_date': days\_ago(2),

'email': ['airflow@example.com'],

'email\_on\_failure': False,

'email\_on\_retry': False,

'retries': 1,

'retry\_delay': timedelta(minutes=5),

# 'queue': 'bash\_queue',

# 'pool': 'backfill',

# 'priority\_weight': 10,

# 'end\_date': datetime(2016, 1, 1),

# 'wait\_for\_downstream': False,

# 'dag': dag,

# 'sla': timedelta(hours=2),

# 'execution\_timeout': timedelta(seconds=300),

# 'on\_failure\_callback': some\_function,

# 'on\_success\_callback': some\_other\_function,

# 'on\_retry\_callback': another\_function,

# 'sla\_miss\_callback': yet\_another\_function,

# 'trigger\_rule': 'all\_success'

}

dag = DAG(

'tutorial',

default\_args=default\_args,

description='A simple tutorial DAG',

schedule\_interval=timedelta(days=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',

depends\_on\_past=False,

bash\_command='sleep 5',

retries=3,

dag=dag,

)

dag.doc\_md = \_\_doc\_\_

t1.doc\_md = """\

Task Documentation

You can document your task using the attributes `doc\_md` (markdown),

`doc` (plain text), `doc\_rst`, `doc\_json`, `doc\_yaml` which gets

rendered in the UI's Task Instance Details page.

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',

depends\_on\_past=False,

bash\_command=templated\_command,

params={'my\_param': 'Parameter I passed in'},

dag=dag,

)

t1 >> [t2, t3]

# Testing

Running the Script

Time to run some tests. First, let’s make sure the pipeline is parsed successfully.

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.

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

Testing

Let’s test by running the actual task instances for a specific date. The date specified in this context is called execution\_date. This is the logical date, which simulates the scheduler running your task or dag at a specific date and time, even though it physically will run now ( or as soon as its dependencies are met).

# 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.

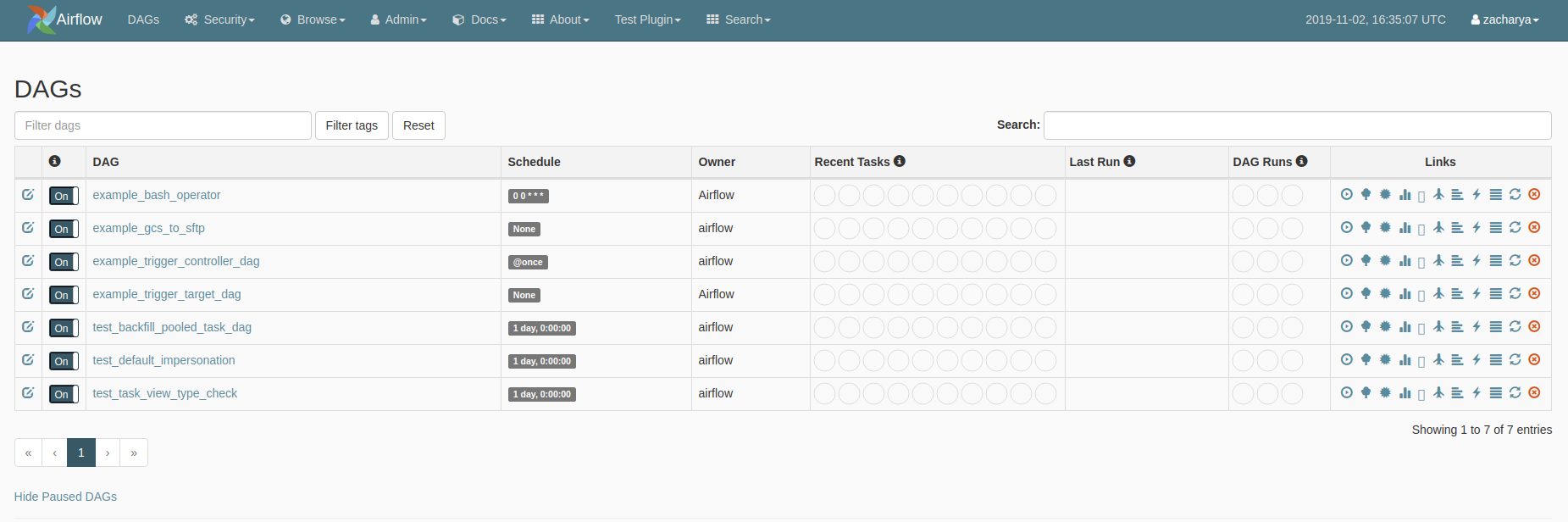
Note that the airflow test command runs task instances locally, outputs their log to stdout (on screen), doesn’t bother with dependencies, and doesn’t communicate state (running, success, failed, …) to the database. It simply allows testing a single task instance.

# DAG 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.

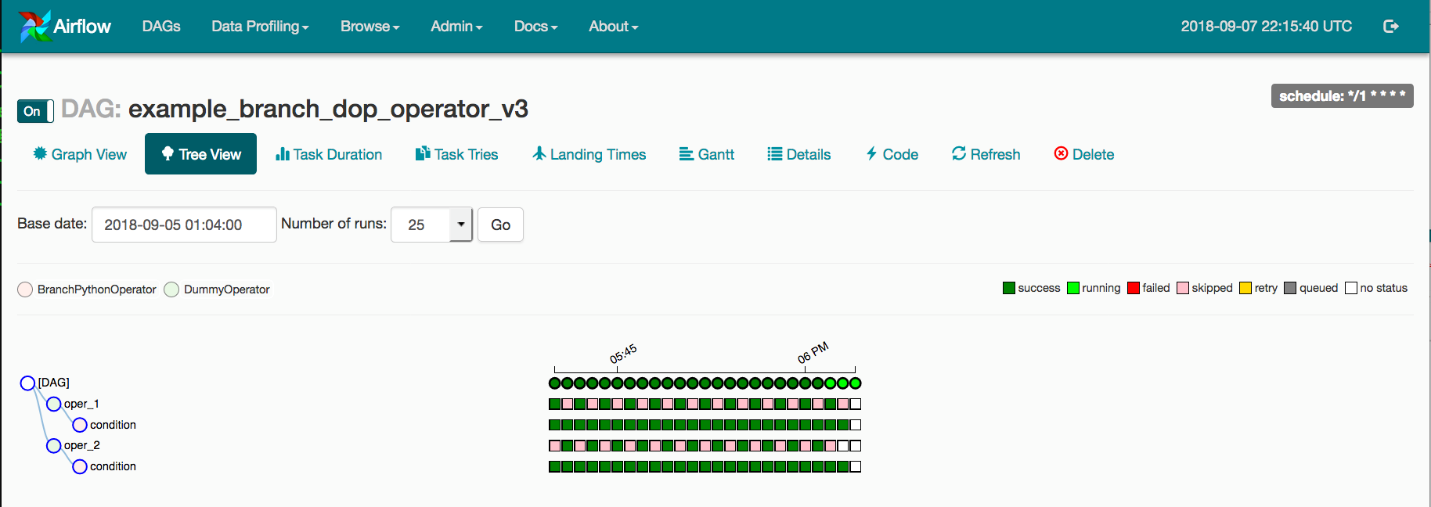
In order to filter DAGs (e.g by team), you can add tags in each dag. The filter is saved in a cookie and can be reset by the reset button. For example:

dag = DAG('dag', tags=['team1', 'sql'])



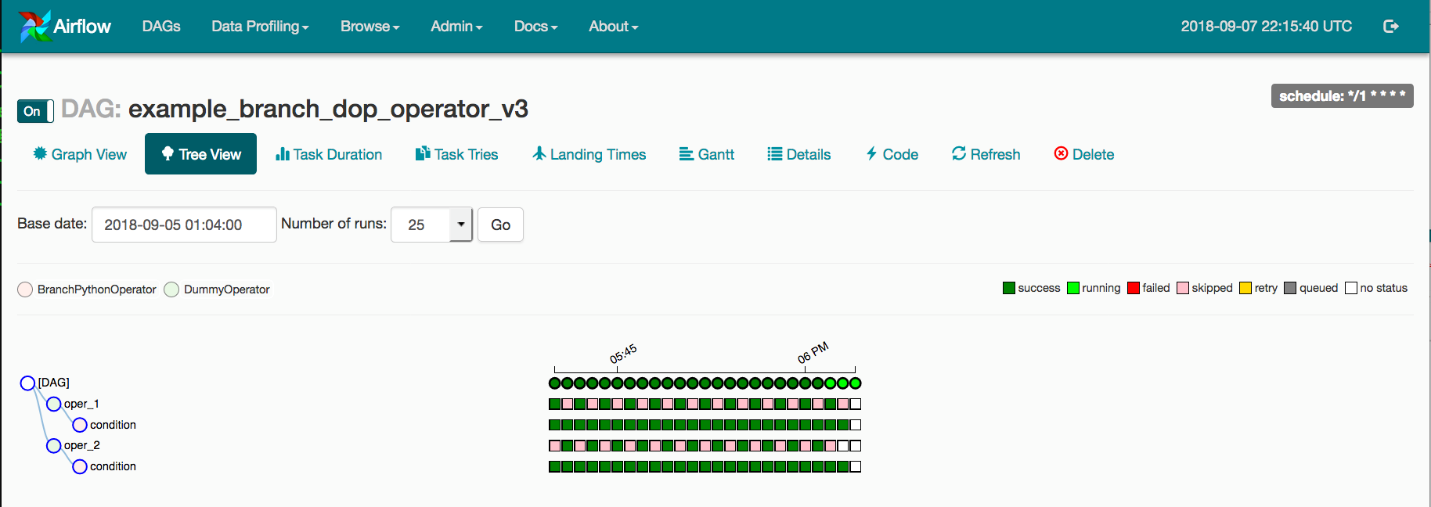
## Tree View

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



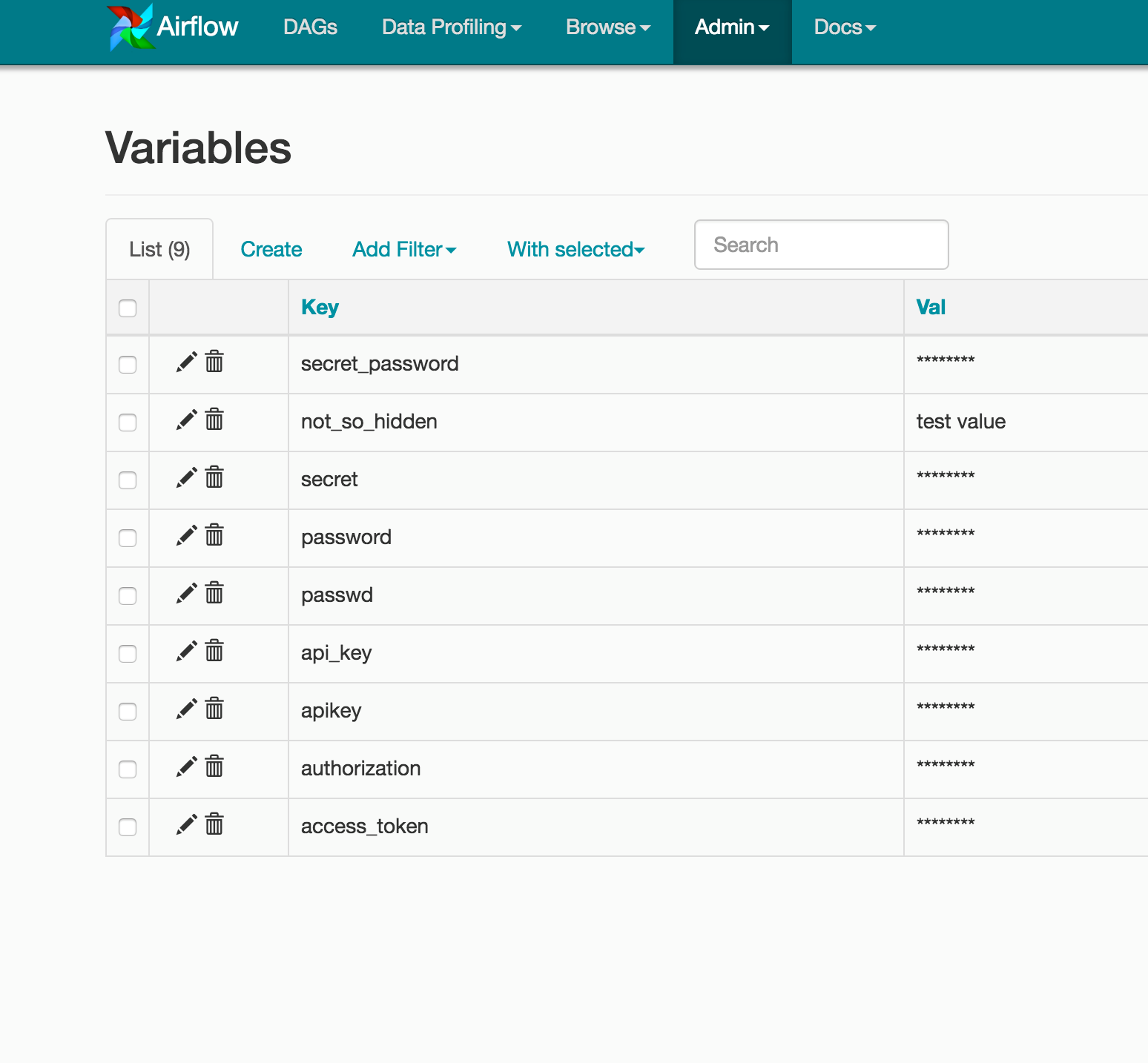
## Graph View

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



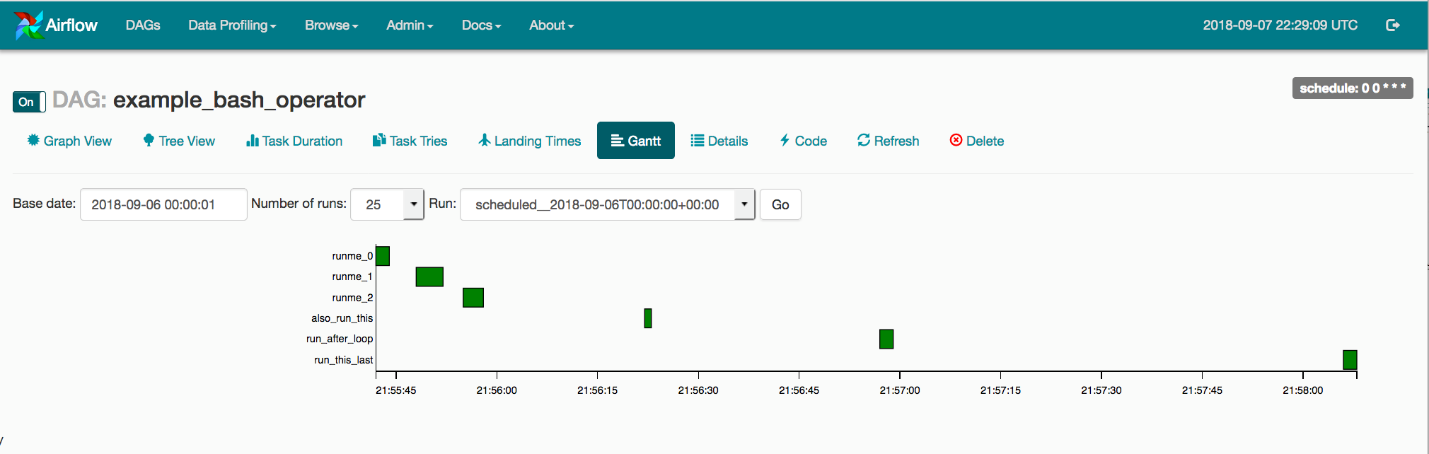
## Variable View

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



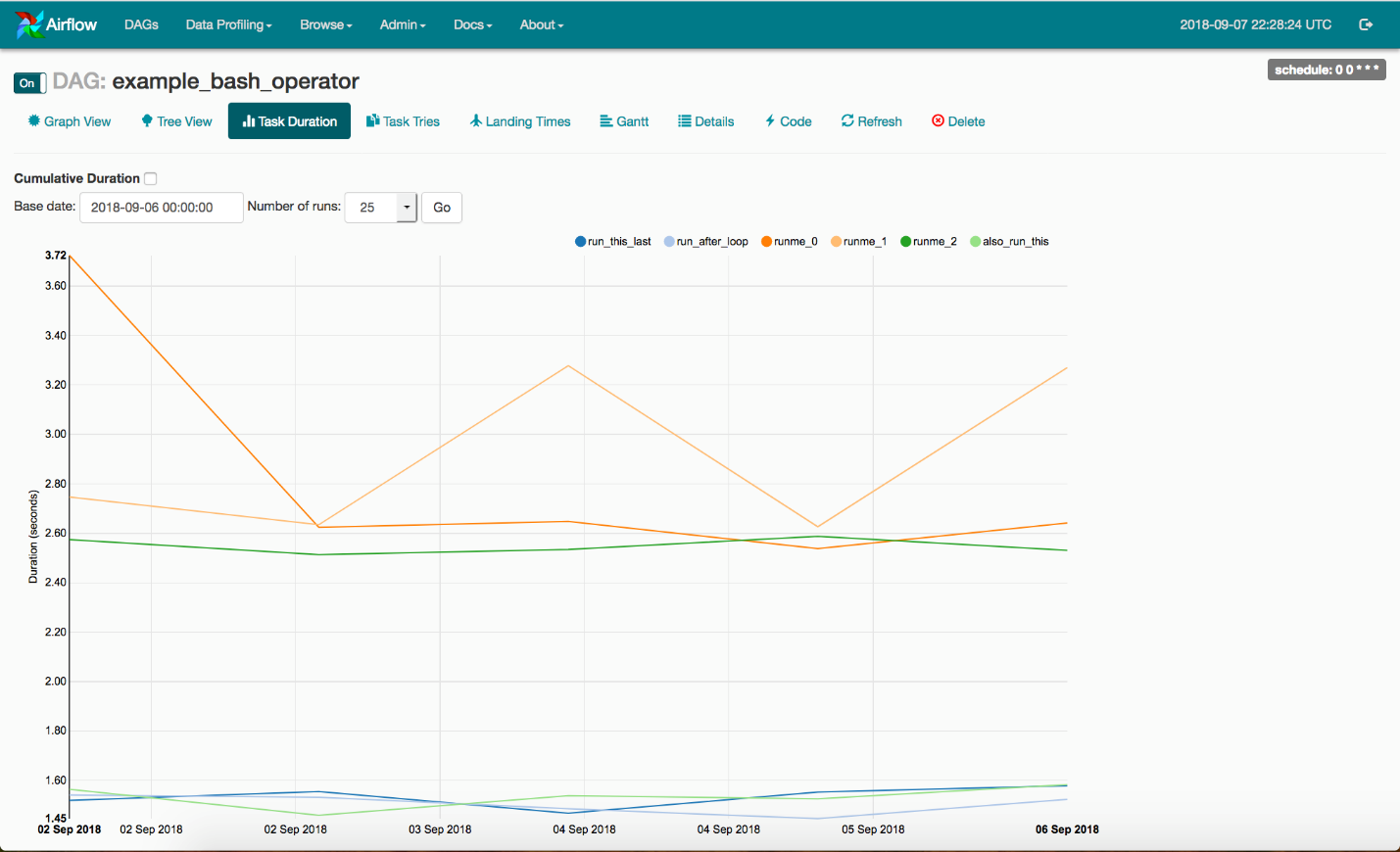
## Gantt Chart

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 run



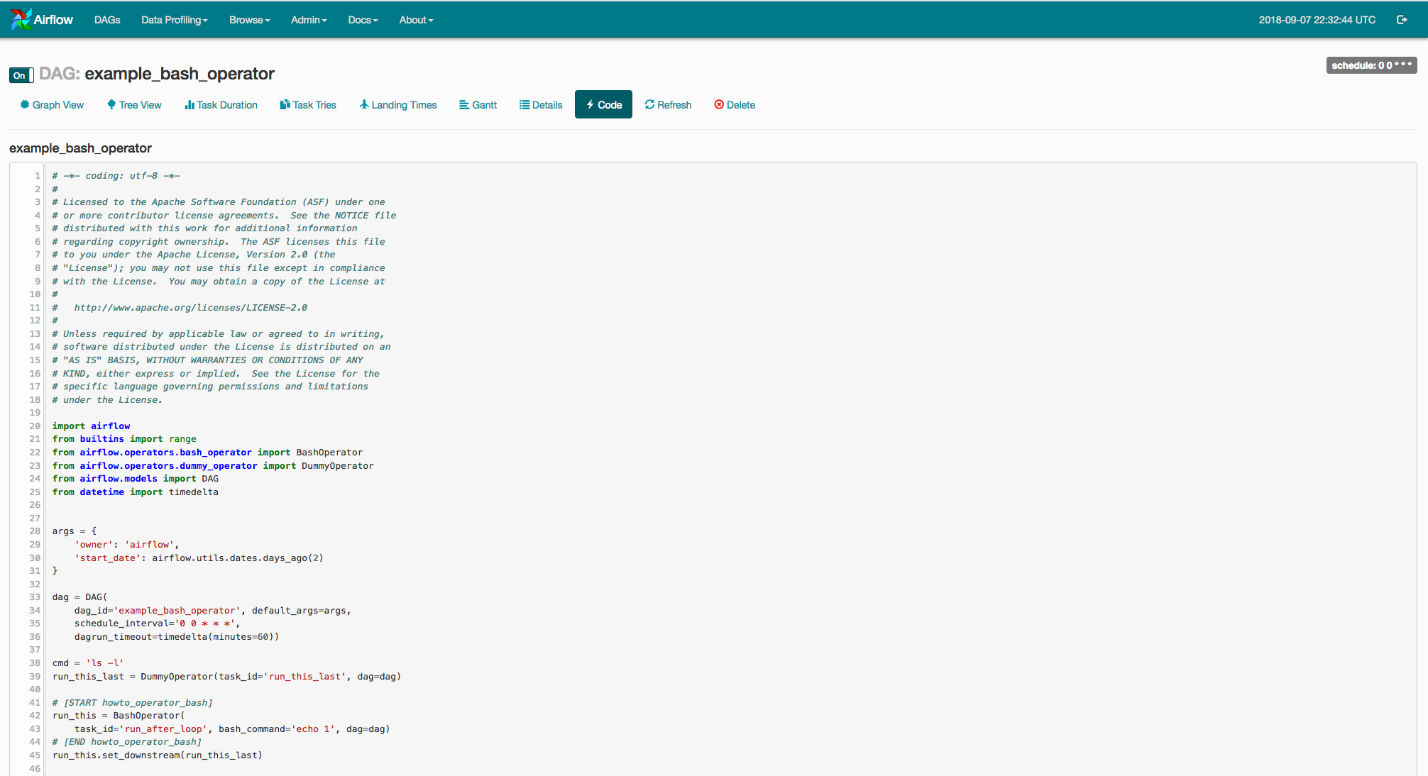
**Task Duration**

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.



## 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.



## 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.

