Versión Castellano

Flujos de trabajo en Cloud Computing con Apache AirFlow.

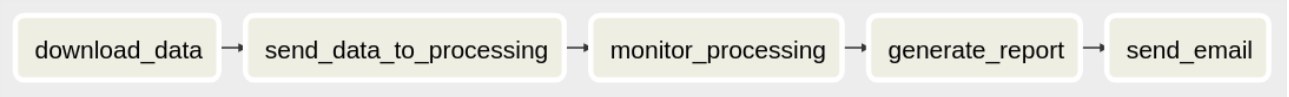
¿Qué es Apache Airflow? 1
¿Para que se usa Apache Airflow? 2
¿Qué es un flujo de trabajo en Cloud Computing? 3
¿Por qué usar AirFlow o una herramienta de Flujo de Trabajo en Cloud Computing? 4
Instalación de Apache AirFlow 6
Inicialización del planificador 7
Inicialización del servicio web y línea de órdenes para de gestión de Flujos de trabajo 8
Creación de un flujo de trabajo en AirFlow 9
Operadores o tipos de tareas en el flujo de trabajo 12
Ejemplo completo de flujo de trabajo con múltiples operadores 13
Referencias 18

¿Qué es Apache Airflow?

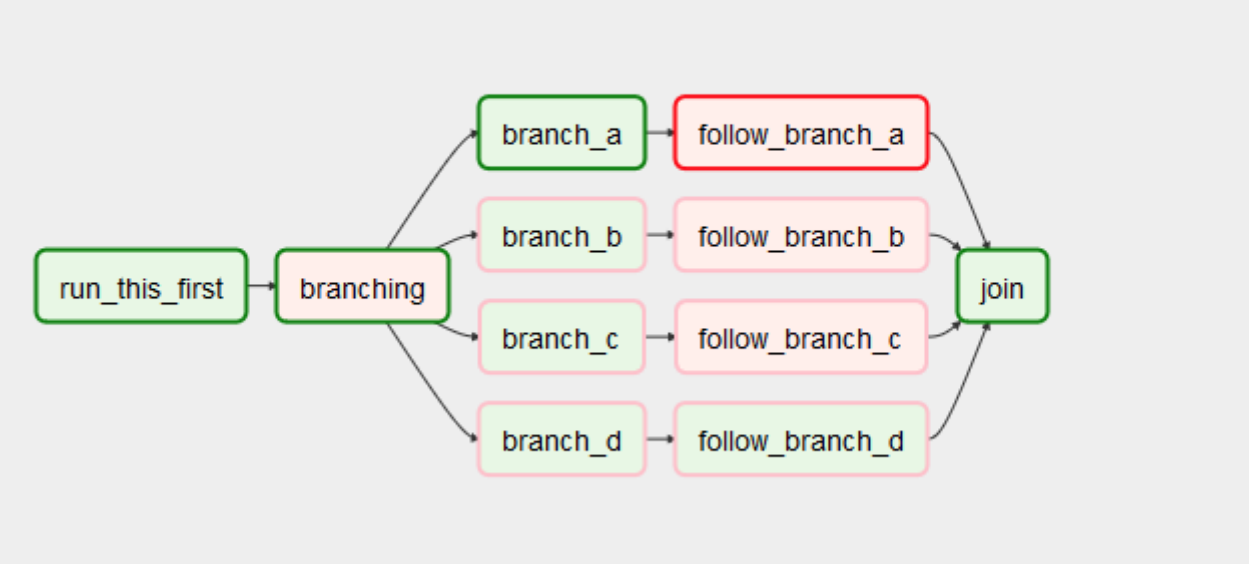
Airflow (https://airflow.apache.org/) es una plataforma para programar y supervisar los flujos de trabajo. Sirve para para crear flujos de trabajo compuestos por tareas cuyas dependencias se pueden modelar mediante Grafos Acíclicos Dirigidos (DAG). Airflow integra un “planificador” que ejecuta las tareas en un conjunto de nodos/instancias llamados “workers”. Permite gestionar todos los flujos de trabajo desde una interfaz en Línea de órdenes (Shell) o bien desde una web de administración. Esta web permite la visualización de los flujos de datos que se ejecutan en la producción, el control del progreso y la gestión de problemas con datos y los flujos de trabajo a gran escala.

¿Para que se usa Apache Airflow?

Si reducimos al mínimo AirFlow, básicamente AirFlow, ayuda a automatizar “las cosas” para realizar tareas (de cualquier tipo). Está hecho en Python, pero puedes ejecutar un programa independientemente del lenguaje. Por ejemplo, la primera etapa de su flujo de trabajo tiene que ejecutar un programa basado en C++ para realizar el análisis de imágenes y luego un programa basado en Python para transferir esa información a un servicio cloud para publicar resultados. Las posibilidades son infinitas.
Puede hacer cosas tan simples como la mostrada en la imagen inferior (descargar datos, enviar datos, monitorizar el procesamiento, crear un informe y luego enviar correo con resultados):



hasta otras mucho más completas que requieren de cálculo distribuido intensivo por ejemplo como en la imagen siguiente, donde se utiliza en otros paquetes de software (muy relevantes hoy en día, como Spark y Tensorflow), para crear modelos de ejecución:



Algunos de los aspectos más destacados de Airflow son los siguientes:

Gestión de los fallos en las fuentes:
Gestión integrada de los errores en el acceso a los datos. Es posible definir un comportamiento diferente para cada caso.

Reprocesamiento de trabajos históricos:
ordenando los trabajos pasados para ser reprocesados directamente desde la GUI de una manera muy simple.

Parámetros entre trabajos:
intercambio de parámetros entre diferentes trabajos a través de un buffer. Esto nos permite asignar estados a los flujos de trabajo.

Reintentos automáticos:
gestión automática de reintentos basada en la configuración de cada DAG.

Soporte de CI y CD en los flujos de trabajo:
integrando fácilmente Airflow con nuestros sistemas de integración o despliegue continuo ya sea usando herramientas como Jenkins o gestionando los DAG en GitHub.

Muchas integraciones:
Airflow se puede integrar con una gran cantidad de plataformas y software de terceros a través de los operadores. Muchas de ellas son contribuciones de la comunidad: Hive, Presto, Druid, AWS, Google Cloud, Azure, Databricks, Jenkins, Kubernetes, Mongo, Oracle, SSH, etc. con lo que podemos usar de forma transparente esos recursos como tareas dentro del procesamiento.

Sensores de datos:
son un tipo particular de operador que nos permite esperar hasta que se cumpla una determinada condición, por ejemplo, que un archivo sea escrito en HDFS o S3.

Capacidades de prueba de trabajo:
nos permite probar y validar nuestras pruebas antes del despliegue.

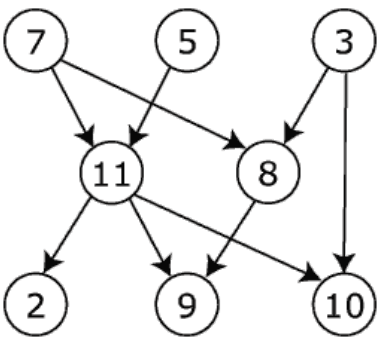
Triggers de tareas:
hay diferentes maneras en las que podemos lanzar nuestros flujos de trabajo para que se ejecuten automáticamente en función de un calendario; también podemos ejecutarlos a mano.

Monitorización y alertas de RT:
podemos monitorear el estado de ejecución de nuestros flujos de trabajo en tiempo real desde la interfaz gráfica.

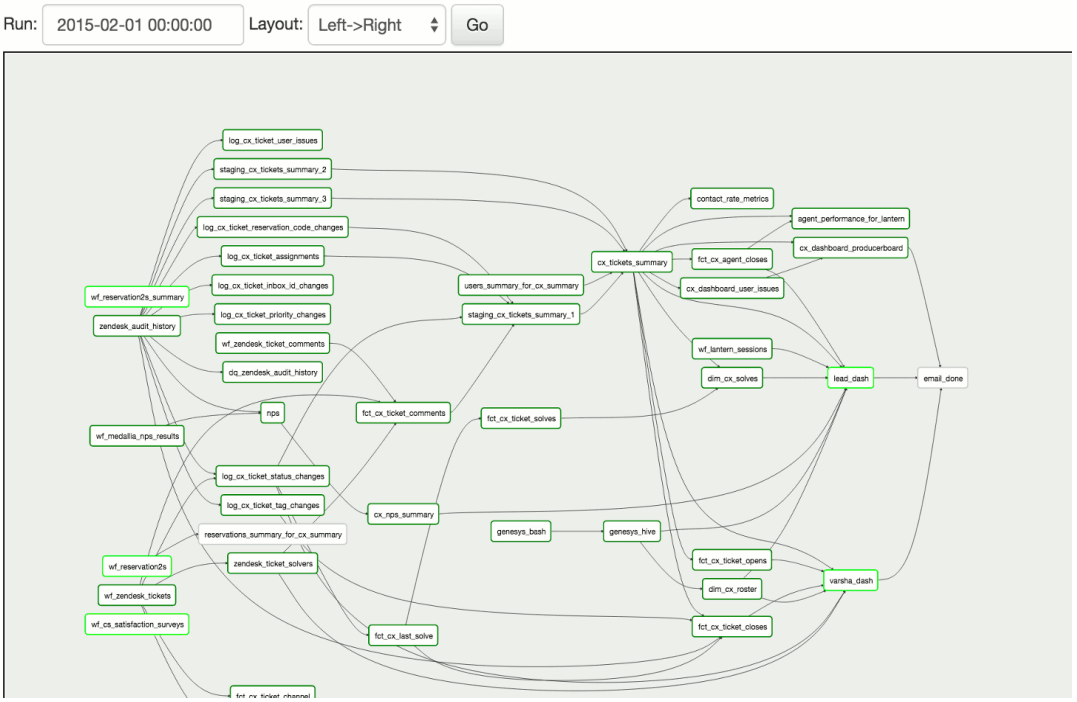
¿Qué es un flujo de trabajo en Cloud Computing?

Un flujo de trabajo puede representarse como un grafo dirigido acíclico, también llamado DAG (del inglés, Directed Acyclic Graph). Se trata de es un grafo dirigido (los arcos tienen un sentido definido) que no tiene ciclos. Esto significa que para cada vértice v, no hay un camino que empiece y otro que termine en v.

En este ejemplo de DAG, las tareas que se inician son (7,5,3) y podrían hacerse en paralelo.



Cuando nos referimos a un flujo de trabajo en Cloud Computing, nos referimos a tareas a realizar mediante servicios cloud cuyas interdependencias se pueden expresar mediante un DAG. Esto permite recoger todas las posibilidades que se presentan habitualmente una o varias tareas secuenciales, paralelas o una mezcla de ambas concreto será desplegar un servicio en Cloud Computing siguiendo todos los aspectos clave del concepto de este paradigma. Una filosofía de arquitecturas de especial interés es Cloud Native. La filosofía se centra en **cómo se crean y despliegan las aplicaciones**, pero no dónde, explotando las ventajas del modelo de despliegue que ofrece Cloud Computing. Un aspecto clave para entender el modelo de Cloud Native es que tanto la creación, el despliegue, la autoría, y el procesamiento de todos los componentes desde que tenemos el código fuente hasta que se tiene el producto final o servicio funcionando en Cloud, se corresponde con una serie de operaciones vistas como un DAG.
En la imagen siguiente se puede ver un DAG con varias decenas de operaciones correspondientes a todo el ciclo de vida de un servicio Cloud Native utilizando flujos de trabajo con AirFlow, donde prácticamente admite cualquier tipo de operación, tarea, o funcionalidad en cada uno de los nodos.

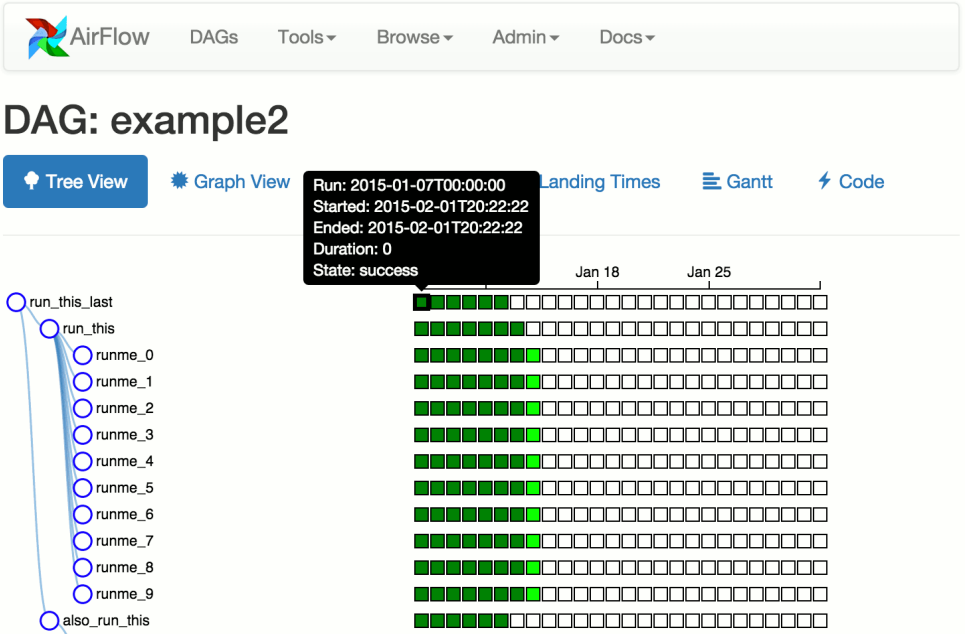


¿Por qué usar AirFlow o una herramienta de Flujo de Trabajo en Cloud Computing?

La respuesta a esta cuestión se basa en estos cuatro principios:

- Dinámico: Los flujos de datos están configurados como código (Python, Scala, R, Java, JS, ...), lo que permite la generación de flujos dinámicos. Esto permite escribir código que instancie tareas de forma dinámica.
- Extensible: Permite definir fácilmente sus propios operadores, ejecutores y extender las bibliotecas para que se ajuste al nivel de abstracción que se adapte a su entorno.
- Elegante: Los flujos de datos son sencillos y muy explícitos.
- Escalable: Airflow tiene una arquitectura modular y utiliza una cola de mensajes para orquestar un número arbitrario de workers. Airflow está listo para escalar sin límites[1].

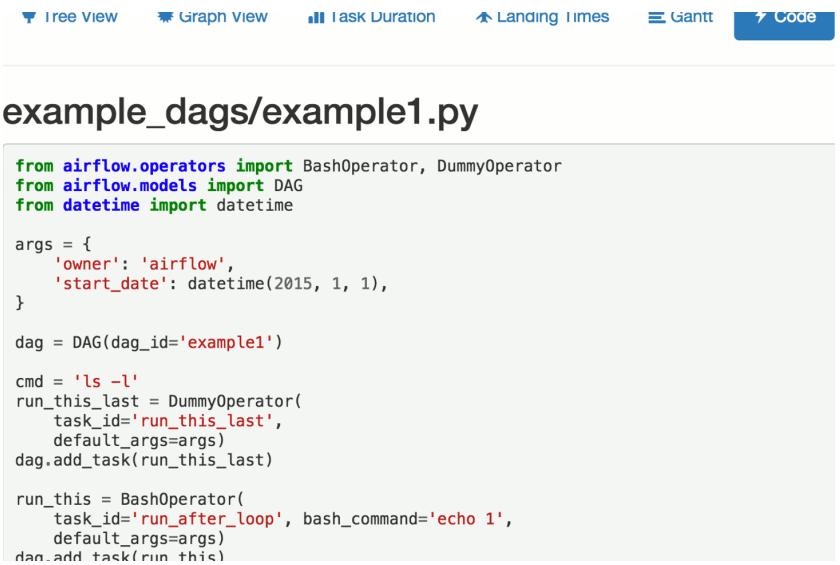
En la imagen siguiente se puede ver el estado de la ejecución de un DAG de tareas en como flujo de trabajo para cada una de las operaciones, además de la secuenciación temporal de las mismas:



Otro elemento clave para el uso de Flujos de trabajo con AirFlow en Cloud Computing es que permite codificar los flujos de trabajo como código, lo cual da mucha versatilidad a la hora de programáticamente crear este tipo de Flujos desde diferentes APIs.

En la imagen siguiente se puede ver un ejemplo de cómo se realiza la implementación de un Flujo de trabajo en Cloud Computing, con diferentes operadores (o tareas), para Python, por ejemplo:

[1] Limitado a la cantidad de recursos que se tenga.



Instalación de Apache AirFlow

La instalación de AirFlow es sencilla, ya que se instala como cualquier paquete en Python desde la línea de órdenes de tu sistema y por tanto es independiente de la plataforma (hardware o Sistema Operativo) que se use, ya sea Mac, Linux o Windows. La instalación se realiza desde un shell (o terminal de Linux, o bien desde PowerShell si usas Windows).

Para instalar Airflow, necesitas:

- Python >= 3.6 y pip3 o pip instalado con Python (prueba `python -v` para saber tu versión).
- En caso de no tener tener Python en la versión 3.6, descargar CONDA para conmutar entre la versión 2.7 y la 3.X
  https://salishsea-meopar-docs.readthedocs.io/en/latest/work_env/python3_conda_environment.html

Pasos para la instalación rápida (en Linux):

Los pasos a seguir para instalar Airflow en un equipo con Linux son sencillos. Desde cualquiera de las distribuciones de linux, abre una ventana de Shell/Terminal:

Primer paso, instalar el software con PIP:

```
pip install apache-airflow
```

O bien si tienes pip3 usa:

```
pip3 install apache-airflow
```

Segundo paso, inicializar la Base de Datos:

```
airflow initdb
```

Con estos pasos AirFlow está instalado

Otras opciones de instalación más avanzadas:

La forma más fácil de instalar la última versión estable de Airflow es con pip:

```
pip install apache-airflow
```

También puedes instalar Airflow con soporte para características adicionales como gcp o postgres dependiendo de las necesidades de tu sistema:

```
pip install apache-airflow[postgres,gcp]
```

O bien si necesitas soporte para Kubernetes, AWS, Azure, etc. usamos lo siguiente:

```
pip install 'apache-airflow[aws]'

pip install 'apache-airflow[azure]'

pip install 'apache-airflow[kubernetes]'
```

Tiene paquetes/tareas para prácticamente cualquier entorno Cloud que queramos desplegar, aquí están todos los soportados:
http://airflow.apache.org/docs/stable/installation.html#extra-packages

Nosotros instalaremos AirFlow con :

```
pip install apache-airflow
```

Una vez hecho esto, necesitamos inicializar la Base de Datos que controla todo el sistema de AirFlow. Para ello, por defecto usaremos la que trae (es SQLite), aunque es posible utilizar otros SGBD como MySQL, PostgreSQL, etc. Por tanto usamos:
`airflow initdb`

En este momento ya tenemos AirFlow funcionando, ahora sólo falta revisar unos conceptos adicionales.

Inicialización del planificador

El planificador del flujo de trabajo de AirFlow monitoriza todas las tareas y todos los DAG, y activa las instancias de las tareas cuyas dependencias se han cumplido. Realmente lo que hace es instanciar un subproceso, que monitoriza y se mantiene sincronizado con una carpeta para todos los objetos DAG que pueda contener, y periódicamente (cada minuto más o menos o lo

configurado) recoge los resultados del análisis del DAG[2] e inspecciona las tareas activas para ver si pueden ser activadas.

El planificador está diseñado para funcionar como un servicio persistente en un entorno de producción, por lo tanto siempre se debe iniciar cada vez que se quieran ejecutar los flujos de trabajo. Para ponerlo en marcha, todo lo que es necesario hacer es ejecutar el planificador de AirFlow. Utilizará la configuración especificada en airflow.cfg:

```
airflow scheduler
```

Para ejecutar este servicio y poder consultar los mensajes de operación es preferible ejecutar la orden de inicio en una shell (ventana de consola) independiente de otra desde la que interacciones con AirFlow.

En las siguientes secciones explicamos cómo usar AirFlow mediante distintos ejemplos y desde varios puntos de vista con respecto al modo de interacción en gestión de los flujos de trabajo con la plataforma.

Inicialización del servicio web y línea de órdenes para de gestión de Flujos de trabajo

AirFlow puede ser gestionado desde la línea de órdenes (shell) o desde un interfaz web.

Para iniciar el servicio de gestión vía web usamos:

```
airflow webserver -p 8080
airflow webserver -p 8090
```

Hecho esto, abrimos un navegador y solicitamos el siguiente URL (desde el mismo equipo en que se está ejecutando el servicio):

http://localhost:8080
http://localhost:8090

y accedemos al sistema de gestión de AirFlow.

[2] Handbook of Graph Theory, Combinatorial Optimization, and Algorithms por Krishnaiyan “KT” Thulasiraman, Subramanian Arumugam, Andreas Brandstädt, Takao Nishizeki y Algorithmic Graph Theory por Alan Gibbons

Todas las gestiones que se hacen desde el entorno web pueden ser realizadas desde el interfaz de órdenes Shell (a través de órdenes que interpreta y ejecuta el programa `airflow`) como vemos aquí:

```
airflow list_dags
```

Muestra el conjunto de flujos de trabajo disponibles en el sistema.

```
airflow list_tasks <DAG_ID>
```

Muestra el conjunto de tareas de un flujo de trabajo dado su `DAG_ID`

```
airflow task_state <DAG_ID>  <TASK_ID> <execution_date>
```

Muestra el estado de un tarea de un Flujo de Trabajo dado su TASK_ID , el DAG_ID y el `execution_date`

```
airflow list_dag_runs <DAG_ID>
```

Muestra el conjunto ejecuciones de un flujo de trabajo dado su `DAG_ID`

```
airflow pause <DAG_ID>
```

Para un flujo de trabajo dado su `DAG_ID`

```
airflow unpause <DAG_ID>
```

Reinicia un un flujo de trabajo dado su `DAG_ID`

En las siguientes secciones pasamos a mostrar ejemplos detallados de cómo se construye de forma programática un flujo de trabajo, a través de diferentes ejemplos. El alumno debe realizar todas las tareas que se van indicando en para la creación de flujos de trabajo y verificando que son correctas al introducirlas en el planificador.

Creación de un flujo de trabajo en AirFlow

La creación de flujos de trabajo con AirFlow se realiza creando un fichero con código fuente Python donde se definen todos los aspectos clave un DAG en AirFlow. La estructura es la siguiente:

- Bibliotecas de AirFlow y DAG.
- Elementos de configuración del flujo de trabajo.

- Creación del DAG.
- Creación/definición de los operadores o tareas a realizar en el flujo de trabajo.
- Especificación de las dependencias (creación del grafo) ente tareas.

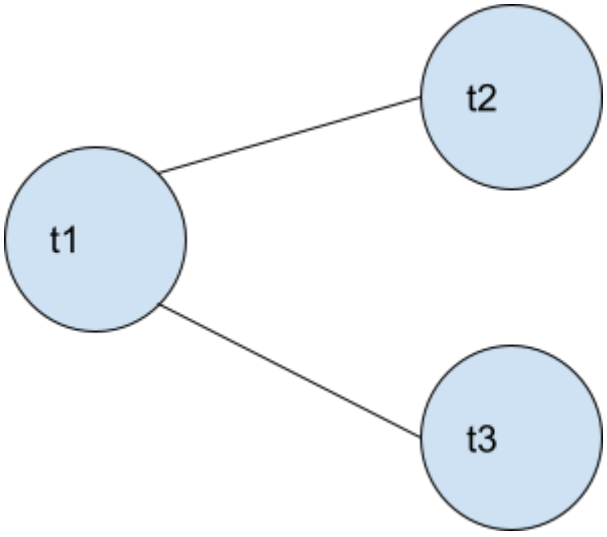
Este es un ejemplo completo en Python (llámalo `dag_practica2.py`), puedes descargarlo desde:

https://github.com/manuparra/MaterialCC2020/blob/master/airflow/p2example1.py

Aquí está el código:

```
from datetime import timedelta
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.utils.dates import days_ago

def validaAlgo(p1):
    ...
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'
}
#Inicialización del grafo DAG de tareas para el flujo de trabajo
dag = DAG(
    'practica2_ejemplo',
    default_args=default_args,
    description='Un grafo simple de tareas',
    schedule_interval=timedelta(days=1),
)
t0 = PythonOperator(
        task_id='print_date',
        python_callable=validaAlgo,
        op_kwargs={'p1': 10}, # validaAlgo(p1) → validaAlgo (op_kwargs)
        dag=dag,

)
```

```
# Operadores o tareas
# t1, t2 y t3 son ejemplos de tareas que son operadores bash, es decir comandos
t1 = BashOperator(
    task_id='print_date',
    bash_command='ls -l >> /tmp/mils.txt',
    dag=dag,
)

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

t3 = BashOperator(
    task_id='new_file',
    depends_on_past=False,
    bash_command='ps ax > /tmp/procs.txt',
    dag=dag,
)



t1 >> [t2,t3]
```



Una vez hecho esto cópialo a un fichero con extensión `.py` y guárdalo en `$HOME/airflow/dags/` que es el lugar donde AirFlow consulta los flujos de trabajo que existen en el sistema. En caso de no estar el directorio 'dags' dentro de la carpeta airflow.

Hecho esto esperamos a que AirFlow procese el flujo de trabajo y lo cargue en el planificador. Esta operación dura un par de minutos y depende de la configuración, por defecto cada 30 segundos AirFlow comprueba si hay nuevos Flujos creados. Por tanto podréis ver el nuevo flujo

creado desde la web http://localhost:8080 o bien utilizando la línea de órdenes con: `airflow list_dags` una vez sea procesado.

Explicación del flujo de trabajo:

Este flujo de trabajo se compone de 3 tareas `T1`, `T2`, y `T3` muy simples, ya que son operaciones del shell, es decir órdenes del shell.

`T1 → imprime la fecha actual → bash_command='date',`
`T2 → usa la orden 'sleep 5' para dormir el proceso → bash_command='sleep 5'`
`T3 → crea un fichero en /tmp/procs.txt con el contenido de 'ps' → bash_command='ps ax > /tmp/procs.txt'`

Para modelar las tareas se realiza de este modo:

`T1 >> [T2, T3]` significa que primero se ejecuta `T1` y luego `[T2, T3]` esperando `T2` y `T3` a que `T1` termine antes de comenzar con ambas.

Otra forma de modelarlas sería:

`T1 >> T2 >> T3` donde sería una secuencia de tareas empezando en T1 y terminando en T3

De este modo se puede modelar cualquier estructura de grafo DAG que se quiera con diferentes combinaciones de ese tipo.

Operadores o tipos de tareas en el flujo de trabajo

Un operador representa una única tarea, idealmente independiente. Los operadores determinan lo que realmente se ejecuta cuando el DAG se ejecuta. Son las unidades mínimas de ejecución. Mientras que los DAGs describen cómo ejecutar un flujo de trabajo, los operadores determinan lo que realmente se hace con una tarea.

Un operador describe una sola tarea en un flujo de trabajo. Los operadores suelen ser (aunque no siempre) atómicos, lo que significa que pueden valerse por sí mismos y no necesitan compartir recursos con otros operadores. El DAG se asegurará de que los operadores funcionen en el orden correcto; aparte de esas dependencias, los operadores suelen funcionar de forma independiente. De hecho, pueden funcionar en dos máquinas completamente diferentes.
Con AirFlow, existen los siguientes operadores para muchas tareas comunes, incluyendo:

- `BashOperator` - ejecuta un comando bash
- `PythonOperator` - llama a una función Python arbitraria

- `EmailOperator` - envía un correo electrónico
- `SimpleHttpOperator` - envía una petición HTTP
- `MySqlOperator`, SqliteOperator, PostgresOperator, MsSqlOperator, OracleOperator, JdbcOperator, etc. - ejecuta un comando SQL
- `Sensor` - un Operador que espera (polling) por un cierto tiempo, archivo, una fila de la base de datos, key en S3, etc...

Nosotros trabajaremos con `BashOperator` y `PythonOperator` durante las prácticas.

Operador Bash -- `BashOperator`

Permite ejecutar una orden del shell, por lo que cualquier mandato, macro, fichero de comandos o aplicación puede ser lanzada con este operador. Para definirlo dentro de un fichero de DAG usaremos:

```
Tarea1 = BashOperator(
    task_id='primera_tarea',
    bash_command='echo 1',
    dag=dag,
)
```

O algo más complejo:

```
BuscaFich1G = BashOperator(
    task_id='buscar_ficheros',
    bash_command='find /path -mtime +180 -size +1G',
    dag=dag,
)
```

Operador Python -- `PythonOperator`

Con el operador de Python podemos ejecutar cualquier servicio, código, función o biblioteca de python que esté disponible o hayamos desarrollado o codificado. Por tanto, es posible acceder a bases de datos, procesar datos con bibliotecas como `Pandas`, `ScikitLearn`, etc, o realizar cualquier operación o proceso desde código en Python.

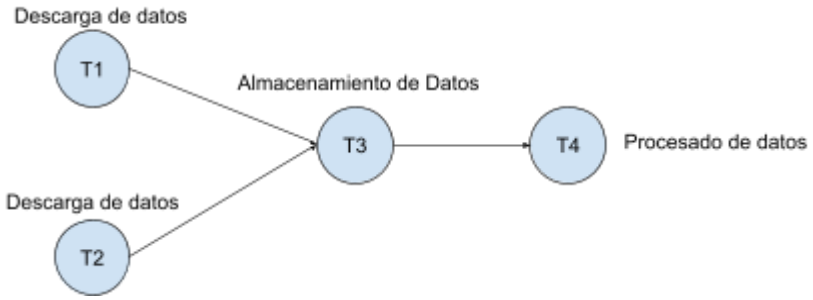
```
def imprime(ds, **kwargs):
    pprint(kwargs)
    print(ds)
    return 'Lo que devuelvas se imprime en los logs'

tarea1 = PythonOperator(
    task_id='imprime_contexto',
    provide_context=True,
```

```
        python_callable=imprime,
        op_kwargs={'parameter1': 10},
        dag=dag,
    )
```

Ejemplo completo de flujo de trabajo con múltiples operadores

Para este ejemplo vamos a mezclar todos los aspectos que hemos visto hasta el momento, por un lado los operadores y por otro las dependencias. Aquí está el código completo del ejemplo: https://github.com/manuparra/MaterialCC2020/blob/master/airflow/p2example2.py

Este flujo de trabajo hará lo siguiente:




```
T1 → BashOperator con orden: CURL con el que descargamos los datos
T2 → BashOperator con orden: CURL con el que descargamos los datos
T3 → BashOperator con orden: MV con el que movemos los datos a un
datastore.
T4 → PythonOperator, donde realizaremos una fusión de datos y envío de
resultados.
```

Aquí definimos cada tarea:

T1:

```
    DescargaDatos1 = BashOperator(
        task_id='descarga1',
        bash_command='curl -o /tmp/parte1.csv
    https://data.cityofnewyork.us/Transportation/2017-Yellow-Tax
    i-Trip-Data/biws-g3hs',
        dag=dag,
```

```
)
```

T2:

```
DescargaDatos2 = BashOperator(
    task_id='descarga2',
    bash_command='curl -o /tmp/parte2.csv
https://data.cityofnewyork.us/Transportation/2017-Yellow-Taxi-Trip-Data/biws-g1hs',
    dag=dag,
)
```

T3:

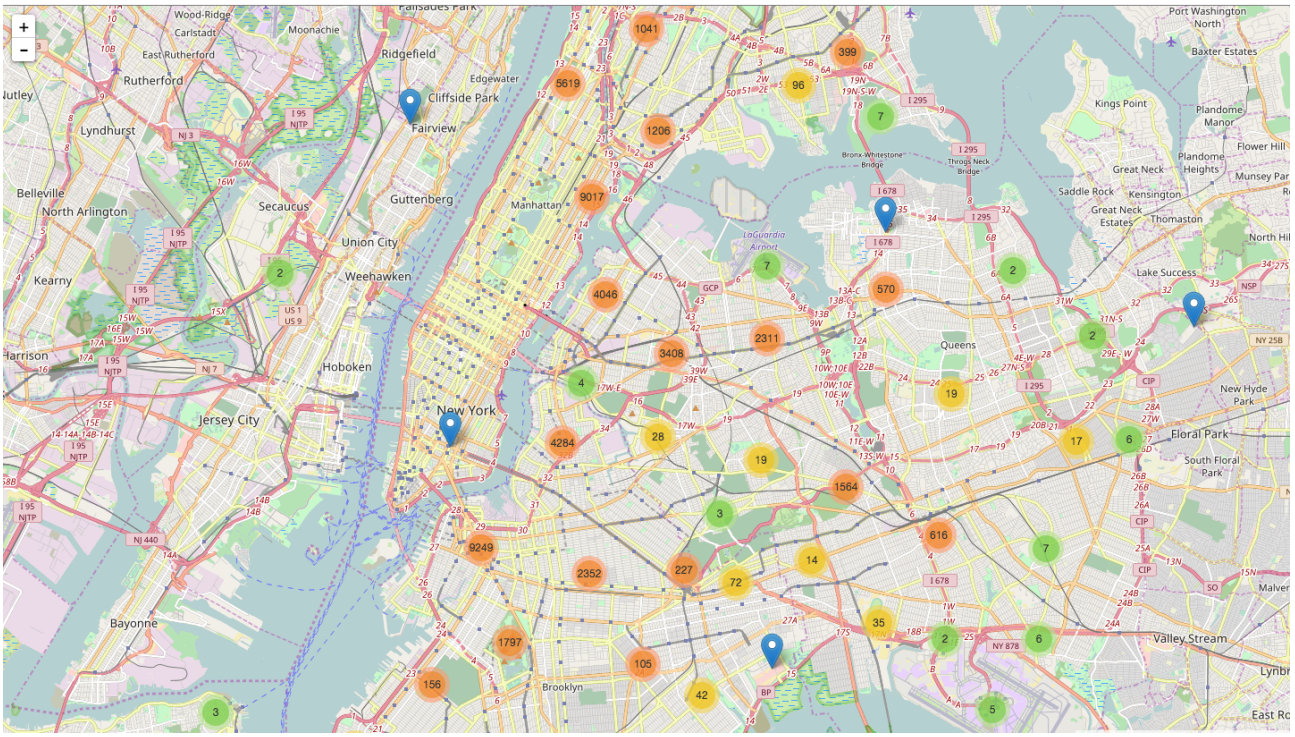
```
MoverDataStore = BashOperator(
    task_id='mover_aDataStore',
    bash_command='cat /tmp/parte1.csv <(tail +2
/tmp/parte2.csv) > /tmp/DataStore/output.csv',
    dag=dag,
)
```

T4:

```
# Estas bibliotecas deberían ser instaladas previamente
import pandas as pd
import numpy as np
import matplotlib.pylab as plt
from datetime import datetime
import os
import folium
from folium.plugins import MarkerCluster

def generaMapa():
       NY_COORDINATES = (40, -73)
       gdata = pd.read_csv('green_tripdata.csv')
       MAX_RECORDS = 1048576
       map_nyctaxi = folium.Map(location=NY_COORDINATES, zoom_start=9)
       marker_cluster = folium.MarkerCluster().add_to(map_nyctaxi)
       for each in gdata[0:MAX_RECORDS].iterrows():
          folium.Marker(
               location =
       [each[1]['Pickup_latitude'],each[1]['Pickup_longitude']],
                popup='picked here').add_to(marker_cluster)
       map_nyctaxi

CreaVisualizacion = PythonOperator(
    task_id='extrae_mapa',
    provide_context=True,
    python_callable=generaMapa,
```

```
            dag=dag,
        )
```

Esta operación T4 `(CreaVisualizacion)` generará un servicio web donde los datos ya están procesados y aparecerá el mapa siguiente en la salida de esta tarea final:

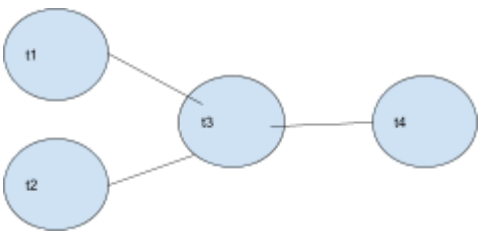


Por tanto con este flujo de trabajo, realizamos una toma de datos desde diferentes fuentes, luego las almacenamos, las fusionamos y realizamos una visualización de los datos.

Una vez definidas las tareas, hay que construir el grafo con las mismas operaciones del dibujo anterior, de modo que se indica:

```
[DescargaDatos1, DescargarDatos2] >> MoverDataStore >>
CrearVisualizacion
```

```
#       T1                  T2                      T3                  T4
```



Por tanto, poniendo todo en orden, tendremos que el fichero que almacena este flujo de trabajo es:

```
from datetime import timedelta
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.utils.dates import days_ago

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'
}

#Inicialización del grafo DAG de tareas para el flujo de trabajo
dag = DAG(
    'practica2_ejemplo_Mapas',
    default_args=default_args,
    description='Varias tareas mas elaboradas,
    schedule_interval=timedelta(days=1),
)

# Operadores o tareas
# t1, t2 t3 y t4 son ejemplos de tareas que son operadores bash, es decir comandos
DescargaDatos1 = BashOperator(
    task_id='descarga1',
    bash_command='curl -o /tmp/parte1.csv
https://data.cityofnewyork.us/Transportation/2017-Yellow-Taxi-Trip-Data/biws-g3hs',
    dag=dag,
)


DescargaDatos2 = BashOperator(
    task_id='descarga2',
    bash_command='curl -o /tmp/parte2.csv
https://data.cityofnewyork.us/Transportation/2017-Yellow-Taxi-Trip-Data/biws-g1hs',
    dag=dag,
)

MoverDataStore = BashOperator(
    task_id='mover_aDataStore',
    bash_command='cat /tmp/parte1.csv <(tail +2 /tmp/parte2.csv) > /tmp/DataStore/output.csv',
    dag=dag,
```

```
)

def generaMapa():
        NY_COORDINATES = (40, -73)
        gdata = pd.read_csv('green_tripdata.csv')
        MAX_RECORDS = 1048576
        map_nyctaxi = folium.Map(location=NY_COORDINATES, zoom_start=9)
        marker_cluster = folium.MarkerCluster().add_to(map_nyctaxi)
        for each in gdata[0:MAX_RECORDS].iterrows():
                folium.Marker(
                     location = [each[1]['Pickup_latitude'],each[1]['Pickup_longitude']],
                      popup='picked here').add_to(marker_cluster)
             map_nyctaxi

CreaVisualizacion = PythonOperator(
    task_id='extrae_mapa',
    provide_context=True,
    python_callable=generaMapa,
    dag=dag,
)


#Dependencias - Construcción del grafo DAG
[DescargaDatos1, DescargarDatos2] >> MoverDataStore >> CrearVisualizacion
```

Finalmente copiamos este fichero al directorio de despliegue de flujos de trabajo de AirFlow para que el planificador lo procese y comience el trabajo desarrollado.

Referencias

Apache AirFlow:
http://airflow.apache.org/
How to start with AirFlow:
http://airflow.apache.org/docs/stable/
AirFlow Git:
https://github.com/apache/airflow
Data Management with AirFlow:
https://towardsdatascience.com/why-apache-airflow-is-a-great-choice-for-managing-data-pipelines-48effcce3e41
AirFlow and AirBnB:
https://airbnb.io/projects/airflow/
AirFlow para Cloud Native:
https://www.datacouncil.ai/talks/running-airflow-reliably-with-kubernetes
Desarrollo de DAGs para flujos de trabajo con AirFlow:
http://michal.karzynski.pl/blog/2017/03/19/developing-workflows-with-apache-airflow/