Introduction To Airflow

Lab 8 - CSEN1095

Overview

What is Airflow?

- Airflow is a platform to programmatically author, schedule, and monitor workflows or data pipelines.

What is a Workflow?

- a sequence of tasks
- started on a schedule or triggered by an event
- frequently used to handle big data processing pipelines

A Typical Workflow

1. download data from source
2. send data somewhere else to process
3. Monitor when the process is completed
4. Get the result and generate the report
5. Send the report out by email

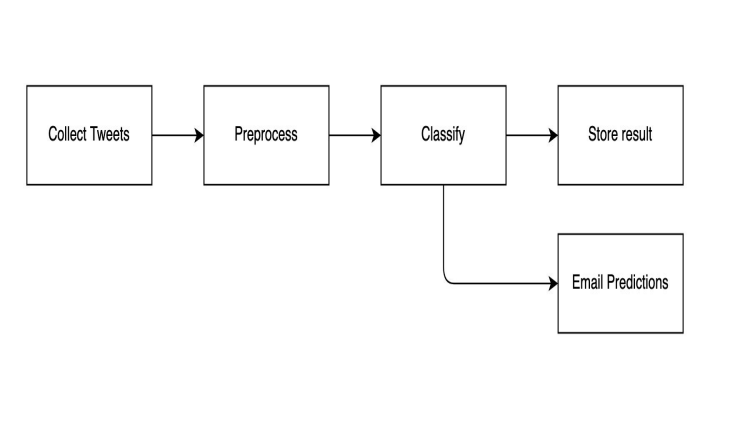
Why do we need Airflow? - Example

- Data grows fast, gets more complex and harder to manage as your company scales.
- In order to provide insights, you need to have some kind of visualization to explain your findings and monitor them over time.
- For these data to be up to date, you need to extract, transform, load them into your preferred database from multiple data sources in a fixed time interval (hourly , daily, weekly, monthly).

Why do we need Airflow? - Example

- Imagine you have an ML model that does twitter sentiment analysis and you want to run that model on a specific tweets category everyday.

- As you can see the data flows from one end of the pipeline to the other end.
- You can automate the process using Airflow

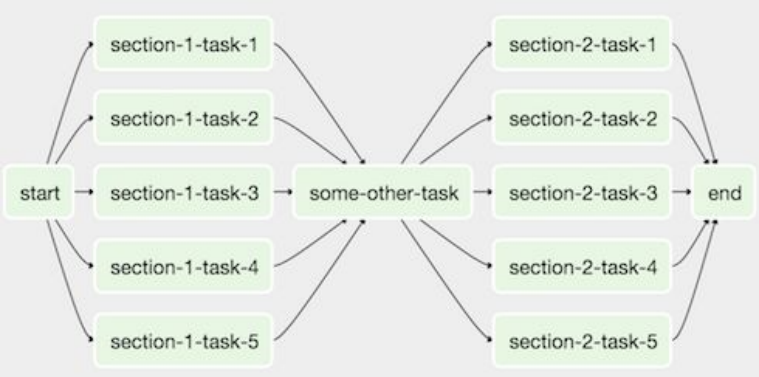


What makes Airflow great?

- Monitoring: success or failure status, how long does the process run? + email alerts
- Can handle dependencies gracefully
- Easy to reprocess historical jobs by date, or re-run for specific intervals
- Handle errors and failures gracefully. Automatically retry when a task fails.
- Community support
- Integrations with a lot of infrastructure (Hive, Presto, AWS, Google cloud, etc)

Airflow DAG

- Airflow provides DAG python class to create a Directed Acyclic Graph in order for you to define your workflow.
- Airflow DAGs are composed of Tasks.



Scheduling your DAGs

- There are two parameters that you can define when instantiating your DAG to specify when your DAG will be run:
  1. start_date
  2. schedule_interval: dictates how often to run the DAG
     - Can be defined using:
       i. cron expressions
       ii. cron presets
       iii. datetime.timedelta object

Scheduling your DAGs - Cont.

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

Scheduling your DAGs - Cont.

The cron expression is made of five fields. Each field can have the following values.

*	*	*	*	*
minute (0-59)	hour (0 - 23)	day of the month (1 - 31)	month (1 - 12)	day of the week (0 - 6)

- You can play around with its syntax here: https://crontab.guru/

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.
- A **DAG** is a container that is used to organize tasks and set their execution context.

Types of Operators

- Operators define the nodes of the DAG. Each operator is an independent task.
- Examples:
  1. PythonOperator: when a PythonOperator is run, it will run the code of the python_callable function

```
def print_function():
    print ("Hey I am a task")

run_this_last = PythonOperator(
    task_id='run_this_last',
    dag=dag,
    python_callable=print_function
)
```

Types of Operators - Cont.

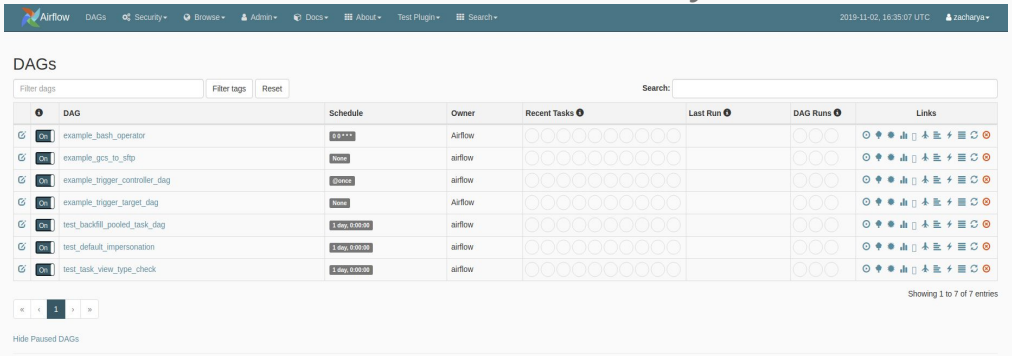
2. BashOperator: runs a bash command

```
run_this = BashOperator(
    task_id='run_after_loop',
    bash_command='echo 1',
    dag=dag,
)
```

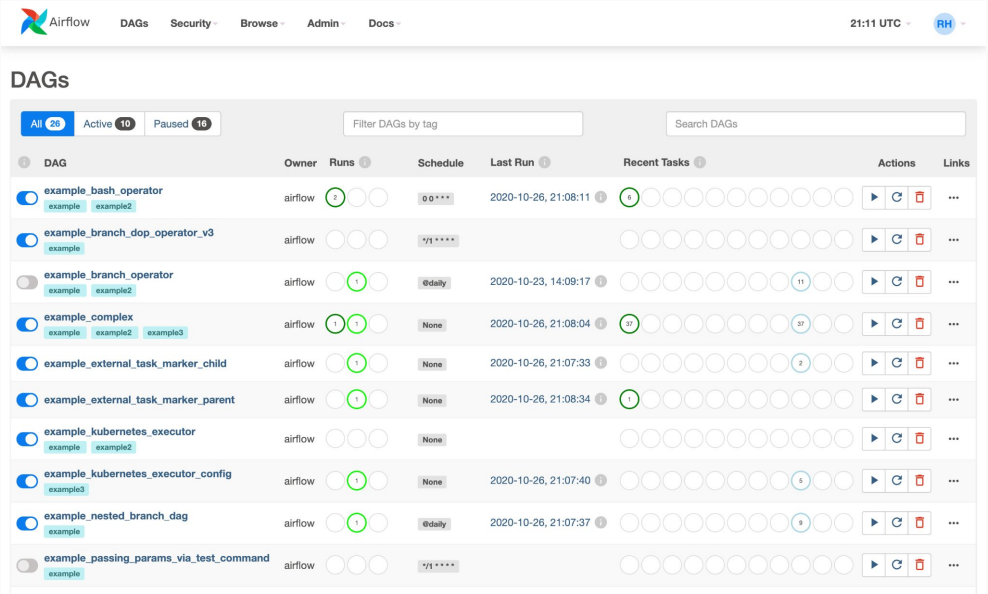
- There are tons of Operators that are open source that perform multiple tasks. You can even write your own operator.

UI

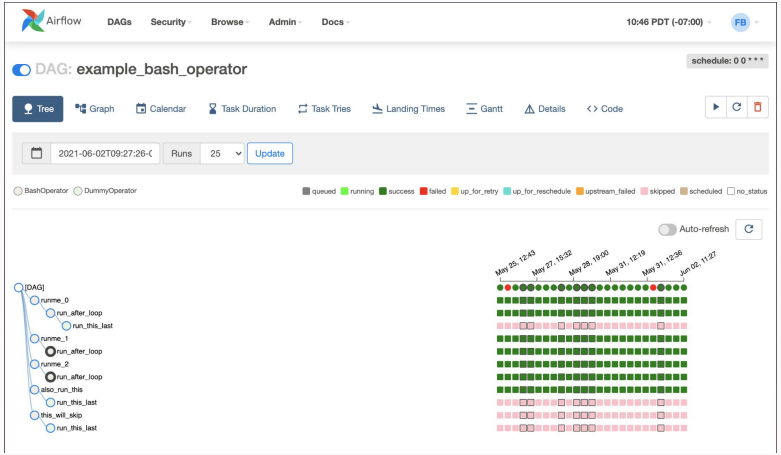
- Airflow also has a shiny UI that allows you to manage and monitor your workflows. Dependencies are built more easily, logs are easily accessible, code can be easily read, time spent on each task, time to finish, trigger/pause workflows with a click of a button and many more can be done with the UI.



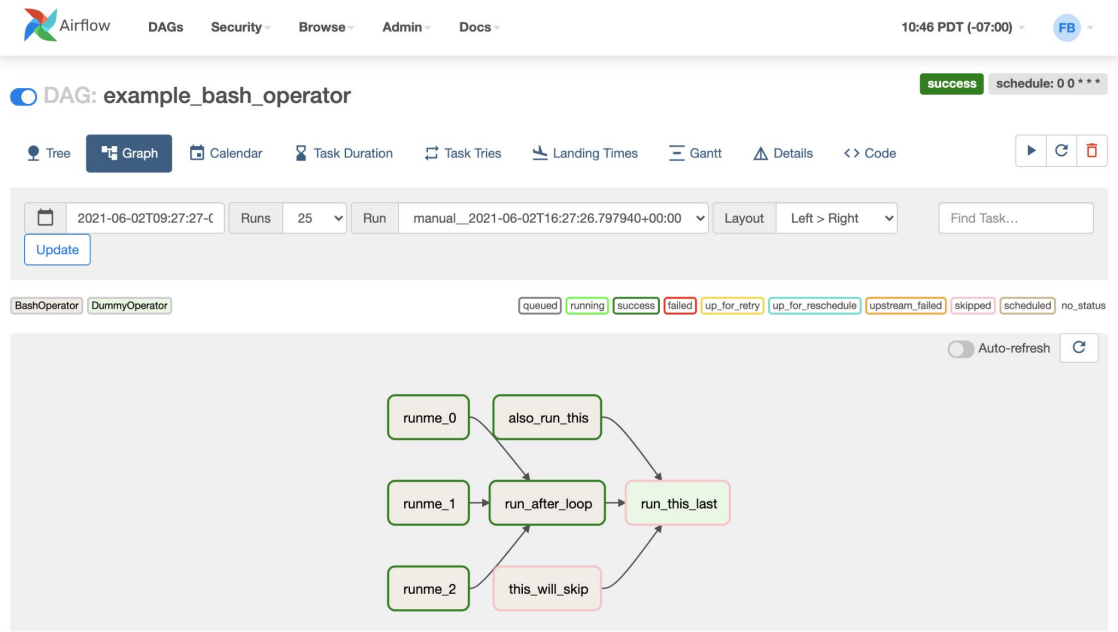
UI - DAGs View


Airflow
DAGs
Security
Browse
Admin
Docs
21:11 UTC
RH
DAGs
All 26
Active 10
Paused 16
Filter DAGs by tag
Search DAGs
DAG
Owner
Runs
Schedule
Last Run
Recent Tasks
Actions
Links
example_bash_operator
example
example2
airflow
2
0 0 * * *
2020-10-26, 21:08:11
6
example_branch_dop_operator_v3
example
airflow
*/1 * * * *
example_branch_operator
example
example2
airflow
1
@daily
2020-10-23, 14:09:17
11
example_complex
example
example2
example3
airflow
1
1
None
2020-10-26, 21:08:04
37
37
example_external_task_marker_child
airflow
1
None
2020-10-26, 21:07:33
2
example_external_task_marker_parent
airflow
1
None
2020-10-26, 21:08:34
1
example_kubernetes_executor
example
example2
airflow
None
example_kubernetes_executor_config
example3
airflow
1
None
2020-10-26, 21:07:40
5
example_nested_branch_dag
example
airflow
1
@daily
2020-10-26, 21:07:37
9
example_passing_params_via_test_command
example
airflow
*/1 * * * *

UI - Tree View


Airflow
DAGs
Security
Browse
Admin
Docs
10:46 PDT (-07:00)
FB
DAG: example_bash_operator
schedule: 0 0 * * *
Tree
Graph
Calendar
Task Duration
Task Tries
Landing Times
Gantt
Details
Code
2021-06-02T09:27:26-(
Runs
25
Update
BashOperator
DummyOperator
queued
running
success
failed
up_for_retry
up_for_reschedule
upstream_failed
skipped
scheduled
no_status
Auto-refresh
May 25, 12:43
May 27, 15:32
May 28, 19:00
May 31, 12:19
May 31, 12:36
Jun 02, 11:27
[DAG]
runme_0
run_after_loop
run_this_last
runme_1
run_after_loop
runme_2
run_after_loop
also_run_this
run_this_last
this_will_skip
run_this_last

UI - Graph View



Example 1

Step 1: Import 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

```
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),
    }
```

Step 3: Instantiate a DAG

```
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

```
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,
)
```

Step 5: Setting up Dependencies

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

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

Step 5 - Note

In case of parallel dependencies:

```
# 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]
```

Example 2

Covid DAG

- Define a DAG that receives covid statistics from an API and stores the data in a csv file.

Step 1 - Import modules

- Import Python dependencies needed for the workflow

```
# step 1 - import modules
import requests
import json

from airflow import DAG
from datetime import datetime
from datetime import date
# Operators; we need this to operate!
from airflow.operators.python_operator import PythonOperator

import pandas as pd
```

Step 2 - Default Arguments

- Define default and DAG-specific arguments

```
# step 2 - define default args
# 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': datetime(2020, 12, 13)
    }
```

Step 3 - Instantiate a DAG

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

```
# step 3 - instantiate DAG
dag = DAG(
    'covid-DAG',
    default_args=default_args,
    description='Fetch covid data from API',
    schedule_interval='@once',
)
```

Step 4 - Tasks

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

```python
# step 4 Define tasks
def store_data(**context):
    df = context['task_instance'].xcom_pull(task_ids='extract_data')
    df = df.set_index("date_of_interest")
    df.to_csv("data/nyccovid.csv")


def extract_data(**kwargs):
    url = "https://data.cityofnewyork.us/resource/rc75-m7u3.json"
    response = requests.get(url)
    df = pd.DataFrame(json.loads(response.content))
    return df



t1 = PythonOperator(
    task_id='extract_data',
    provide_context=True,
    python_callable=extract_data,
    dag=dag,
)

t2 = PythonOperator(
    task_id='store_data',
    provide_context=True,
    python_callable=store_data,
    dag=dag,
)
```

Step 5 - Setting up Dependencies

- Set the dependencies or the order in which the tasks should be executed.

```
# step 5 - define dependencies
t1 >> t2
```

Task

Task Description

- Implement an Airflow pipeline that does the following tasks **every 5 minutes**:
  1. Read a csv file: https://raw.githubusercontent.com/raneemsultan/Data-Engineering-W21main/Lab8/people.csv
  2. Drop any null values
  3. Store the csv file with the timestamp concatenated to the name

References

- https://www.applydatascience.com/airflow/airflow-tutorial-introduction/
- https://towardsdatascience.com/data-engineering-basics-of-apache-airflow-build-your-first-pipeline-eefecb7f1bb9
- https://bhavaniravi.com/blog/apache-airflow-introduction
- https://morioh.com/p/643f1be319e9
- https://bigishdata.com/2020/04/05/apache-airflow-part-1-introduction-setup-and-writing-data-to-files/
- https://github.com/jduran9987/airflow-covid
- https://airflow.apache.org/docs/apache-airflow/1.10.1/scheduler.html