Apache Airflow (PoC)

2020 December

Rossano Marcos
Big Data Architect

modak

Agenda

- *Apache Airflow*
  - *Introduction*
  - *Architecture*
  - *Core Components*
  - *Demo 1 - (GCP + pyspark)*
    - *Understanding the Code behind Airflow*
  - *Demo 2 (GCP + pyspark)*
  - *Airflow 2.0*

Apache Airflow

What is Apache Airflow?

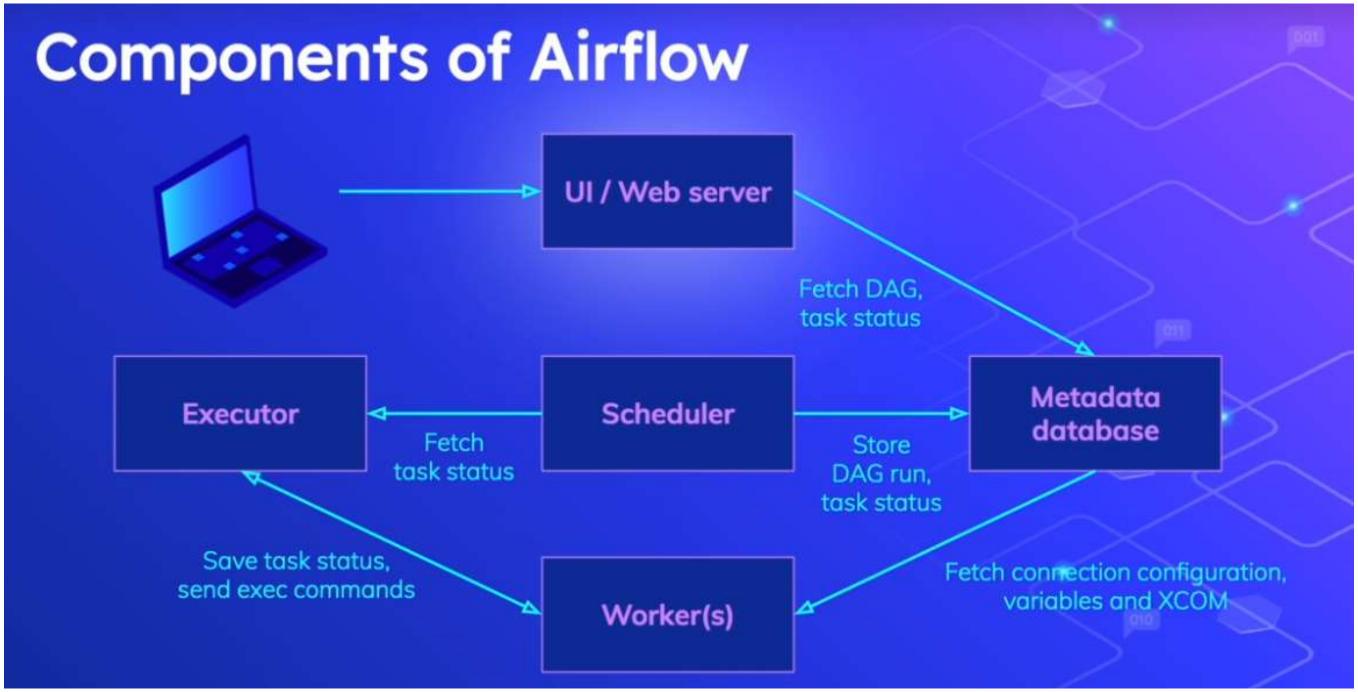
The primary use of Apache airflow is managing the workflow of a system. It is an open-source and still in the incubator stage. It was initialized in 2014 under the umbrella of Airbnb since then it got an excellent reputation with approximately 500 contributors on GitHub and 8500 stars. The main functions of Apache Airflow are to schedule workflow, monitor and author. These functions achieved with Directed Acyclic Graphs (DAG) of the tasks.

Comparing Airflow to other tools

	Airflow	Oozie	Azkaban
Owner	Apache	Apache	Linkedin
Github starts*	16.8k	565	3.2k
# of contributors*	1139	15	92
Started at	2014	2010	2011
Purpose	General purpose job scheduling	Hadoop job scheduling	Hadoop job scheduling
Flow definition	Python	Java/XML	Hadoop DSL
Scalability	Depends on executor setup	Good	Good

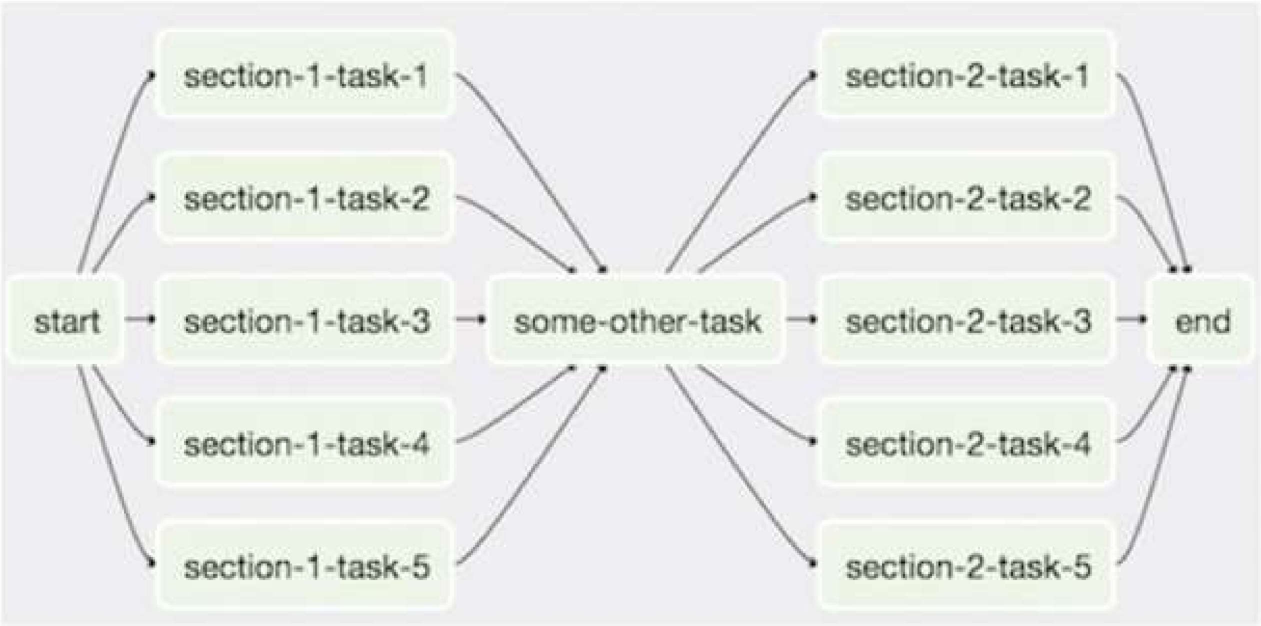
Components of Apache Airflow

- **DAG**: It is the Directed Acyclic Graph – a collection of all the tasks that you want to run which is organized and shows the relationship between different tasks. It is defined in a python script.
- **Web Server**: It is the user interface built on the Flask. It allows us to monitor the status of the DAGs and trigger them.
- **Metadata Database**: Airflow stores the status of all the tasks in a database and do all read/write operations of a workflow from here.
- **Scheduler**: As the name suggests, this component is responsible for scheduling the execution of DAGs. It retrieves and updates the status of the task in the database.


Components of Airflow
UI / Web server
Fetch DAG,
task status
Executor
Fetch
task status
Scheduler
Store
DAG run,
task status
Metadata
database
Save task status,
send exec commands
Worker(s)
Fetch connection configuration,
variables and XCOM

Airflow DAG

- A workflow as a Directed Acyclic Graph (DAG) with multiple tasks which can be executed independently
- Airflow DAGs are composed of Tasks



Demo:

- http://localhost:8080/admin/

Airflow Characteristics

- Can handle upstream/downstream dependencies gracefully (Example: upstream missing tables)
- Easy to reprocess historical jobs by date, or re-run for specific intervals
- Jobs can pass parameters to other jobs downstream
- Handle errors and failures gracefully. Automatically retry when a task fails.
- Ease of deployment of workflow changes (continuous integration)
- Integrations with a lot of infrastructure (Hive, Presto, Druid, AWS, Google cloud, etc)
- Data sensors to trigger a DAG when data arrives
- Job testing through airflow itself
- Accessibility of log files and other meta-data through the web GUI
- Implement trigger rules for tasks
- Monitoring all jobs status in real time + Email alerts
- Community support

Writing your first workflow

Steps to write an Airflow DAG

- A DAG file, which is basically just a Python script, is a configuration file specifying the DAG's structure as code.
- There are only 5 steps you need to remember to write an Airflow DAG or workflow:
  - Step 1: Importing modules
  - Step 2: Default Arguments
  - Step 3: Instantiate a DAG
  - Step 4: Tasks
  - Step 5: Setting up Dependencies

Step 1: Importing modules

- Import Python dependencies needed for the workflow

```
from datetime import timedelta

import airflow
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
```

Step 2: Default Arguments

- Define default and DAG-specific arguments

```
default_args = {
    'owner': 'airflow',
    'start_date': airflow.utils.dates.days_ago(2),
    # 'end_date': datetime(2018, 12, 30),
    'depends_on_past': False,
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    # If a task fails, retry it once after waiting
    # at least 5 minutes
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    }
```

DAGs Summary

- **Directed Acyclic Graph** is a graph that has **no cycles** and the data in each node flows forward in only **one direction.**

Step 3: Instantiate a DAG

- Give the DAG name, configure the schedule, and set the DAG settings

```
dag = DAG(
    'tutorial',
    default_args=default_args,
    description='A simple tutorial DAG',
    # Continue to run DAG once per day
    schedule_interval=timedelta(days=1),
)
```

Step 4: Tasks

- The next step is to lay out all the tasks in the workflow.

Operators, and Tasks

- **DAGs** do not perform any actual computation. Instead, **Operators** determine what actually gets done.
- **Task**: Once an operator is instantiated, it is referred to as a "task". An operator describes a single task in a workflow.
  - Instantiating a task requires providing a unique `task_id` and `DAG` container
- A **DAG** is a container that is used to organize tasks and set their execution context.

```
# t1, t2 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',
    dag=dag,
)
```

```python
# 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',
    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',
    depends_on_past=False,
    bash_command=templated_command,
    params={'my_param': 'Parameter I passed in'},
    dag=dag,
)
```

Operators categories

Typically, **Operators** are classified into three categories:

- **Sensors**
- **Operators**
- **Transfers**

- **Sensors**: a certain type of operator that will keep running until a certain critera is met. Example include waiting for a certain time, external file, or upstream data source.
  - `HdfsSensor`: Waits for a file or folder to land in HDFS
  - `NamedHivePartitionSensor`: check whether the most recent partition of a Hive table is available for downstream processing.

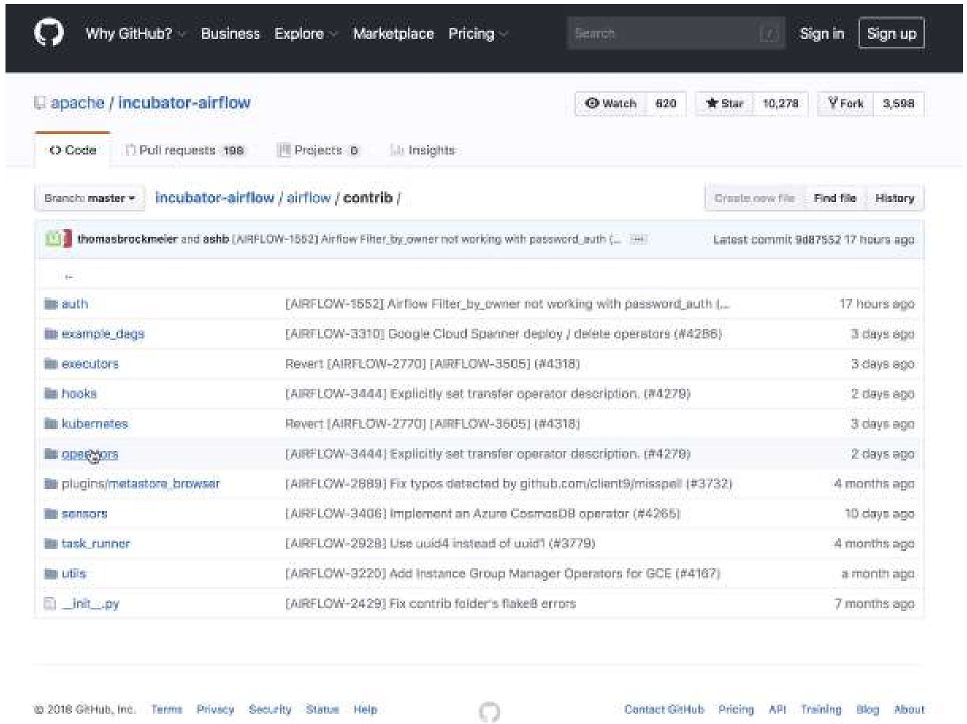
- **Operators**: triggers a certain action (e.g. run a bash command, execute a python function, or execute a Hive query, etc)
  - `BashOperator`: executes a bash command
  - `PythonOperator`: calls an arbitrary Python function
  - `HiveOperator`: executes hql code or hive script in a specific Hive database.
  - `BigQueryOperator`: executes Google BigQuery SQL queries in a specific BigQuery database

- **Transfers**: moves data from one location to another.
  - `MySqlToHiveTransfer`: Moves data from MySql to Hive.
  - `S3ToRedshiftTransfer`: load files from s3 to Redshift

Working with Operators

- Airflow provides prebuilt operators for many common tasks.
- There are more operators being added by the community. You can just go to the Airflow official Github repo, specifically in the `airflow/contrib/` directory to look for the community added operators.
- All operators are derived from `BaseOperator` and acquire much functionality through inheritance. Contributors can extend `BaseOperator` class to create custom operators as they see fit.

```
class HiveOperator(BaseOperator):
    """
    HiveOperator inherits from BaseOperator
    """
```



https://github.com/apache/airflow/tree/master/airflow/operators

Step 5: Setting up Dependencies

- Set the dependencies or the order in which the tasks should be executed in.
- Here's a few ways you can define dependencies between them:

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

# similar to above where t3 will depend on t1
t3.set_upstream(t1)
```

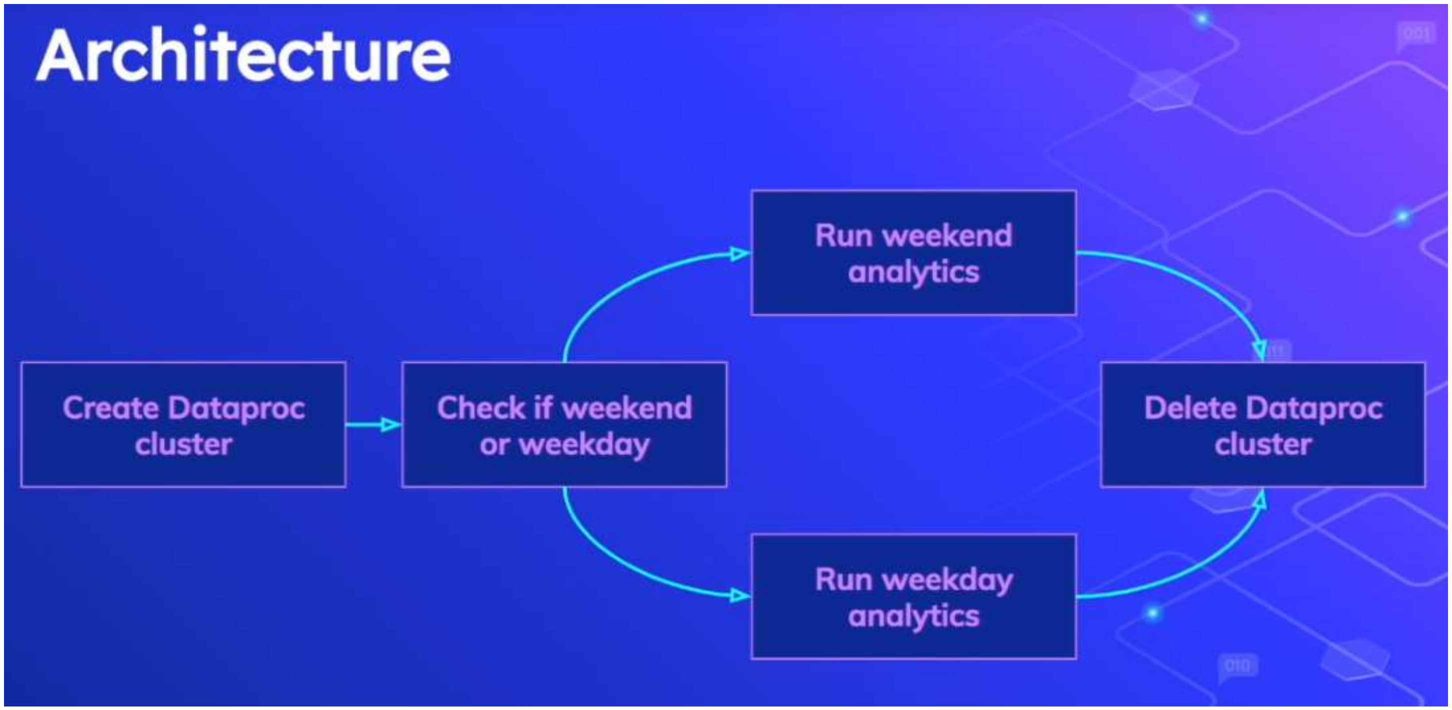
```
# The bit shift operator can also be
# used to chain operations:
t1 >> t2

# And the upstream dependency with the
# bit shift operator:
t2 << t1
```

```python
# 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
```

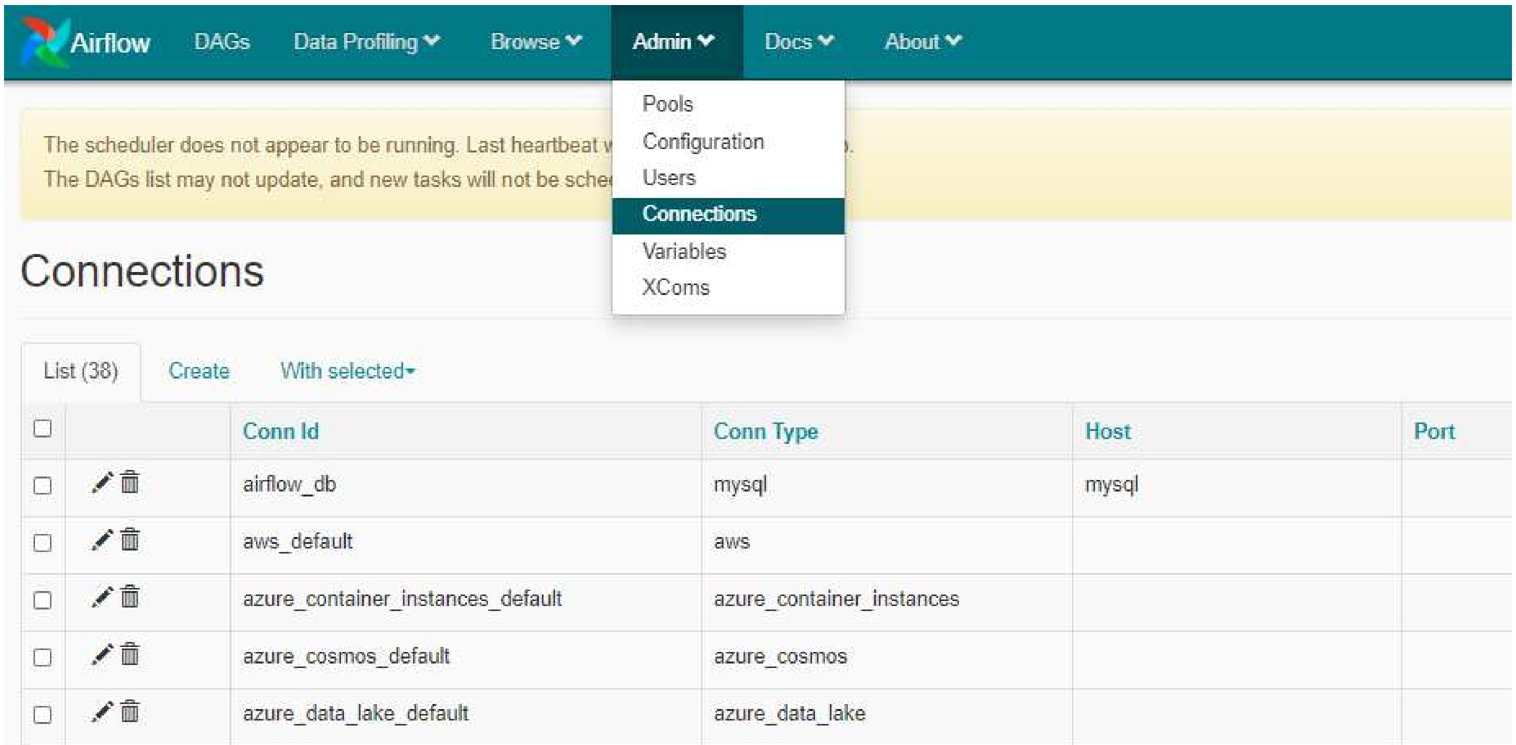
Recap

- Basically a DAG is just a Python file, which is used to organize tasks and set their execution context. DAGs do not perform any actual computation.
- Instead, tasks are the element of Airflow that actually “do the work” we want performed. And it is your job to write the configuration and organize the tasks in specific orders to create a complete data pipeline.

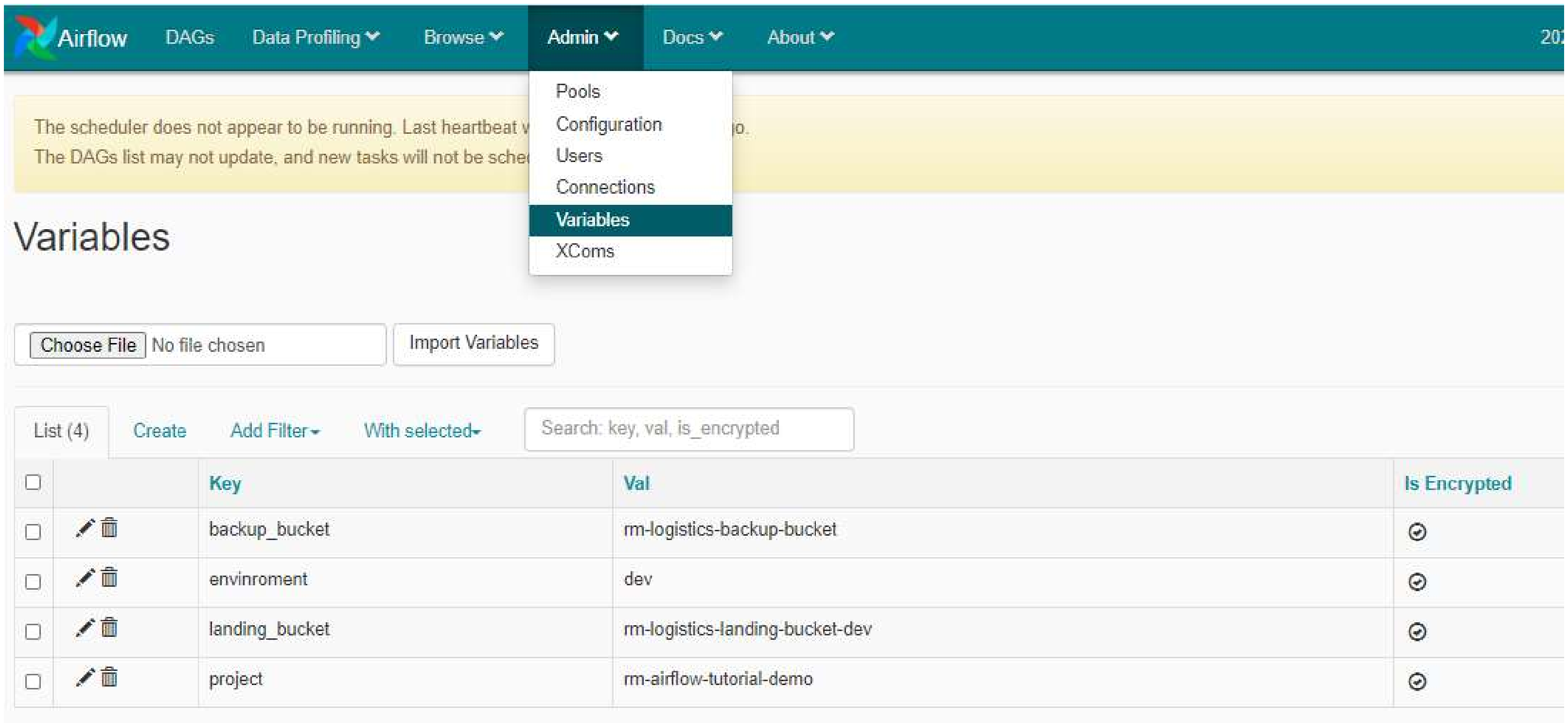
1


Architecture
Create Dataproc cluster
Check if weekend or weekday
Run weekend analytics
Run weekday analytics
Delete Dataproc cluster

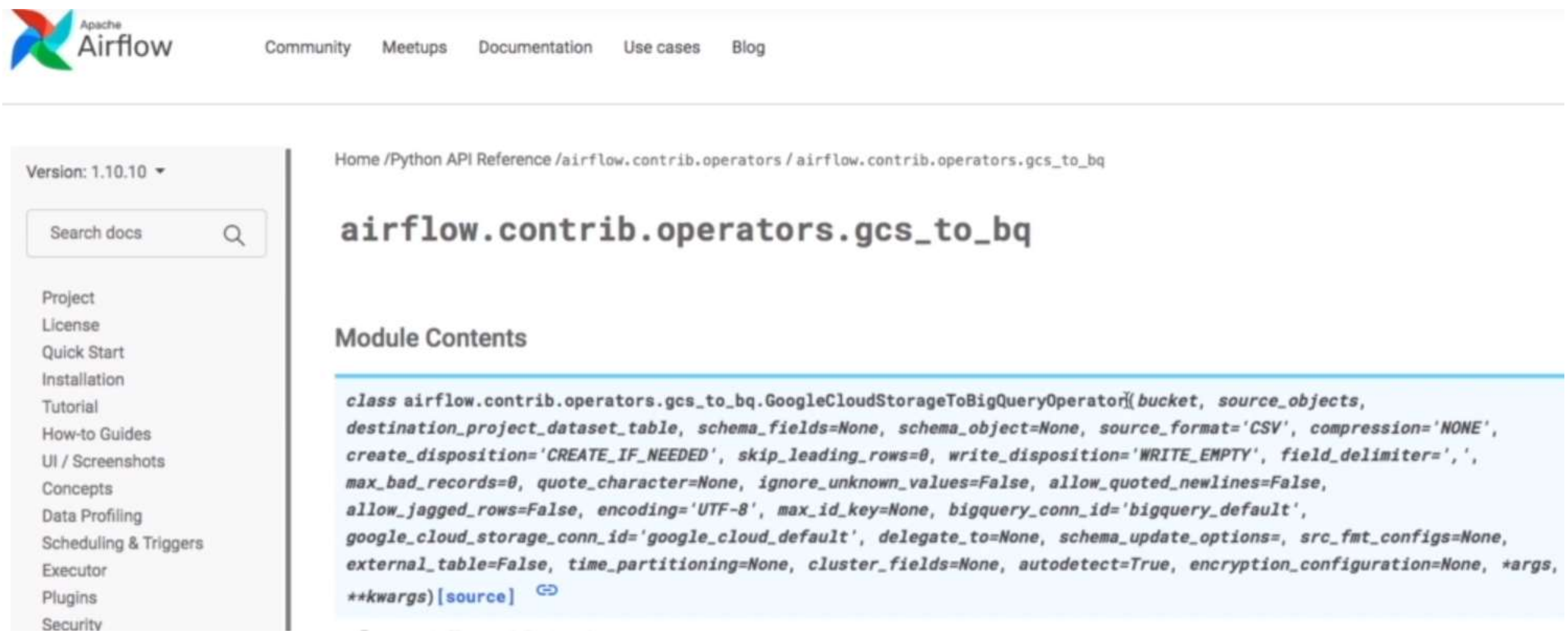
1.Create a Connection



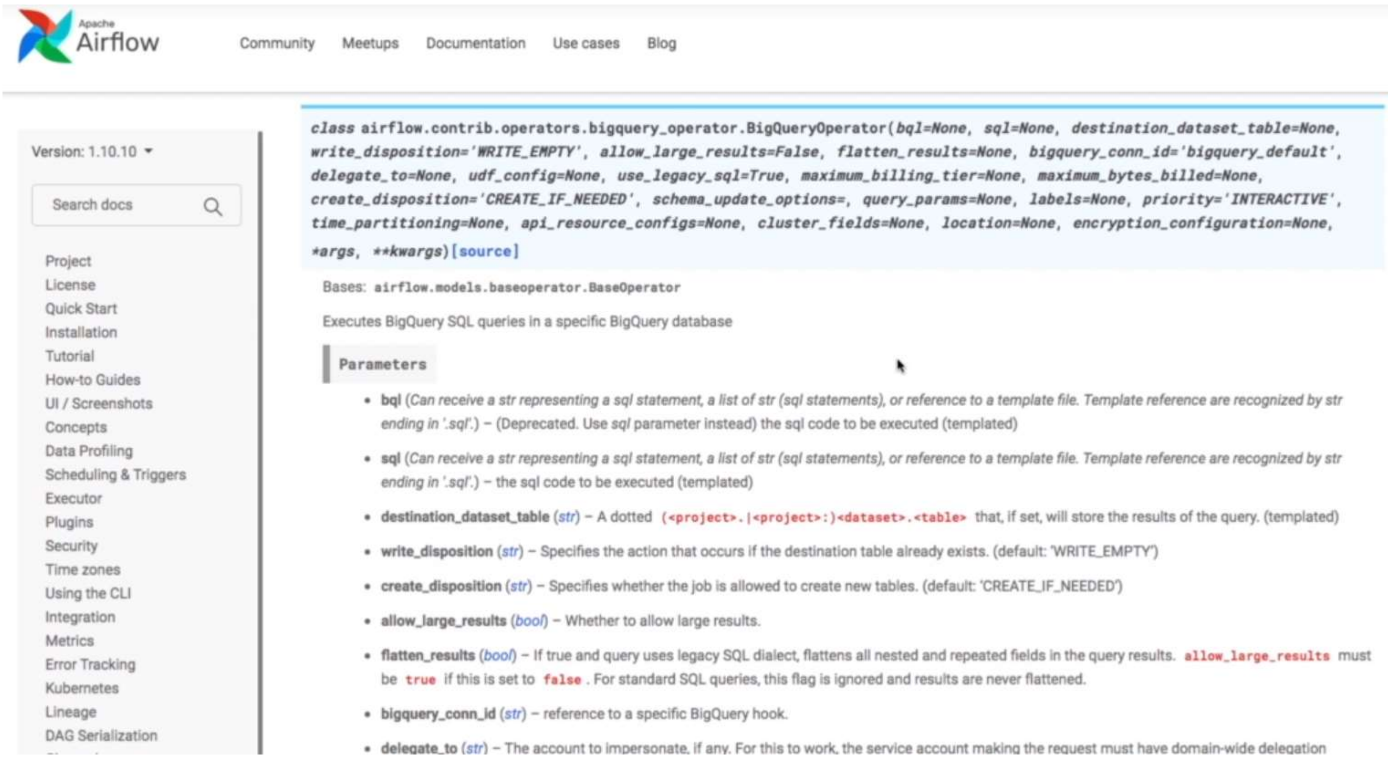
2. Create Variables



S3 to BigQuery



modak



Apache Airflow

Community Meetups Documentation Use cases Blog

Version: 1.10.10

Search docs

Project
License
Quick Start
Installation
Tutorial
How-to Guides
UI / Screenshots
Concepts
Data Profiling
Scheduling & Triggers
Executor
Plugins
Security
Time zones
Using the CLI
Integration
Metrics
Error Tracking
Kubernetes
Lineage
DAG Serialization

class **airflow.contrib.operators.bigquery_operator.BigQueryOperator**(*bql=None, sql=None, destination_dataset_table=None, write_disposition='WRITE_EMPTY', allow_large_results=False, flatten_results=None, bigquery_conn_id='bigquery_default', delegate_to=None, udf_config=None, use_legacy_sql=True, maximum_billing_tier=None, maximum_bytes_billed=None, create_disposition='CREATE_IF_NEEDED', schema_update_options=, query_params=None, labels=None, priority='INTERACTIVE', time_partitioning=None, api_resource_configs=None, cluster_fields=None, location=None, encryption_configuration=None, *args, **kwargs*)[source]

Bases: `airflow.models.baseoperator.BaseOperator`

Executes BigQuery SQL queries in a specific BigQuery database

Parameters

- **bql** (*Can receive a str representing a sql statement, a list of str (sql statements), or reference to a template file. Template reference are recognized by str ending in '.sql'.*) – (Deprecated. Use sql parameter instead) the sql code to be executed (templated)
- **sql** (*Can receive a str representing a sql statement, a list of str (sql statements), or reference to a template file. Template reference are recognized by str ending in '.sql'.*) – the sql code to be executed (templated)
- **destination_dataset_table** (*str*) – A dotted `(<project>.|<project>:)<dataset>.<table>` that, if set, will store the results of the query. (templated)
- **write_disposition** (*str*) – Specifies the action that occurs if the destination table already exists. (default: 'WRITE_EMPTY')
- **create_disposition** (*str*) – Specifies whether the job is allowed to create new tables. (default: 'CREATE_IF_NEEDED')
- **allow_large_results** (*bool*) – Whether to allow large results.
- **flatten_results** (*bool*) – If true and query uses legacy SQL dialect, flattens all nested and repeated fields in the query results. `allow_large_results` must be `true` if this is set to `false`. For standard SQL queries, this flag is ignored and results are never flattened.
- **bigquery_conn_id** (*str*) – reference to a specific BigQuery hook.
- **delegate_to** (*str*) – The account to impersonate, if any. For this to work, the service account making the request must have domain-wide delegation

https://airflow.apache.org/docs/apache-airflow/1.10.14/_api/airflow/contrib/operators/bigquery_operator/index.html

LET´S SEE SOME CODE…

2

DEMO 2 – Airflow GCP and PySpark

- Create a Dataproc
- Process PySpark and convert JSON to Parquet file
- Delete the cluster

LET´S SEE SOME CODE...

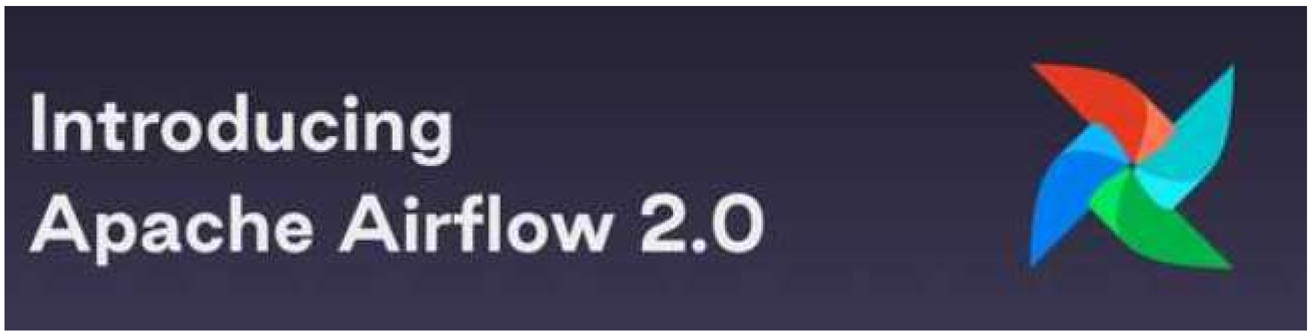
OTHER INFO

How to Install Apache Airflow

- https://www.analyticsvidhya.com/blog/2020/11/getting-started-with-apache-airflow/

Apache Airflow Providers

- https://github.com/apache/airflow/tree/master/airflow/providers



https://airflow.apache.org/blog/airflow-two-point-oh-is-here/

APACHE AIRFLOW 2.0 !!!

The full changelog is about 3,000 lines long, here some of the major features in 2.0.0 compared to 1.x:

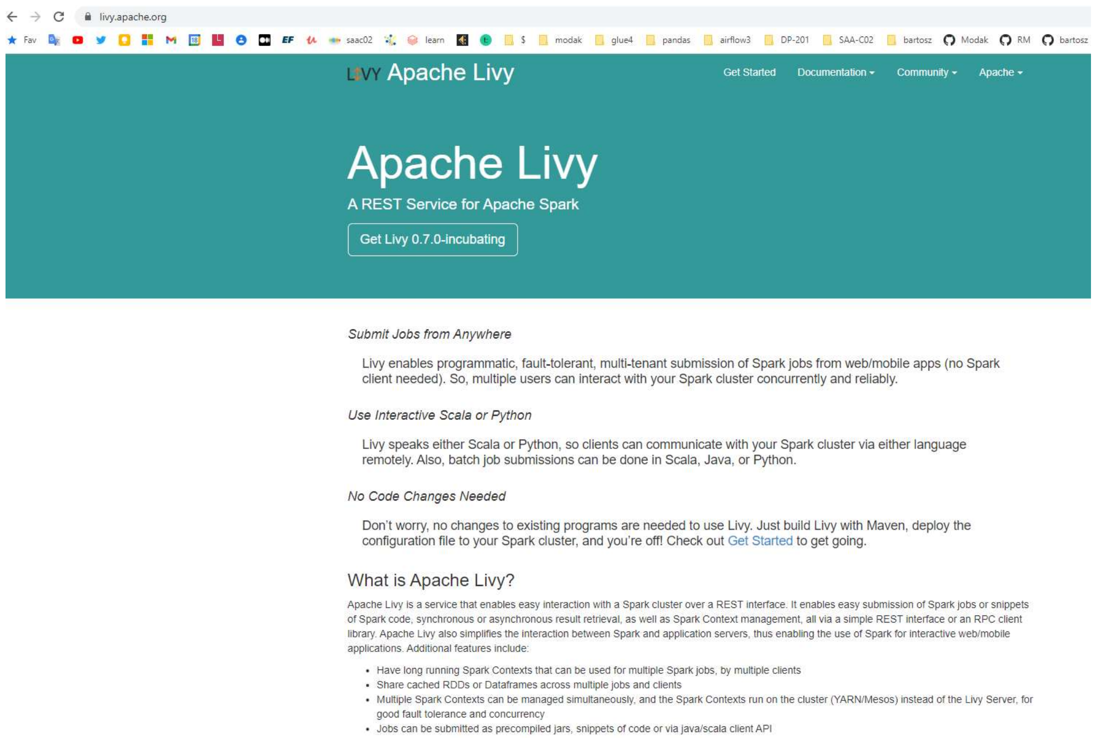
- ✔ TaskFlow API
- ✔ Fully specified REST API
- ✔ Massive Scheduler performance improvements
- ✔ Scheduler is now HA compatible
- ✔ Task Groups
- ✔ Refreshed UI
- ✔ Smart Sensors
- ✔ Simplified KubernetesExecutor
- ✔ Splitting Airflow into 60+ packages
- ✔ Improved Security & Configuration

More info:
https://www.astronomer.io/blog/introducing-airflow-2-0

NEXT STEPS

PoC EMR and Livy

1. Create a EMR Cluster using Obot3
2. Submit a Spark Job using Apache Livy - https://livy.apache.org/



livy.apache.org

Apache Livy | Get Started | Documentation | Community | Apache

Apache Livy

A REST Service for Apache Spark

Get Livy 0.7.0-incubating

Submit Jobs from Anywhere

Livy enables programmatic, fault-tolerant, multi-tenant submission of Spark jobs from web/mobile apps (no Spark client needed). So, multiple users can interact with your Spark cluster concurrently and reliably.

Use Interactive Scala or Python

Livy speaks either Scala or Python, so clients can communicate with your Spark cluster via either language remotely. Also, batch job submissions can be done in Scala, Java, or Python.

No Code Changes Needed

Don't worry, no changes to existing programs are needed to use Livy. Just build Livy with Maven, deploy the configuration file to your Spark cluster, and you're off! Check out Get Started to get going.

What is Apache Livy?

Apache Livy is a service that enables easy interaction with a Spark cluster over a REST interface. It enables easy submission of Spark jobs or snippets of Spark code, synchronous or asynchronous result retrieval, as well as Spark Context management, all via a simple REST interface or an RPC client library. Apache Livy also simplifies the interaction between Spark and application servers, thus enabling the use of Spark for interactive web/mobile applications. Additional features include:

- Have long running Spark Contexts that can be used for multiple Spark jobs, by multiple clients
- Share cached RDDs or Dataframes across multiple jobs and clients
- Multiple Spark Contexts can be managed simultaneously, and the Spark Contexts run on the cluster (YARN/Mesos) instead of the Livy Server, for good fault tolerance and concurrency
- Jobs can be submitted as precompiled jars, snippets of code or via java/scala client API

https://livy.apache.org/

LIVY Apache Livy   Get Started   Documentation ▾   Community ▾   Apache ▾

Apache Livy Examples

Spark Example

Here's a step-by-step example of interacting with Livy in Python with the Requests library. By default Livy runs on port 8998 (which can be changed with the `livy.server.port` config option). We'll start off with a Spark session that takes Scala code:

```
sudo pip install requests
```

```
import json, pprint, requests, textwrap
host = 'http://localhost:8998'
data = {'kind': 'spark'}
headers = {'Content-Type': 'application/json'}
r = requests.post(host + '/sessions', data=json.dumps(data), headers=headers)
r.json()

{u'state': u'starting', u'id': 0, u'kind': u'spark'}
```

Once the session has completed starting up, it transitions to the idle state:

```
session_url = host + r.headers['location']
r = requests.get(session_url, headers=headers)
r.json()

{u'state': u'idle', u'id': 0, u'kind': u'spark'}
```

Now we can execute Scala by passing in a simple JSON command:

```
statements_url = session_url + '/statements'
data = {'code': '1 + 1'}
r = requests.post(statements_url, data=json.dumps(data), headers=headers)
r.json()

{u'output': None, u'state': u'running', u'id': 0}
```

If a statement takes longer than a few milliseconds to execute, Livy returns early and provides a statement URL that can be polled until it is complete:

```
statement_url = host + r.headers['location']
r = requests.get(statement_url, headers=headers)
pprint.pprint(r.json())

{u'id': 0,
  u'output': {u'data': {u'text/plain': u'res0: Int = 2'},
              u'execution_count': 0,
```

https://livy.apache.org/examples/

modak

Questions & Answers

Thank You !!!