ARMADO DE WORKFLOWS DINÁMICOS CON APACHE AIRFLOW

ESPECIALIZACIÓN EN CIENCIA DE DATOS - ITBA

Workshop de Big Data

13 de Abril de 2020

AGENDA

1. **Un (breve) repaso teórico de Airflow**
2. **DAGs y Ejercicios Guiados**
3. **Ejercicios Adicionales**

MUTT DATA

AIRFLOW RECAP

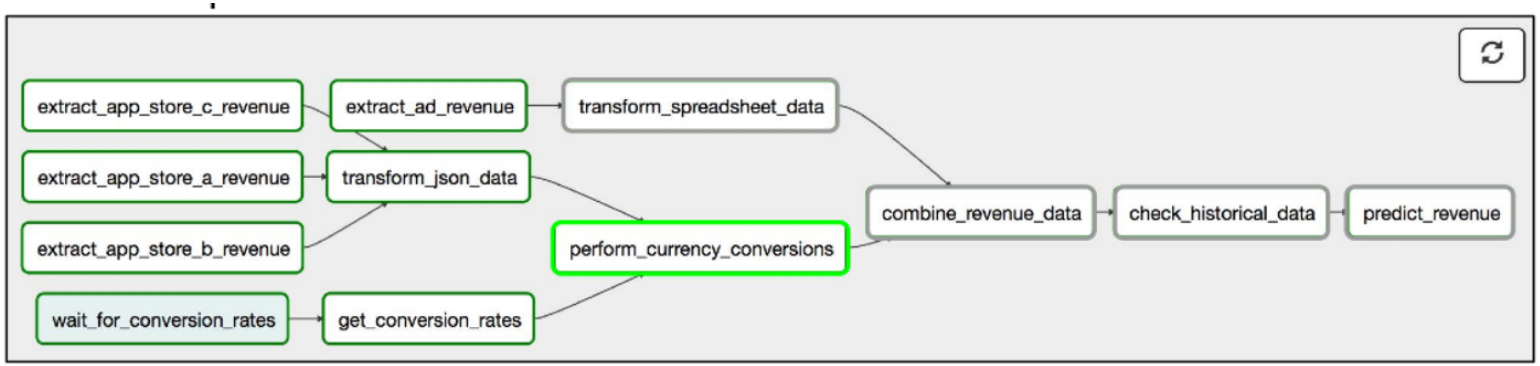
- ¿Qué es?
  - Plataforma para armar, monitorear y schedulear workflows de manera programática
  - Inicio en 2014 en Airbnb y Open Source desde mediados de 2015.
    - Utilizado por HBO, Twitter, ING, Paypal, Reddit, Yahoo y muchas más.
  - Los workflows se escriben en Python lo que permite abstraer comportamiento, testear y usar las herramientas de desarrollo para Python.
  - Tiene una UI que permite visualizar el estado de cada workflow, monitorear su progreso, métricas de cada tarea, realizar troubleshooting, manejo de credenciales y mucho más.
- ¿Qué ventajas tiene sobre otras herramientas "similares" (cron, oozie, etc)?
  - Tiene una forma eficiente de manejar reintentos de tareas procesamiento de datos pasados.
  - Provee vasta información para monitorear performance de tareas.
    - Por ejemplo visualizar si una tarea está tardando 3x más que lo habitual o si los datos del día 10 del mes actual se procesaron correctamente
  - Al tener workflows en python puro esto son fácilmente versionables.

MUTT DATA

TERMINOLOGÍA Y DAGS

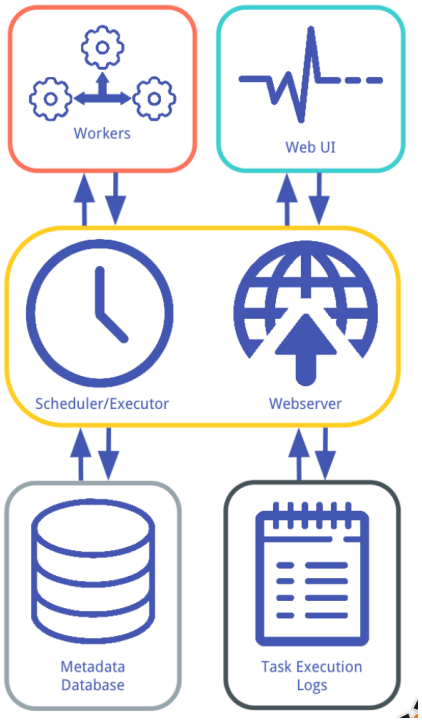
Algo de terminología:

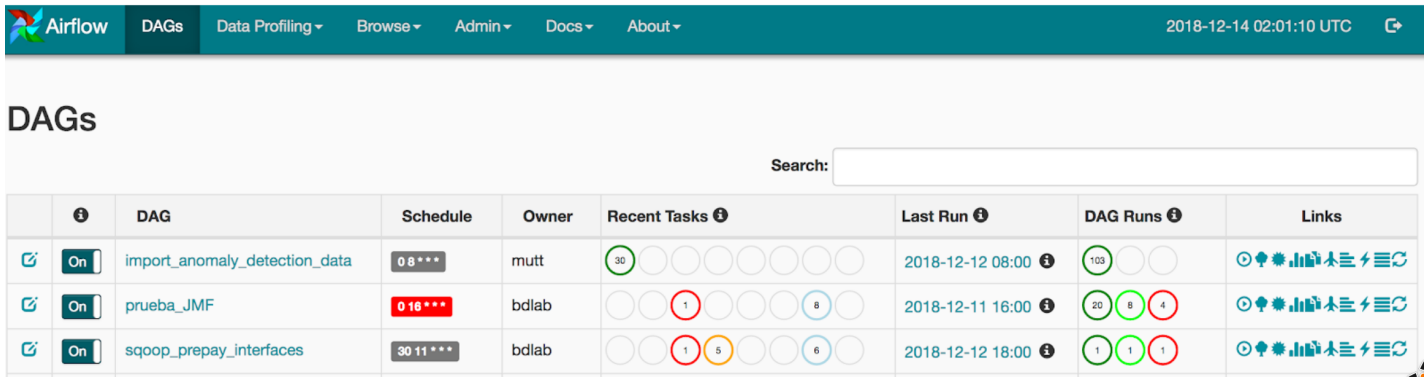
- *DAG*: grafo dirigido acíclico de las tareas que uno quiere correr (workflow)
  - El DAG es simplemente un archivo de python que define la estructura del grafo
- *Operator*: definen que se va a ejecutar (por ejemplo un comando bash, un insert a una tabla, etc)
- *Task*: instancia de un operador que en consecuencia define un nodo del grafo.
- *DAG run*: instancia de un DAG. Cuando un DAG inicia una corrida entonces Airflow orquesta la ejecución de los operadores respetando dependencias y asignando recursos
- *Task instance*: corrida particular de una tarea particular para una corrida particular del DAG para un periodo temporal particular.



MUTT DATA

ARQUITECTURA





- En particular
  - Metadata DB: guarda el estado de las tareas y workflows (DAGs)
  - Scheduler/Executor: scheduler monitorea en forma continua los DAGs y inicia las corridas de estos (las tareas son luego ejecutadas por un worker determinado por el Executor (cola de mensajes))
  - Webserver/Logs/UI: El webserver se comunica con la db y renderiza el estado de las tareas y logs en la UI.

MUTT DATA

AGENDA

1. Un (breve) repaso teórico de Airflow
2. **DAGs y Ejercicios Guiados**
3. Ejercicios Adicionales

MUTT DATA

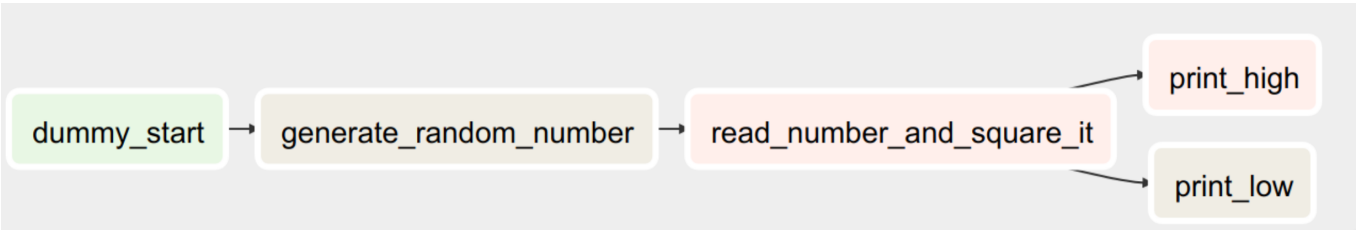
NUESTRO PRIMER DAG: BASH, DUMMY Y PYTHON OPERATORS

```python
from datetime import datetime
from pathlib import Path
from airflow.models import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import PythonOperator

STORE_DIR = Path(__file__).resolve().parent / 'tmp-files' / 'random-num'
Path.mkdir(STORE_DIR, exist_ok=True, parents=True)
bash_cmd = f"echo $(( ( RANDOM % 10 )  + 1 )) > {str(STORE_DIR / 'random_number.txt')}"

def _read_number_and_square(store_dir):
    fn = str(store_dir / 'random_number.txt')
    with open(fn, 'r') as f:
        n = f.readline()
    return int(n) ** 2

default_args = {'owner': 'pedro', 'retries': 0, 'start_date': datetime(2020, 4, 10)}
with DAG(
    'random_number', default_args=default_args, schedule_interval='0 4 * * *'
) as dag:
    dummy_start_task = DummyOperator(task_id=f'dummy_start')
    generate_random_number = BashOperator(
        task_id='generate_random_number', bash_command=bash_cmd)
    read_num_and_square = PythonOperator(
        task_id='read_number_and_square_it',
        python_callable=_read_number_and_square,
        op_args=[STORE_DIR])
    dummy_start_task >> generate_random_number >> read_num_and_square
```

MUTT DATA

EXERCISE 1: EXTENDING RANDOM NUMBER DAG

1. Using the random_number_dag.py file do the following steps in order to build the workflow seen in the figure
   - 1.a Add logging messages that signal a) which file is going to be read and b) what number was read from the file.
   - 1.b The filename is currently hardcoded as random_number.txt, change it to reflect the execution date i.e files should now be named something like 20200413.txt. *Hint*: you need to capture the execution date by adding a **context arg in the python callable.
   - 1.c Hard: convert the PythonOperator into a BranchPythonOperator that will be followed by another PythonOperator (BashOperator) task that prints HIGH (LOW) if the resulting squared numbered is greater (smaller) than 30.
   - 1.d Extra (take-home): Add logic to erase the created temporary files.



MUTT DATA

SEGUNDO DAG: INSERTANDO A DB CON XCOMS

- Realicemos los siguientes pasos antes de correr el dag `stocks_dag.py`:
  1. Instalemos una cli de sqlite llamada `litecli` con `pip`:

```
$> pip install litecli
$> ~/.local/bin/litecli /tmp/sqlite_default.db
```

  2. Chequeamos que `stocks_dag` aparece listado por `airflow list_dags`
     - Si ello no sucede podemos resettear la db con `airflow resetdb` (cuidado con este comando!)
  3. Activar el dag y dejarlo correr por el scheduler
  4. Completadas las tareas chequear que los datos se insertaron desde la cli de sqlite o a través del objeto `SqLiteClient` de python.

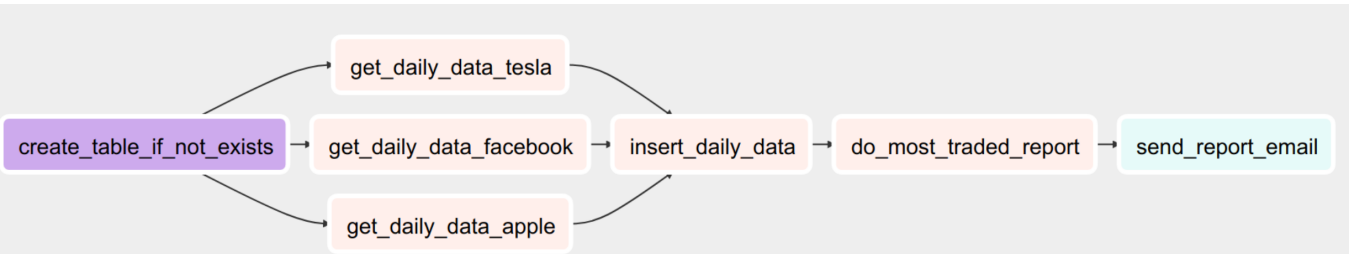
```
$> ~/.local/litecli /tmp/sqlite_default.db
/tmp/sqlite_default.db> select * from stocks_daily;
+------------+--------+--------------------+-----------+
| date       | symbol | avg_num_trades     | avg_price |
+------------+--------+--------------------+-----------+
| 2020-04-08 | aapl   | 29322.097916666666 | 264.3     |
| 2020-04-09 | aapl   | 28145.224305555555 | 267.385   |
| 2020-04-10 | aapl   | <null>             | <null>    |
```

- Notar algunos elementos importantes del dag
  - El *execution_date* es el período de tiempo en el cual vamos a procesar la data (distinto del período/momento de la corrida)
  - Si volvemos a correr uno de los tasks de inserción el dag rompe...

MUTT DATA

EXERCISE 2: SMALL STOCKS REPORT

1. Using the stock_dag.py file do the following steps in order to build the workflow seen in the figure
   - 1.a Modify the workflow to also get (and insert) data from Tesla (tsla) and Facebook (fb) stocks. Note: the easy way to do this is to add a loop inside the method that performs the request. The harder one (seen in the figure) is to have one branch per company spawning from the initial task which then reunite in the insert task.
   - 1.b Note that if you clear an insert task and re-run it you'll get an error (due to the unique constraint). Add some try/catch logic to gracefully avoid this error in the insert callback function.
     - 1.b.i You can alternatively let the task fail and add trigger_rule='all_done' to the successive task in order to trigger it regardless of the failure of the previous task.
   - 1.c Add a new task that reads (i.e queries) all records inserted in the present run and returns a string stating which was the company with the highest average trades per minute in that date.
   - 1.d Bonus: Create a final task that sends an email, using EmailOperator, of the report/insight obtained in 3. Note: for this exercise you will need to modify the smpt entry in the airflow.cfg file



MUTT DATA

AGENDA

1. Un (breve) repaso teórico de Airflow
2. DAGs y Ejercicios Guiados
3. **Ejercicios Adicionales**

MUTT DATA

DIGRESIÓN: POETRY Y PYTHON VIRTUAL ENVIRONMENTS

- *Poetry* es una herramienta que combina el manejador de paquetes *pip* con virtual environments (un directorio con una instalacion aislada de python)
  - ¿Cómo lo instalamos?

```
$> python -m pip install --user poetry
$> which poetry
/home/pedro/.local/bin/poetry
```

    - Si esto devuelve un error agregar `$HOME./local` a la variable de entorno `$PATH`.
  - ¿Cómo instalamos librerías con poetry?
    1. Crear un nuevo directorio airflow-101 y en el correr

```
$> poetry init
$> poetry add colorful
```

    2. Probemos ahora correr

```
$> python -c "import colorful"
ModuleNotFoundError: No module named 'colorful'
$> which python
/usr/bin/python
$> poetry run python -c "import colorful"
```

    3. Podemos *spawnear* una nueva shell dentro del venv:

```
$> poetry shell
(airflow-101) $> which python
/home/pedro/.cache/pypoetry/virtualenvs/
```

  - Notemos además que se creó un archivo `poetry.lock` que especifica los *requerimientos* de librerías para nuestra aplicación junto a la versión de python

MUTT DATA

INSTALACIÓN LOCAL DE APACHE AIRFLOW

- Ejecutar los siguientes comandos
  1. Instalar apache-airflow

```
$> poetry add apache-airflow
```

  2. Cambiar el directorio default de configuración de airflow (`/airflow`) setteando una variable de entorno `AIRFLOW_HOME` en el archivo `/airflow-101/.env`:

```
$> echo 'AIRFLOW_HOME="/home/pedro/airflow-cfg"' > .env
$> cat .env
AIRFLOW_HOME="/home/pedro/airflow-cfg"
```

     - Al correr `poetry shell` se debe luego *sourcear* este archivo con `source .env`.

  3. Abra una shell de Poetry y corra `airflow version`, esto creara la carpeta con los archivos de configuracion, logs, etc:

```
$> ls ~/airflow-cfg
$> poetry shell
(airflow-101) $> source .env && echo $AIRFLOW_HOME
/home/pedro/airflow-cfg
(airflow-101) $> airflow version
(airflow-101) $> ls ~/airflow-cfg
airflow.cfg airflow.db logs unittests.cfg
```

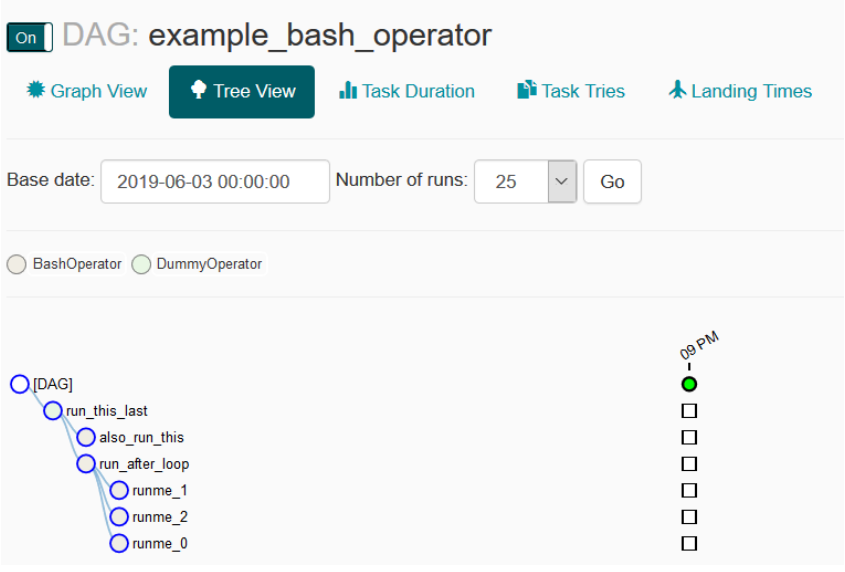
MUTT DATA

INSTALACIÓN LOCAL DE APACHE AIRFLOW II

- Continuar ejecutando los siguientes comandos
  4. Cree nuevo directorio `dags` (aquí guardaremos el archivo `titanic_dag.py` definido en el siguiente slide) y cambie el directorio de dags (`dags_folder`) en `/airflow-cfg/airflow.cfg` para que apunte a este nuevo directorio:

```
$> mkdir dags && cd dags
...
$> head ~/airflow-cfg/airflow.cfg | grep dags_folder
dags_folder = /home/pedro/airflow-101/dags
```

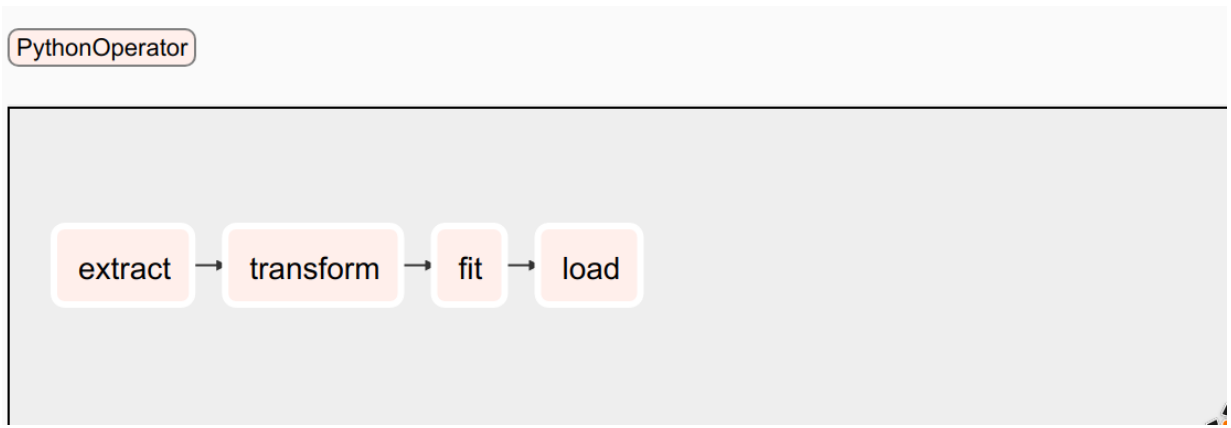
  5. Corra dentro de la shell de Poetry `airflow initdb` y luego `airflow list_dags` para listar los dags.
  6. Corra `airflow webserver` y abra `localhost:8080` en el browser
  7. Inspeccione los dags: para correrlos debe activarlos y luego prender el scheduler con `airflow scheduler`



MUTT DATA

EXERCISE 3 (MEDIUM): SCHEDULING TITANIC PREDICTION

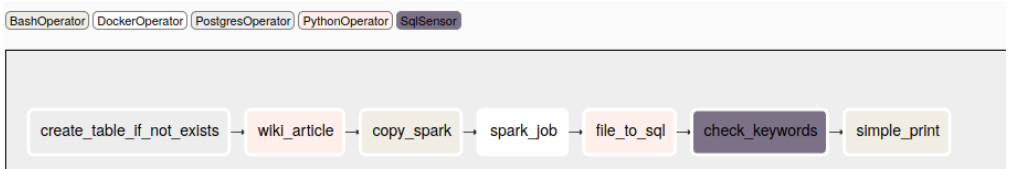
1. (Optional) Install a local version of Airflow following the previous slides.
2. The previous class we solved Kaggle's Titanic Disaster Prediction Challenge using an "ETFL" pipeline. In this exercise you are requested to schedule such pipeline with Airflow using what you've learnt in the past two exercises. The objective is to create a DAG from scratch that defines a workflow like the one shown in the image.
   - 2.a Create PythonOperators that perform the following sequential task
     - Extraction of Titanic data.
     - Trasnformation of such data.
     - Fit a Random Forest to the transformed data.
     - Load results into a csv file.
   - 2.a.i Bonus: In order to have some variability in runs between different dates hash the execution date as use that as the random number generator seed.
   - 2.a.ii Bonus: Load predictions into a DB of your choice.



MUTT DATA

EXERCISE 4 (HARD): DOCKEROPERATOR AND SQLSENSOR

1. Uncomment the commented code in `word_count_dag.py` and try to fix the resulting error when loading the dag. `Hint`: you need to install the python `docker` library.
2. The `word_count_dag.py` DAG launches a Spark job via `DockerOperator` to find out which is the most recurring word in a text file. This is then inserted in a SQL table. You can check out the Spark file at `spark_job.py`. Follow the next steps to build the workflow seen in the figure.
   - 2.a Create a connection to a valid Postgres database. You can do this through the Airflow UI via Admin -> Connections. Remember to properly set your connection ID in your operators.
   - 2.b The Spark job can read the environment variables `INPUT_FILE` and `OUTPUT_FILE` to know which file to count words from and which to store its result respectively. Use DockerOperator's environment variables to set these variables to an input file of your liking and an output file with a dynamic filename like: `<EXECUTION_DATE>.csv`. Also try to keep the working environment clean of files!
   - 2.c Create a PythonOperator to get a text file of your liking to use as `INPUT_FILE`. For example, checkout wikipedia's API.
   - 2.d Create a SqlSensor which pokes the database and checks if a certain keyword was the most used word. If it was, launch a new task.



MUTT DATA

REFERENCIAS

- Maxime Beauchemin (creador de Airflow)
  - The Rise of the Data Engineer
  - The Downfall of the Data Engineer
  - Functional Data Engineering
- Quizlet Beyond Cron Series
- Astronomer Apache Airflow Guides

MUTT DATA