Apache Airflow

Table of Content

Intro

This practical lesson will build the basic architecture to make the Big Data Project.

Episode I: Airflow

What is Airflow ?

Airflow is a platform created by the community to programmatically author, schedule and monitor workflows.

History

2014: It started at Airbnb as a solution to manage the company's increasingly complex workflows. It was created by Maxime Beauchemin.

2016, March: From the beginning, the project was made open source, becoming an Apache Incubator project in March 2016

2019: Become a top-level Apache Software Foundation project in January 2019.

2020, December 17th: the Airflow 2.0 version was released. Big breaking change with the first version.

Principles

Scalable

Airflow has a modular architecture and uses a message queue to orchestrate an arbitrary number of workers. Airflow is ready to scale to infinity.

Dynamic

Airflow pipelines are defined in Python, allowing for dynamic pipeline generation. This allows for writing code that instantiates pipelines dynamically.

Extensible

Easily define your own operators and extend libraries to fit the level of abstraction that suits your environment.

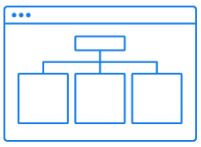
Elegant

Airflow pipelines are lean and explicit. Parametrization is built into its core using the powerful Jinja templating engine.

Features

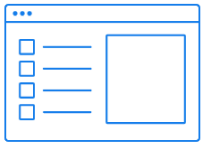
Pure Python

No more command-line or XML black-magic! Use standard Python features to create your workflows, including date time formats for scheduling and loops to dynamically generate tasks. This allows you to maintain full flexibility when building your workflows.



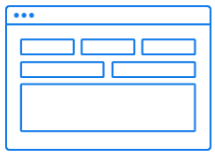
Robust Integrations

Airflow provides many plug-and-play operators that are ready to execute your tasks on Google Cloud Platform, Amazon Web Services, Microsoft Azure and many other third-party services. This makes Airflow easy to apply to current infrastructure and extend to next-gen technologies.



Easy to Use

Anyone with Python knowledge can deploy a workflow. Apache Airflow does not limit the scope of your pipelines; you can use it to build ML models, transfer data, manage your infrastructure, and more.



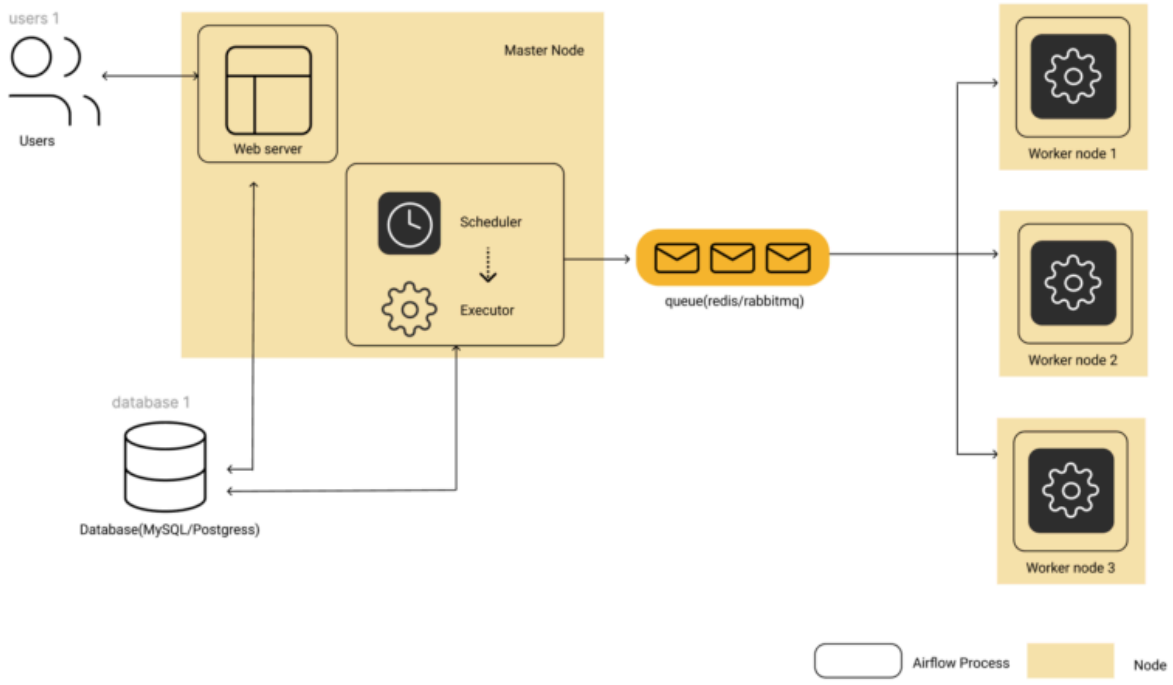
Useful UI

Monitor, schedule and manage your workflows via a robust and modern web application. No need to learn old, cron-like interfaces. You always have full insight into the status and logs of completed and ongoing tasks.

Open Source

Wherever you want to share your improvement you can do this by opening a PR. It's simple as that, no barriers, no prolonged procedures. Airflow has many active users who willingly share their experiences. Have any questions? Check out our buzzing slack.

Architecture



An Airflow installation generally consists of the following components:

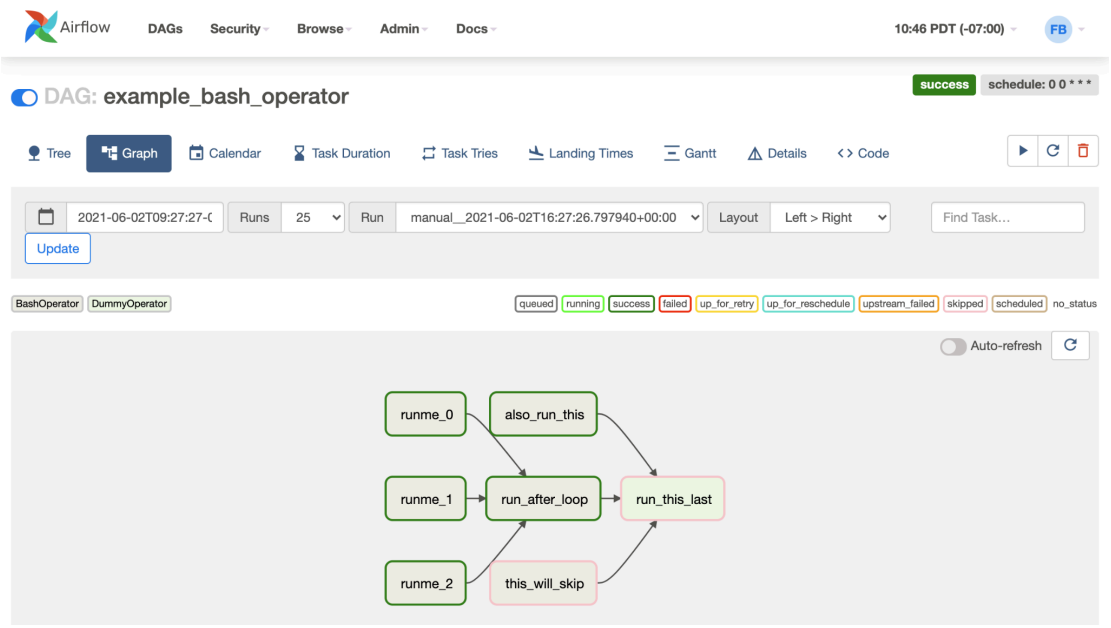
- A scheduler, which handles both triggering scheduled workflows, and submitting Tasks to the executor to run.
- An executor, which handles running tasks. In the default Airflow installation, this runs everything *inside* the scheduler, but most production-suitable executors actually push task execution out to *workers*.
- A *webserver*, which presents a handy user interface to inspect, trigger and debug the behavior of DAGs and tasks.
- A folder of *DAG files*, read by the scheduler and executor (and any workers the executor has)
- A *metadata database*, used by the scheduler, executor and web server to store state.

DAGs

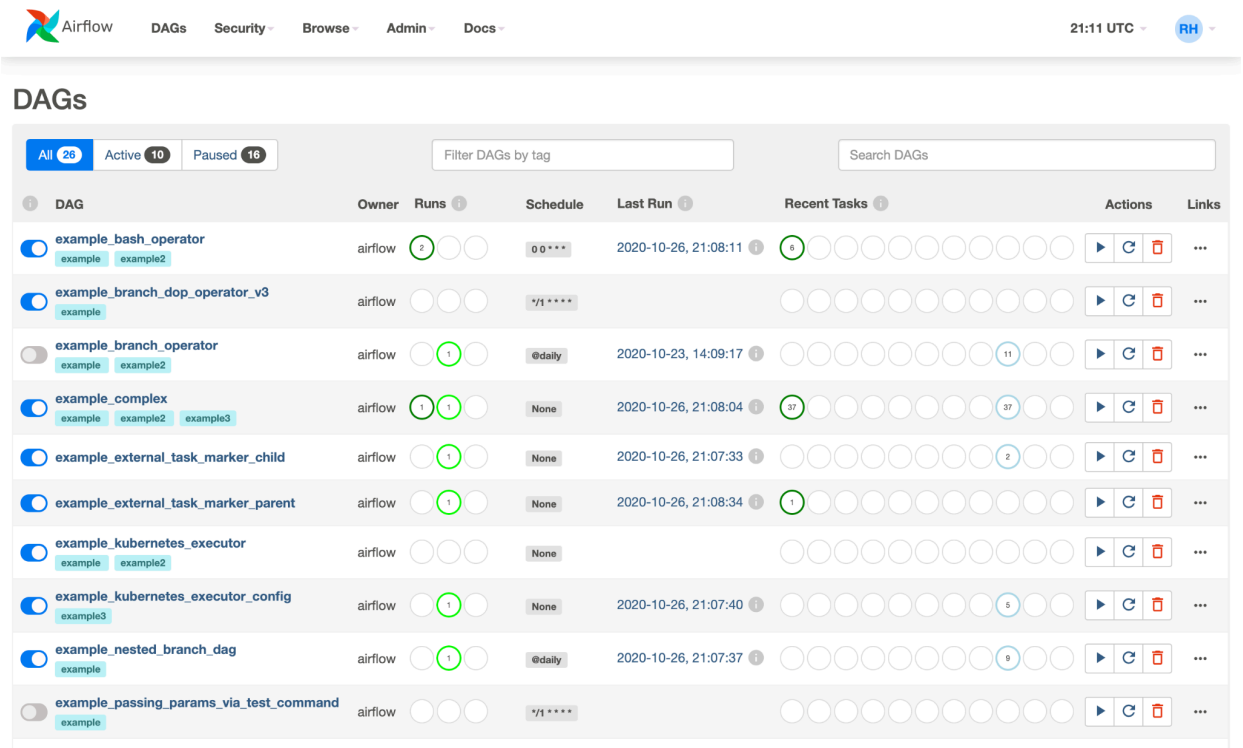
Airflow uses directed acyclic graphs (DAGs) to manage workflow orchestration. Tasks and dependencies are defined in Python and then Airflow manages the scheduling and execution. DAGs can be run either on a defined schedule (e.g. hourly or daily) or based on external event triggers (e.g. a file appearing in S3).

Previous DAG-based schedulers like Oozie and Azkaban tended to rely on multiple configuration files and file system trees to create a DAG, whereas in Airflow, DAGs can often be written in one Python file.

Here is an example of one DAG:



The home of Airflow UI provides an overview of all DAGs and the states of their previous DAG runs:



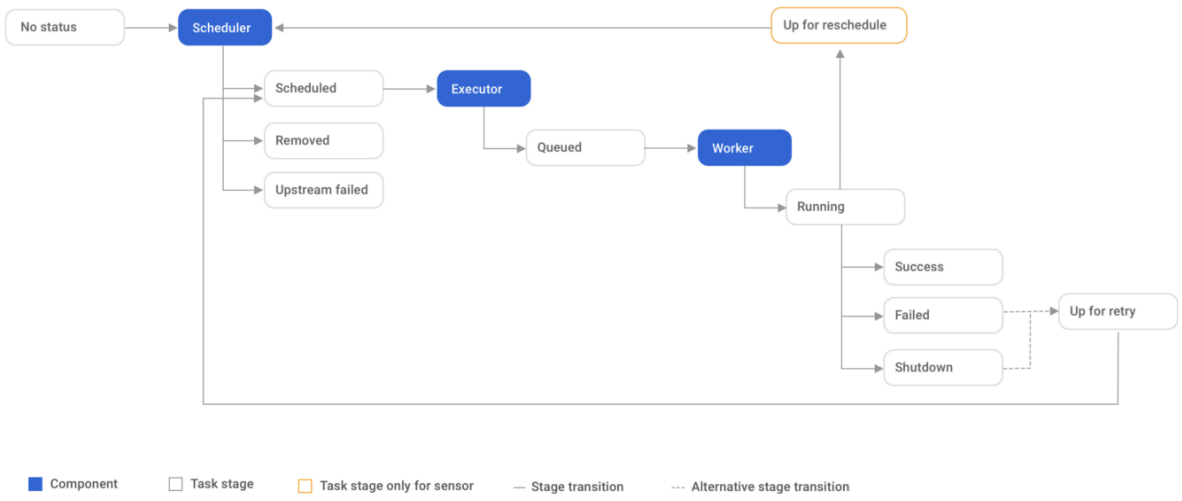
DAGs task

Each node of a DAG is a Task. A Task can be an:

- **Operator**: Launches some code to execute what's needed. It can be any Python custom code. Or it can launching a action on an external system:
    - Trigger a Spark Job
    - Trigger a Lambda function
    - Trigger a Batch located on Kubernetes
    - …
- **Sensor**: Wait for some external event / trigger.

DAGs task states

A Task can have several state, here is the state diagram of the tasks:



On the UI, by default they have these colors:

Documentation

Dig into Airflow official documentation to discover everything:
https://airflow.apache.org/docs/apache-airflow/stable/index.html

Episode II: Hands on Airflow

1. Download & Install PyCharm

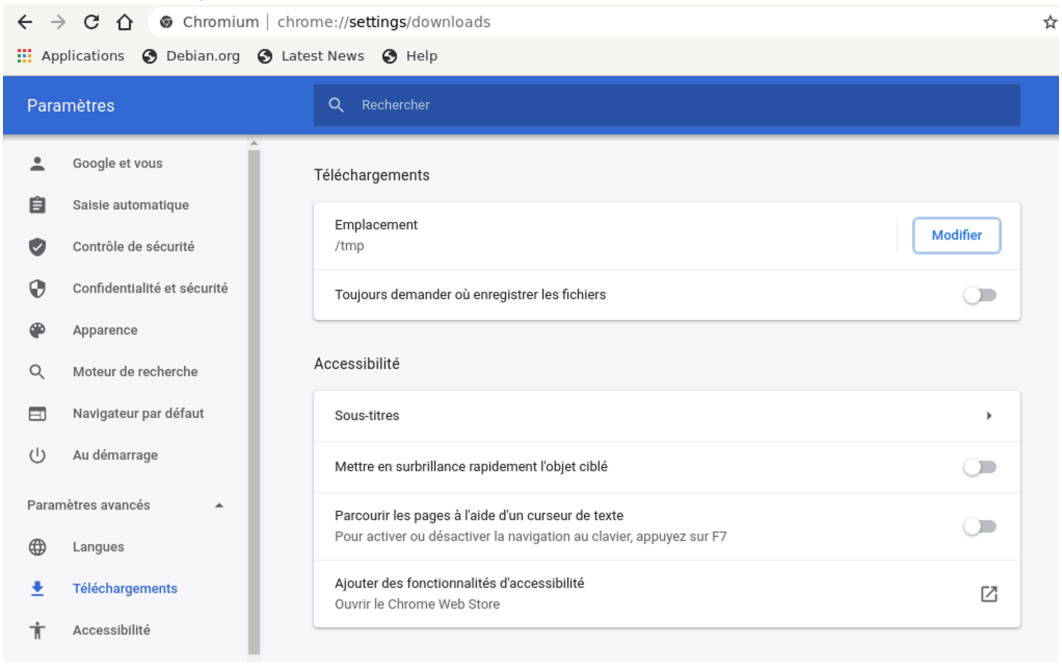
PyCharm is a great IDE for Python. This tutorial will use Pycharm for some specific explanations.

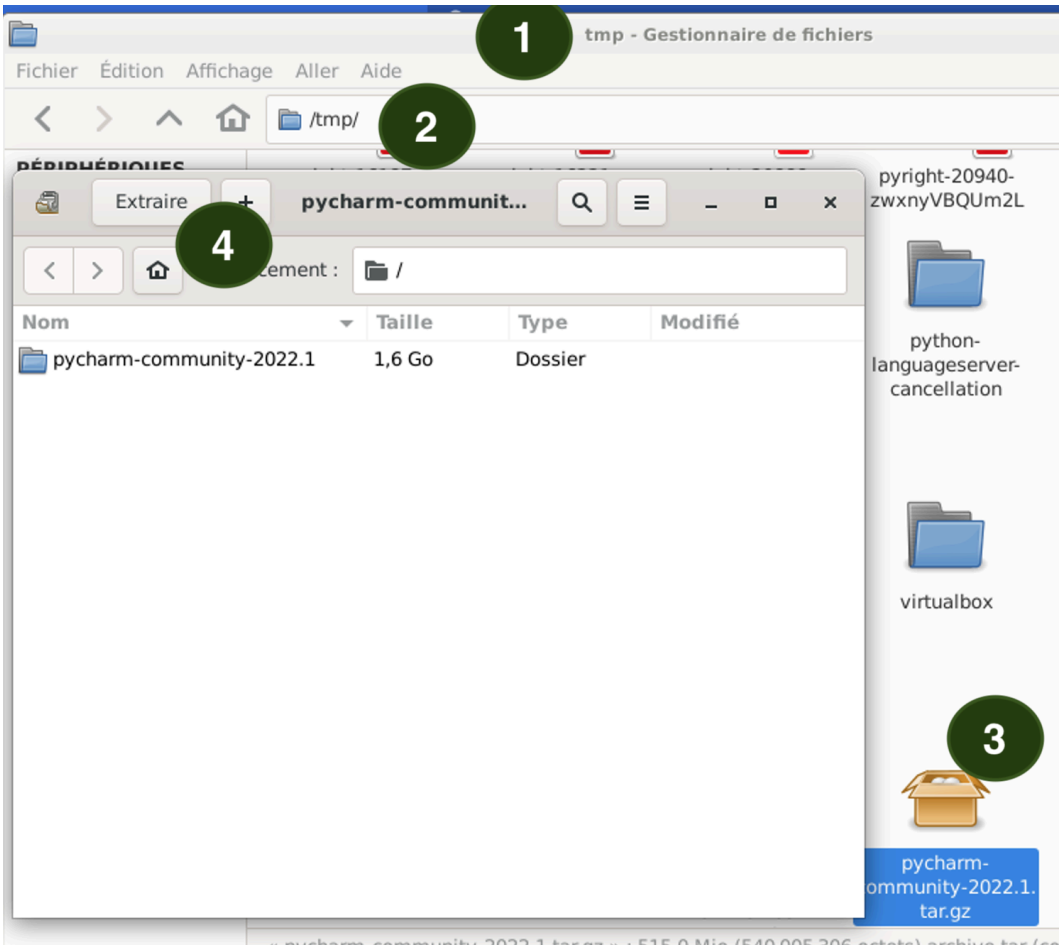
Note: If you are on Windows, it's mandatory to use Pycharm Professional. As you are a student, it's free to have a license here:

https://www.jetbrains.com/shop/eform/students

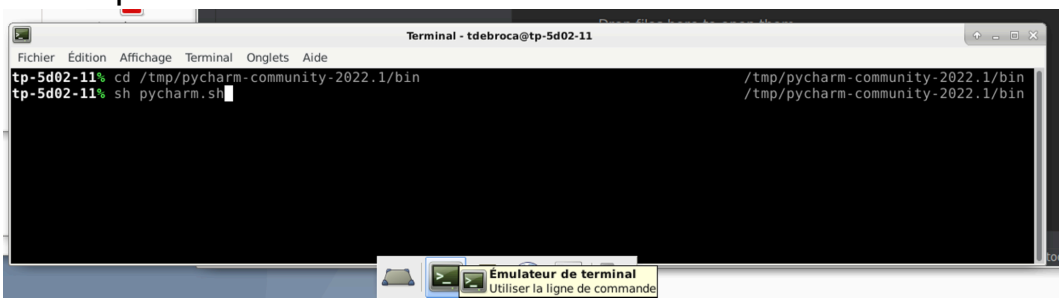
Install Pycharm from this link: https://www.jetbrains.com/fr-fr/pycharm/

Note: If you are comfortable with Python and venv, feel free to use whatever environment you like.

	*Special procedure on **Debian 10** with not enough place on Home:*
	If you don't have enough place on your home and more in /tmp folder, change the download folder here:  As shown in the next screenshot, follow these steps: 1. Open a "Gestionnaire de fichiers" 2. Change the folder to /tmp 3. Open pycharm-community-20XXXXXXX.ta.gz 4. Click on extract and extract in /tmp



Then open a Terminal:



Enter the following commands:

```
cd /tmp/pycharm-community-2022.1/bin

sh pycharm.sh
```

That's it, you have opened Pycharm on Debian 10

2.1 Install Airflow (Mac & Linux)

For Debian, check special “Install Airflow (Linux Debian)”
Pre-requisites you should have Python 3
(https://www.python.org/downloads/) .

Open a Terminal and launches:

```
# Airflow needs a home. `~/airflow` is the default, but you can put it
# somewhere else if you prefer (optional)
export AIRFLOW_HOME=~/airflow

pip3  install --upgrade pip

# Install Airflow using the constraints file
AIRFLOW_VERSION=2.7.3
PYTHON_VERSION="$(python3 --version | cut -d " " -f 2 | cut -d "." -f 1-2)"
# For example: 3.6
CONSTRAINT_URL="https://raw.githubusercontent.com/apache/airflow/constraints-${AIRFLOW_VERS
ION}/constraints-${PYTHON_VERSION}.txt"
# For example:
https://raw.githubusercontent.com/apache/airflow/constraints-2.7.3/constraints-3.6.txt
pip3 install "apache-airflow==${AIRFLOW_VERSION}" --constraint "${CONSTRAINT_URL}"
```

2.1 Install Airflow (With Windows)

Option A (using Airflowctl)

Use https://github.com/kaxil/airflowctl

```
pip install airflowctl
```

```
airflowctl init my_airflow_project --build-start
```

Option B (using Astro)

Use https://docs.astronomer.io/astro/cli/overview

```
brew install astro
astro dev init
astro dev start
```

Option C (classic)

InstallWindows Subsystem for Linux:
https://docs.microsoft.com/fr-fr/windows/wsl/install

Install an linux OS on Microsoft store :
https://www.microsoft.com/en-us/p/ubuntu/9pdxgncfsczv?activetab=pivot:overviewtab

3. Open linux console and write this:

```
sudo apt-get update

sudo apt install python3-venv
python3 -m venv airflow_venv
cd airflow_venv
source bin/activate
pip3 install "apache-airflow==2.7.3"

export PATH=/home/[USER]/.local/bin/:$PATH
```

Replace [USER] by your username

If you have error:
ImportError: cannot import name 'soft_unicode' from 'markupsafe' (/

Please do:
pip install markupsafe==2.0.1

Note: https://docs.astronomer.io/astro/cli/overview provides a CLI in order to install Airflow easily. This can be an option too.

2.1 Install Airflow (Linux Debian)

Launch:

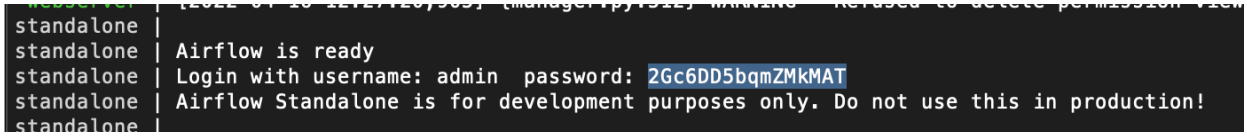
```
pip3 install --upgrade pip

export PYTHON_VERSION=3.7

export PATH=/cal/homes/$USER/.local/bin/:$PATH

pip3 install "apache-airflow==2.3.0" --constraint
https://raw.githubusercontent.com/apache/airflow/constraints-2.3.0/constraints-$PYTHON_VERSION.txt
```

3. Launch Airflow

Launch an Airflow as a standalone server:

```
airflow standalone
```

In the logs you should see something like this:



Copy the password and go to the UI:

http://localhost:8080/login

Enter the login: admin and the password you previously copied

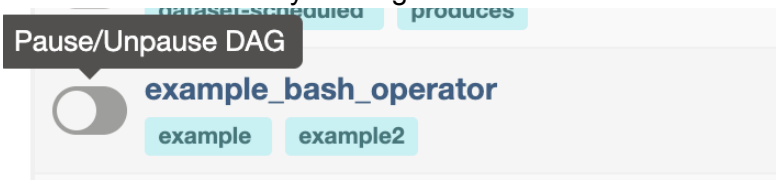
	This installation is only for test purposes as every component is only in 1 Python process and the database is a sqlite. In production you should have a real database like Postgresql and every composants should be in different nodes. Ideally, the executors nodes should be distributed and scalable.

	How to relaunch Airflow when it has been shut down ?
	
	Open your Linux Terminal (On Windows Ubuntu)
	And follow these commands:
	`cd airflow_venv`
	`source bin/activate`
	`export PATH=/home/[USER]/.local/bin/:$PATH`
	`airflow standalone`
	
	If you have a SQL error: If you don't care about the content of airflow.db => You can delete airflow.db / airflow.cfg and relaunch "airflow standalone"

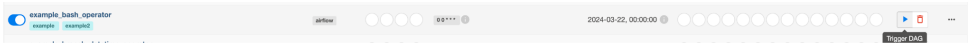
4. Try DAGs and discover UI

On Airflow UI, you should find many sample DAGs already installed.

Activate the first one by clicking on the radio button on the left:

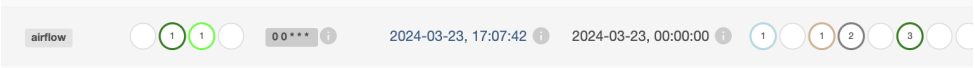


Then launch the DAG by clicking here on the Play button on the right:

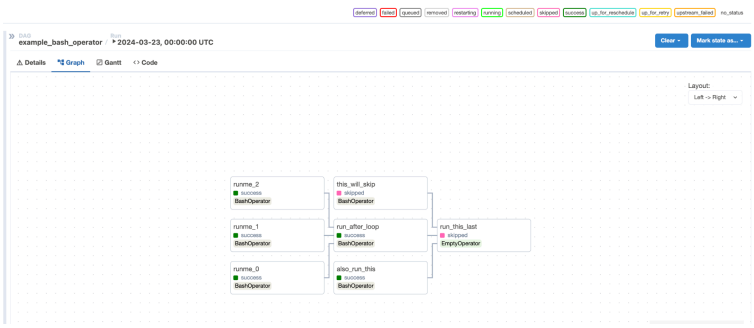


And click Trigger:

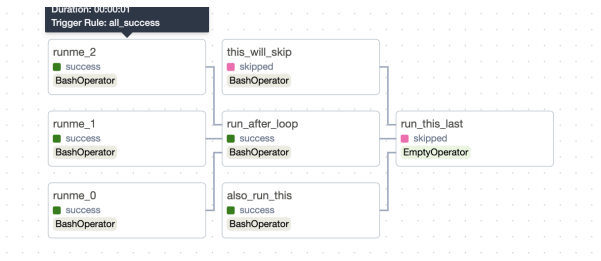
You can view your DAG running by clicking on the date here:



You can view the execution little by little:

	Here: - 1 Task is dark Green, that means it's a successful Task. - 1 Task is light Green, that means it's a running Task. - In brown is when Tasks are queued, that means once the scheduler sees a free slot to run a Task it will run it. - When tasks are white, they wait for the statuses of previous Task in order to be scheduled.

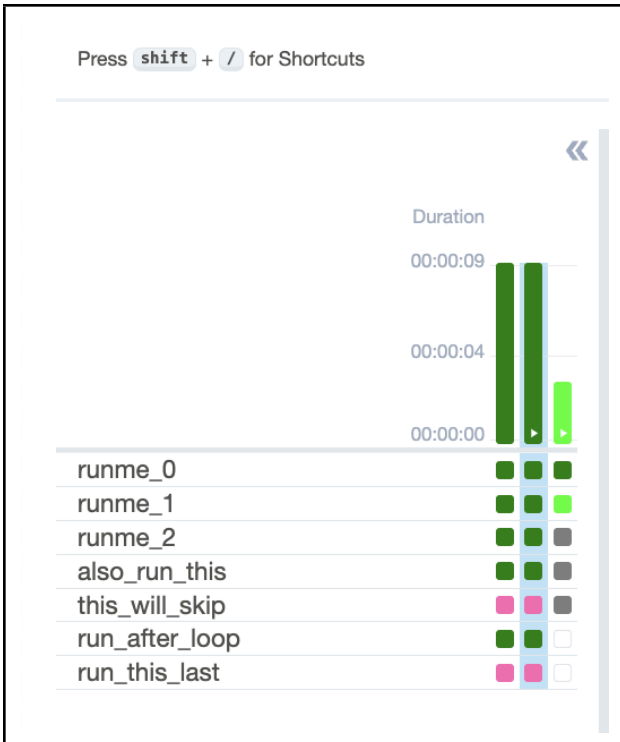
At the end, you have something like this:



- 5 Tasks are Successful.
- 2 Tasks are Skipped (because the code asked them to do so).

Launch this DAG **several times** and observe the Tasks launched on the UI.

Then, go on the Tree view:



You'll see this matrice:

- Each vertical column corresponds to '1 DAG run' (when you have clicked on "Trigger the DAG").
- Each row corresponds to the history of statuses of the Task.

Then, go on the "Code" view (click here: 

):

```
with DAG(
    dag_id="example_bash_operator",
    schedule="0 0 * * *",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    dagrun_timeout=datetime.timedelta(minutes=60),
    tags=["example", "example2"],
    params={"example_key": "example_value"},
) as dag:
```

Here we are creating the DAG.

- The dag_id is the name that will appear in the UI.
- It is scheduled to run as this CRON '0 0 * * *'. Which

	means, every day at midnight (https://crontab.guru/#0_0_*_*_*) - …
42 # [START howto_operator_bash] 43 run_this = BashOperator(44 task_id="run_after_loop", 45 bash_command="echo 1", 46)	Here we are creating 1 Task. It's a bash operator and it will launch the following command 'echo 1'.
62) 63 # [END howto_operator_bash_template] 64 also_run_this >> run_this_last	We are chaining 2 Tasks. 'run_this_last' will run once 'run_this' is completed with a **SUCCESS** status.
50 51 for i in range(3): 52 task = BashOperator(53 task_id=f"runme_{i}", 54 bash_command='echo "{{ task_instance_key_str }}" && sleep 1', 55) 56 task >> run_this	This Task accesses DAG variable via jinja templates (the {{ }})

Episode III: Hands on Airflow - Create your own DAG

1. Where can I create my DAG ?

When you launch airflow in standalone mode, the process has created a folder airflow in your $HOME_FOLDER.

In this folder, you should create a folder named 'dags'. This is where you will be able to create your DAG files.

```
cd ~/airflow
mkdir dags
```

It should look like this:

```
[thibautdebroca@Thibauts-MacBook-Pro:~/airflow$ pwd
/Users/thibautdebroca/airflow
[thibautdebroca@Thibauts-MacBook-Pro:~/airflow$ ls -lrt
total 1656
-rw-r--r--  1 thibautdebroca  staff   44590 Apr 16 12:26 airflow.cfg
-rw-r--r--  1 thibautdebroca  staff    4695 Apr 16 12:26 webserver_config.py
-rw-r--r--  1 thibautdebroca  staff      16 Apr 16 12:27 standalone_admin_password.txt
-rw-r--r--  1 thibautdebroca  staff       5 Apr 16 12:27 airflow-webserver.pid
drwxr-xr-x  6 thibautdebroca  staff     192 Apr 16 14:45 venv_pycharm
drwxr-xr-x  7 thibautdebroca  staff     224 Apr 16 14:55 logs
drwxr-xr-x  4 thibautdebroca  staff     128 Apr 16 15:05 dags
-rw-r--r--  1 thibautdebroca  staff  761856 Apr 16 15:10 airflow.db
```

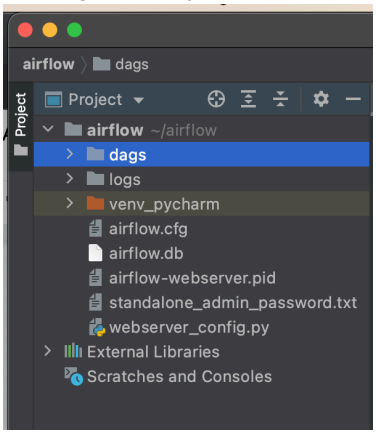
2. Create your First DAG

2.1. Create your First DAG - Create and pyCharm project and DAG

Open PyCharm and Create a new Project pointing to the folder:

Windows	Mac / Linux	Linux Debian
\\wsl$\kali-linux\home\{{USER}}\airflow	/home/{{USER}}/airflow	/cal/homes/$USER/airflow

You'll get a project like this:



Now you can create a Python file **my_first_dag.py** in the **dags** folder. And you can copy/paste this code:

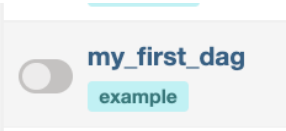
my_first_dag.py

```
from datetime import datetime, timedelta
```

```
from airflow import DAG
from airflow.operators.python import PythonOperator

with DAG(
        'my_first_dag',
        default_args={
            'depends_on_past': False,
            'email': ['airflow@example.com'],
            'email_on_failure': False,
            'email_on_retry': False,
            'retries': 1,
            'retry_delay': timedelta(minutes=5),
        },
        description='A first DAG',
        schedule_interval=None,
        start_date=datetime(2021, 1, 1),
        catchup=False,
        tags=['example'],
) as dag:
    dag.doc_md = """
        This is my first DAG in airflow.
        I can write documentation in Markdown here with **bold text** or __bold
text__.
    """


    def task1():
        print("Hello Airflow - This is Task 1")

    def task2():
        print("Hello Airflow - This is Task 2")

    t1 = PythonOperator(
        task_id='task1',
        python_callable=task1,
    )

    t2 = PythonOperator(
        task_id='task2',
        python_callable=task2
    )

    t1 >> t2
```

Note: You will see some errors in Pycharm (red underlined code), we'll see in 2 paragraphs, how to fix them.

Go back in the Airflow UI, you should see your DAG in the DAG list:

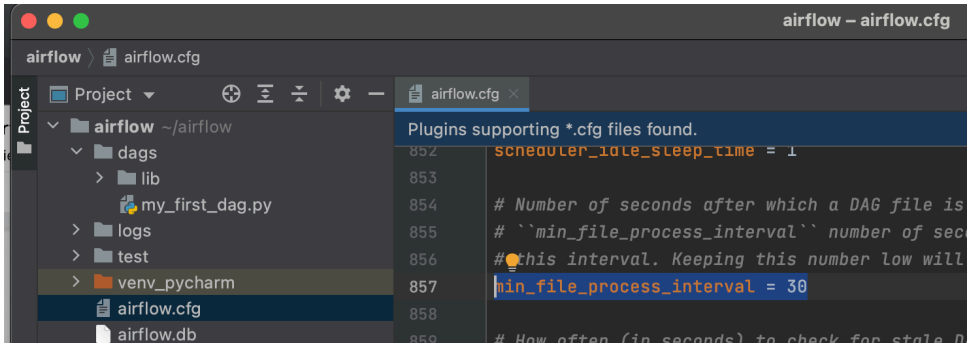


If it doesn't appear. Wait 30 seconds !

2.2. Create your First DAG - Lower DAG refresh

By default, airflow parse DAGs every 30secs.

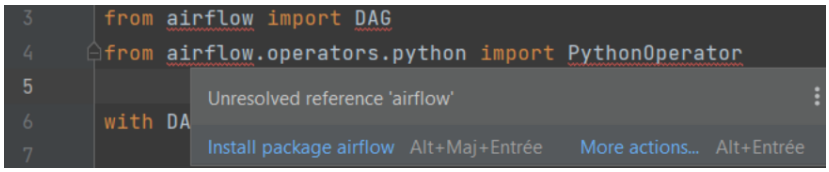
You could lower the refresh value interval, by going in airflow.cfg and change the value of `min_file_process_interval` to 5



Then you should restart your server.
Go back to the Terminal where there are airflow logs.
Type CTRL+C to stop the server.
And relaunch "airflow standalone" in the Terminal.

2.3. Create your First DAG - Remove Errors in PyCharm

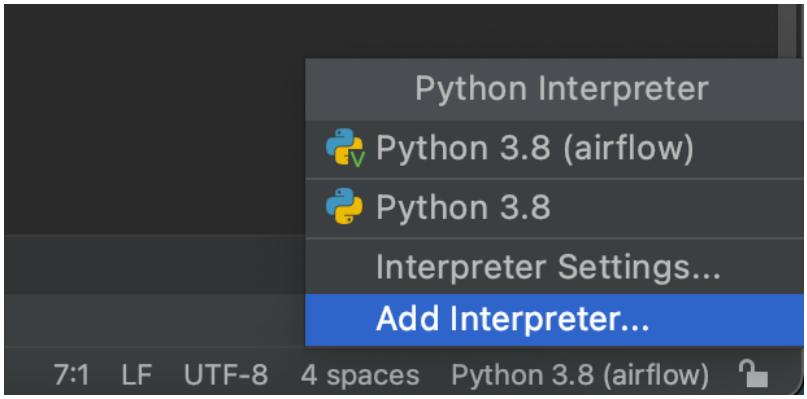
You might have red lines in import like this:



You should **always aim to fix all errors/warnings in PyCharm**, as it's very useful to have auto-completion and be sure your code works as much as possible.

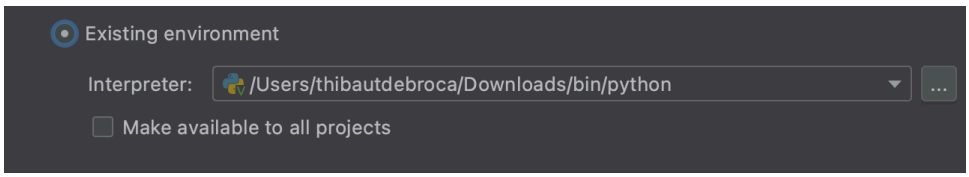
In that case, it's because there is not the package of airflow on this project. So you could install it in the Terminal of pycharm. But better, if you already have a venv of airflow, you should point to it in PyCharm. You can do it like this:

On the bottom right of the screen, click on “Python 3.X (....)”, it will open this popup:

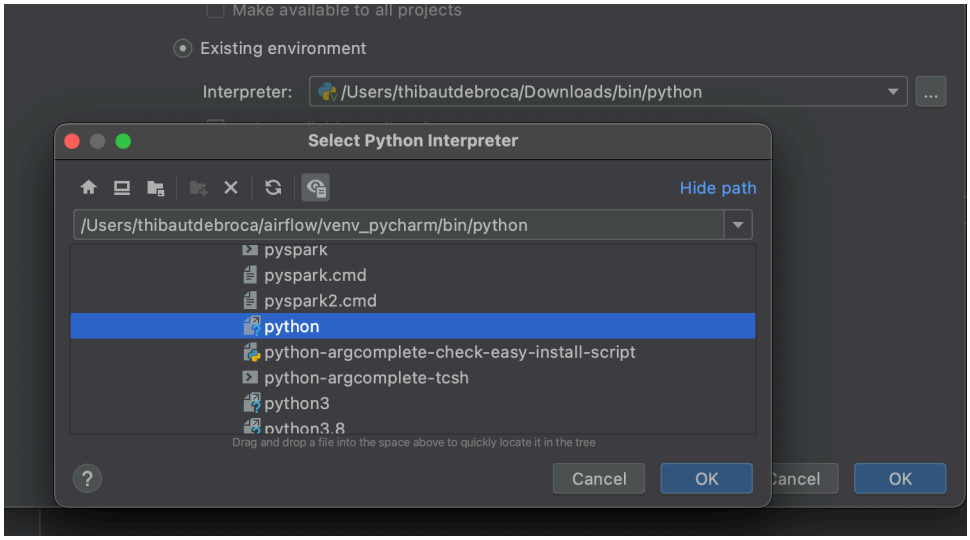


OPTION 1: On Mac/Linux:

Click on Add Interpreter. Then click on:



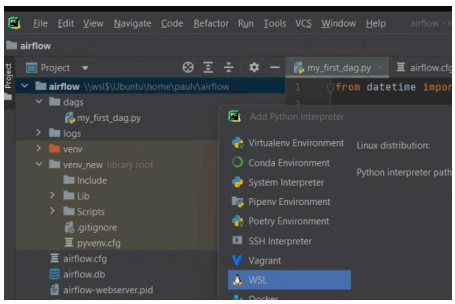
Then on Interpreter, point it to your already created Python venv:



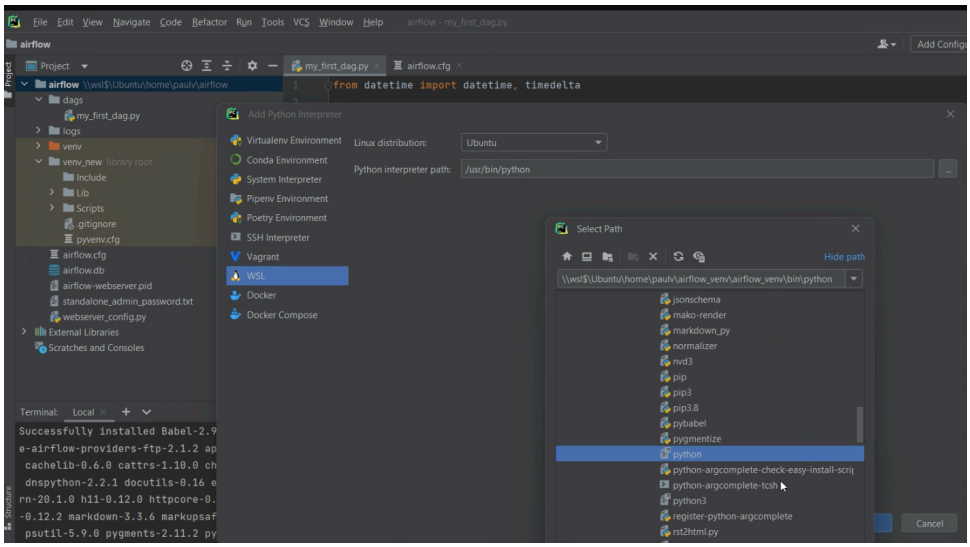
Which should look like this:
/Users/{{YOUR_HOME_USER}}/airflow_venv/bin/python

Click Ok and OK again.

OPTION 2: On WINDOWS with WSL:
You should click on the left “WSL”:

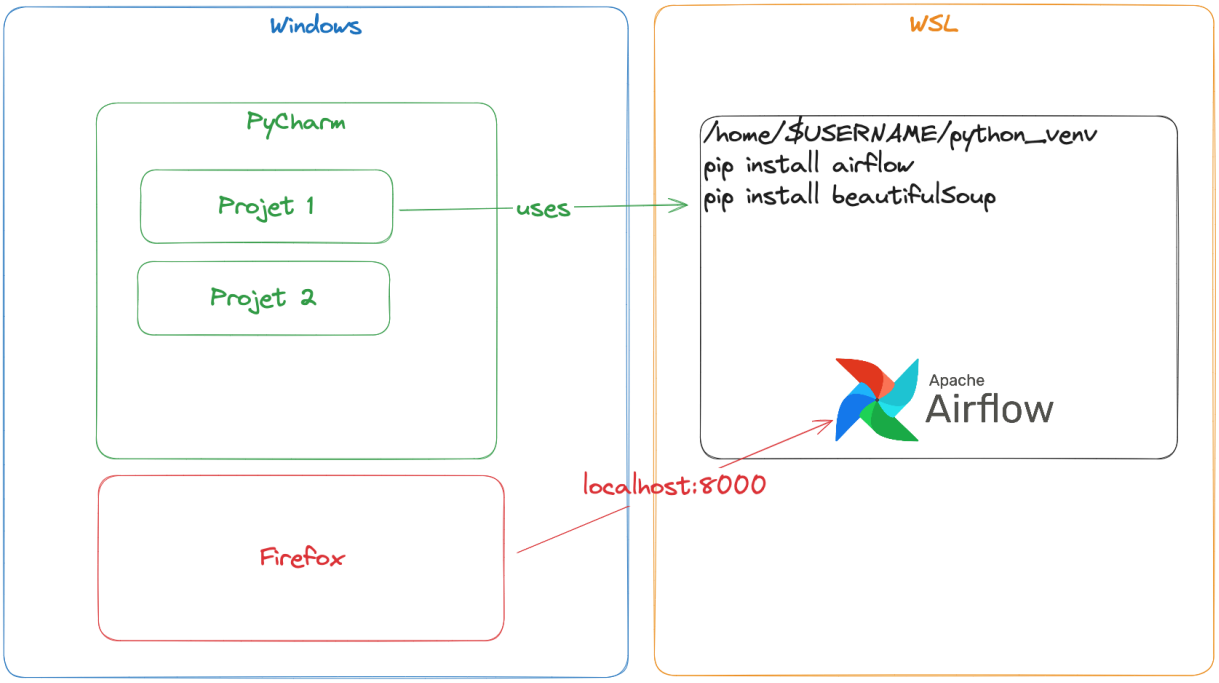


Then add your airflow_venv like this:



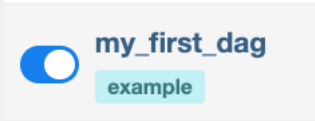
=> Thanks Paul Vidor for this one on Windows !

On Windows, here is a schema that summarizes

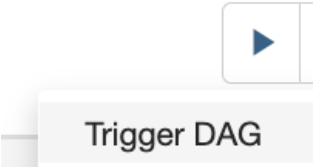


2.4. Create your First DAG - Launch DAG

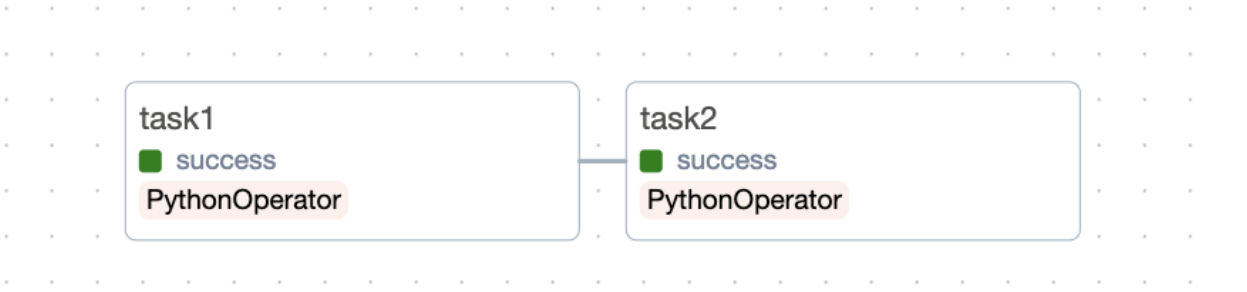
Unpause the DAG, by clicking on the checkbox:



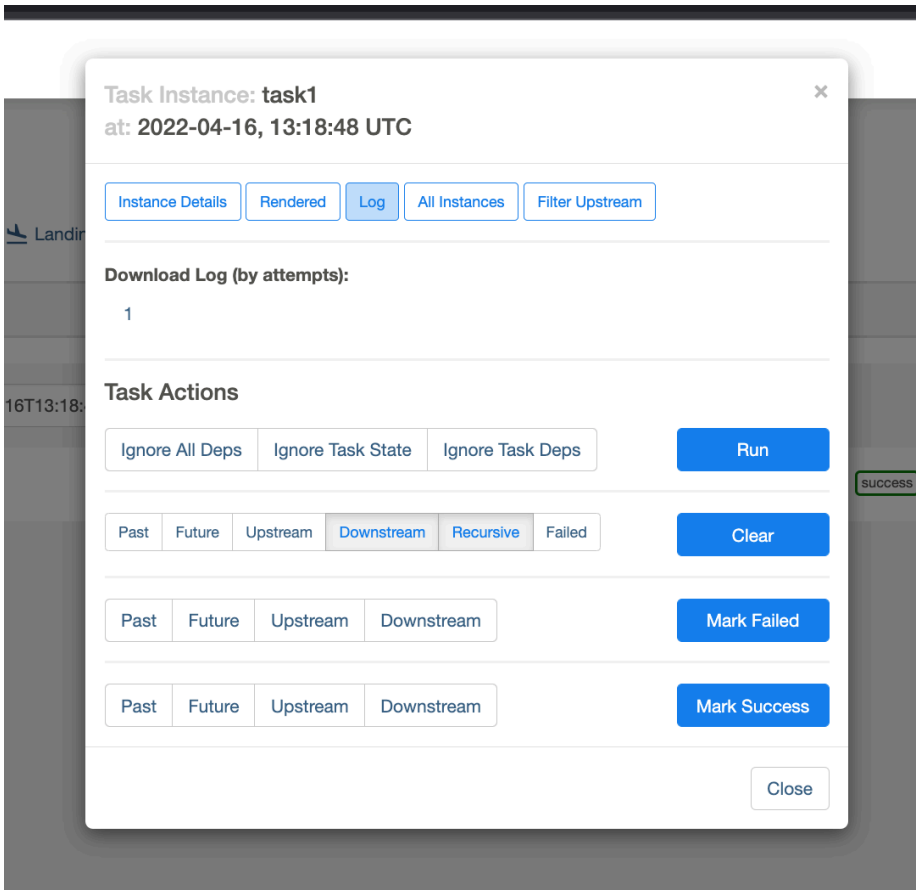
And launch the DAG, by clicking on ‘Trigger DAG’:



Go check the dag Run, at the end of the execution, both tasks should be in dark green, meaning Successful tasks:



Click on “Task 1”:



And click on “Log”. You should see this:

```
[2022-04-16, 15:18:59 UTC] {logging_mixin.py:109} INFO - Hello Airflow - This is Task 1
```

This log corresponds to your code, when you wrote:

```
def task1():
    print("Hello Airflow - This is Task 1")
```

Congrats you have run your first DAG

3. Parametrize Tasks

Change the code for this one:

my_first_dag.py

```
from datetime import datetime, timedelta

from airflow import DAG
from airflow.operators.python import PythonOperator

with DAG(
        'my_first_dag',
        default_args={
            'depends_on_past': False,
            'email': ['airflow@example.com'],
            'email_on_failure': False,
            'email_on_retry': False,
            'retries': 1,
            'retry_delay': timedelta(minutes=5),
        },
        description='A first DAG',
        schedule_interval=None,
        start_date=datetime(2021, 1, 1),
        catchup=False,
        tags=['example'],
) as dag:
    dag.doc_md = """
        This is my first DAG in airflow.
        I can write documentation in Markdown here with **bold text** or __bold
text__.
    """


    def launch_task(**kwargs):
        print("Hello Airflow - This is Task with param1:", kwargs['param1'],
              " and param2: ", kwargs['param2'])

    t1 = PythonOperator(
        task_id='task1',
        python_callable=launch_task,
        provide_context=True,
        op_kwargs={'param1': 'Task 1', 'param2': 'value1'}
    )

    t2 = PythonOperator(
        task_id='task2',
        python_callable=launch_task,
        provide_context=True,
        op_kwargs={'param1': 'Task 2', 'param2': 'value2'}
```

```
    )

    # t1 >> t2 can also be written like this:
    t1.set_downstream(t2)
```

Trigger the DAG, check the logs and explain what we have done.

```
AIRFLOW_CTX_DAG_RUN_ID=manual__2022-04-16T13:33:57.905364+00:00
[2022-04-16, 15:34:01 UTC] {logging_mixin.py:109} INFO - Hello Airflow - This is Task  Task1 value1
```

=> You should an enormous advantage here, that we have re-used the same method for different Tasks, and gave different parameters

3. Generate Tasks dynamically:

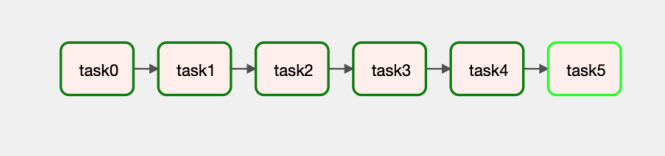
my_first_dag.py

```
from datetime import datetime, timedelta

from airflow import DAG
from airflow.operators.python import PythonOperator

with DAG(
        'my_first_dag',
        default_args={
            'depends_on_past': False,
            'email': ['airflow@example.com'],
            'email_on_failure': False,
            'email_on_retry': False,
            'retries': 1,
            'retry_delay': timedelta(seconds=15),
        },
        description='A first DAG',
        schedule_interval=None,
        start_date=datetime(2021, 1, 1),
        catchup=False,
        tags=['example'],
) as dag:
    dag.doc_md = """
        This is my first DAG in airflow.
        I can write documentation in Markdown here with **bold text** or __bold text__.
    """
```

```
    def launch_task(**kwargs):
        print("Hello Airflow - This is Task with task_number:",
kwargs['task_number'])

    tasks = []
    for i in range(6):
        task = PythonOperator(
            task_id='task' + str(i),
            python_callable=launch_task,
            provide_context=True,
            op_kwargs={'task_number': 'task' + str(i)}
        )
        tasks.append(task)
        if i > 0:
            tasks[i - 1].set_downstream(tasks[i])
```

Read the code, check the UI, launch the DAG, check the logs of the Tasks, and explain what happened.

You should have seen something like this:



This is the huge power of Airflow. As we use Python, we can dynamically create Tasks.

It's even possible to create DAGs dynamically, I'll let you try ;-).

4. Run DAG with a conf

Maybe you want to trigger the DAG with a specific configuration.

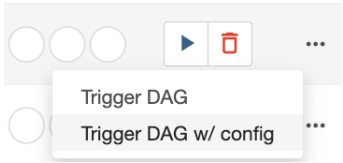
Change the method "launch_task" in the DAG to add the following lines:

my_first_dag.py

```
.....
```

```
    def launch_task(**kwargs):
        print("Hello Airflow - This is Task with task_number:",
kwargs['task_number'])
        print("kwargs", kwargs)
        print("kwargs['conf']", kwargs["conf"])
        print("kwargs['dag_run']", kwargs["dag_run"])
        print("kwargs['dag_run'].conf", kwargs["dag_run"].conf)


.....
```

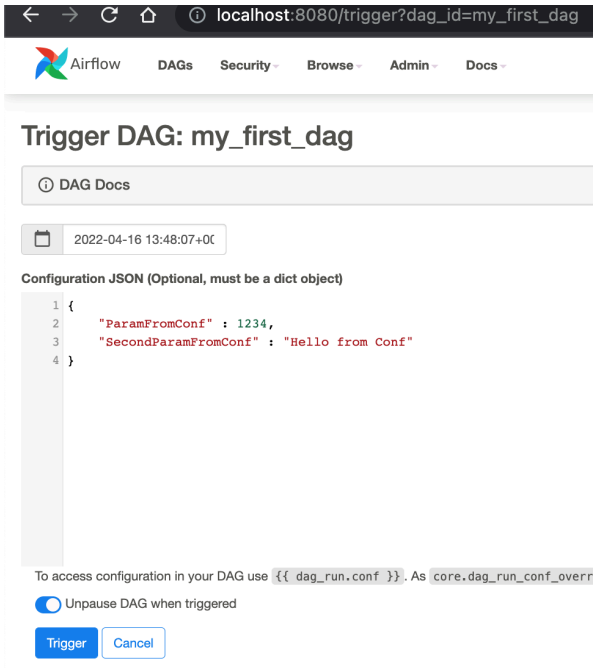
Click on “Trigger DAG with config”:



Copy this conf

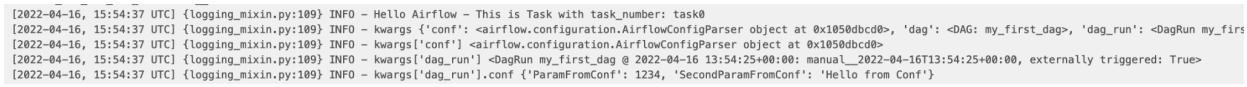
```
{
        "ParamFromConf" : 1234,
        "SecondParamFromConf" : "Hello from Conf"
}
```

here:



And click on “Trigger”.

Check the logs:



And explain which line of code accessed the conf.

This is very useful to make generic DAG and launch them with a different conf.

5. Access date from dag_run

Generally DAG are cronned. Maybe in a daily way. When you launch them, in your code, you probably want to access the date of the run dag.

Be careful, you should never access this date via the system date.
Example:

WRONG CODE

```
from datetime import date
today = date.today()
print("Today's date:", today)
```

You should get the date from the dag_run like this:

GOOD CODE

```
kwargs["dag_run"].execution_date
```

More here about the execution_date:
https://airflow.apache.org/docs/apache-airflow/stable/faq.html#faq-what-does-execution-date-mean

In fact there is many convenient variable that you can get from the dag_run:
https://airflow.apache.org/docs/apache-airflow/stable/templates-ref.html#templates-reference

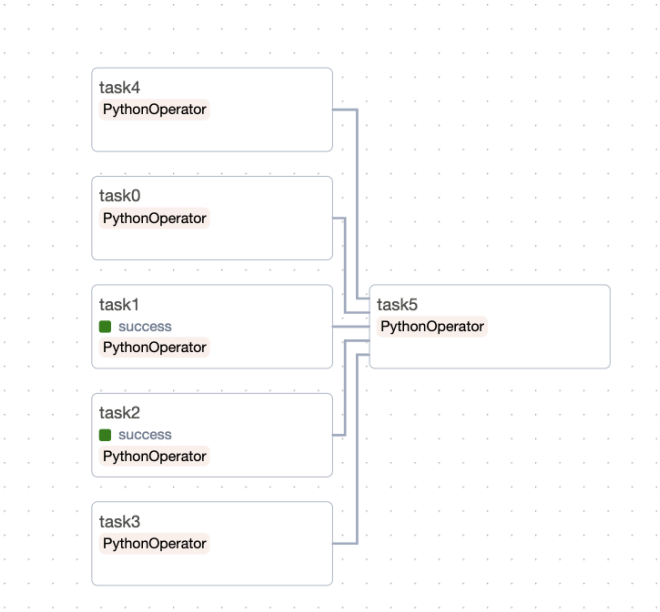
6. Join Tasks

If you want that 1 Task wait for several Tasks, you should do like this:

my_first_dag.py

```
from datetime import datetime, timedelta

from airflow import DAG
from airflow.operators.python import PythonOperator

with DAG(
        'my_first_dag',
        default_args={
            'depends_on_past': False,
            'email': ['airflow@example.com'],
            'email_on_failure': False,
            'email_on_retry': False,
            'retries': 1,
            'retry_delay': timedelta(seconds=15),
        },
        description='A first DAG',
        schedule_interval=None,
        start_date=datetime(2021, 1, 1),
        catchup=False,
        tags=['example'],
) as dag:
    dag.doc_md = """
        This is my first DAG in airflow.
        I can write documentation in Markdown here with **bold text** or __bold
text__.
    """


    def launch_task(**kwargs):
        print("Hello Airflow - This is Task with task_number:",
kwargs['task_number'])
        print("kwargs['dag_run']", kwargs["dag_run"].execution_date)

    tasks = []
    TASKS_COUNT = 6
    for i in range(TASKS_COUNT):
        task = PythonOperator(
            task_id='task' + str(i),
            python_callable=launch_task,
            provide_context=True,
            op_kwargs={'task_number': 'task' + str(i)}
        )
        tasks.append(task)

    # In python [-1] get the last element in an array
    last_task = tasks[-1]

    for i in range(TASKS_COUNT - 1):
        tasks[i].set_downstream(last_task)
```

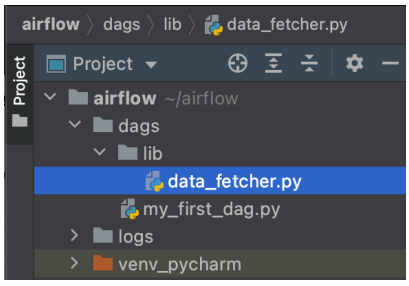
This will give you something like this:



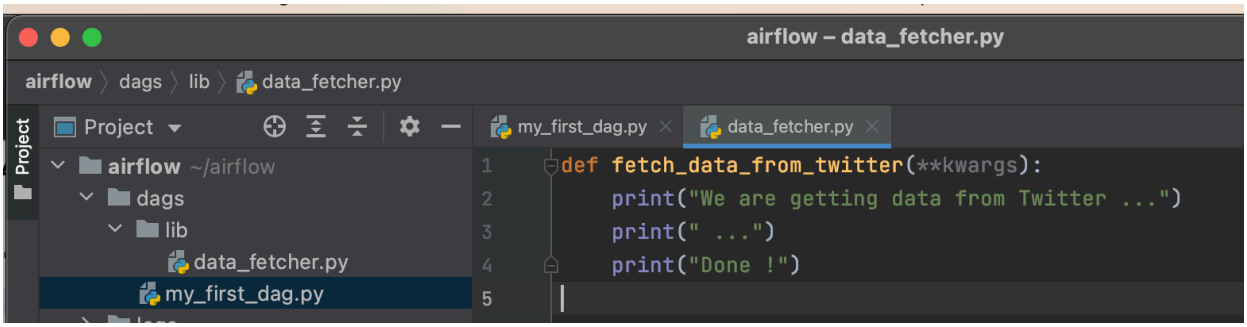
7. Call the Python method from another file.

Sometimes, the code you'll put inside an operator will be long. If you add it inside the Python code of the DAG, it will be hard to unit test unitarly because you would always need to launch the DAG on the UI. So let's see how to launch this Python outside of airflow, in another file.

In PyCharm, inside the dags folder, create a folder 'lib' and a file 'data_fetcher.py' inside:



In data_fetcher.py, add the following code:



Then, replace the code of my_first_dag.py by this one:

my_first_dag.py

```
from datetime import datetime, timedelta

from airflow import DAG
from airflow.operators.python import PythonOperator

from lib.data_fetcher import fetch_data_from_twitter

with DAG(
        'my_first_dag',
        default_args={
            'depends_on_past': False,
            'email': ['airflow@example.com'],
            'email_on_failure': False,
            'email_on_retry': False,
            'retries': 1,
            'retry_delay': timedelta(seconds=15),
        },
        description='A first DAG',
        schedule_interval=None,
        start_date=datetime(2021, 1, 1),
        catchup=False,
        tags=['example'],
) as dag:
    dag.doc_md = """
        This is my first DAG in airflow.
        I can write documentation in Markdown here with **bold text** or __bold
text__.
    """

    task = PythonOperator(
        task_id='fetch_data_from_twitter',
        python_callable=fetch_data_from_twitter,
        provide_context=True,
        op_kwargs={'task_number': 'task1'}
    )
```

Launch the DAG to test it.

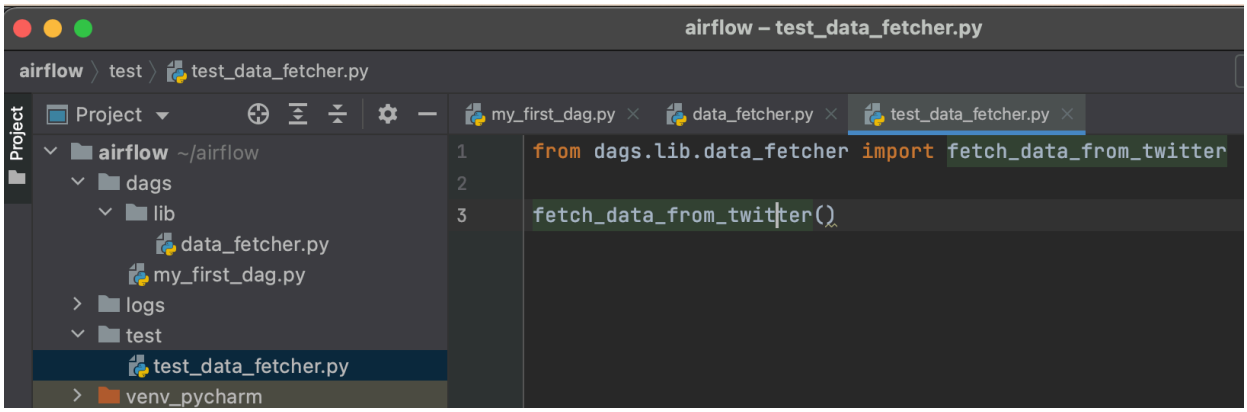
Now let's test this method outside of airflow.

Create a folder test in the root of your PyCharm project. Inside this folder, create a file named test_data_fetcher.py and add the following code:

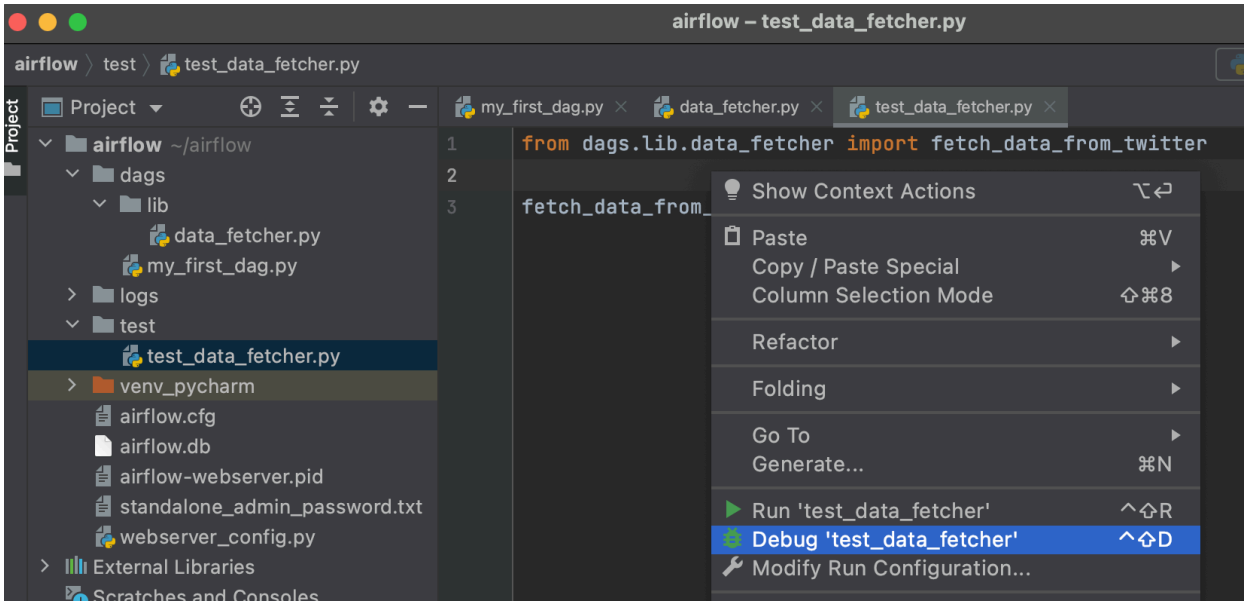
test_data_fetcher.py

```
from dags.lib.data_fetcher import fetch_data_from_twitter
fetch_data_from_twitter()
```

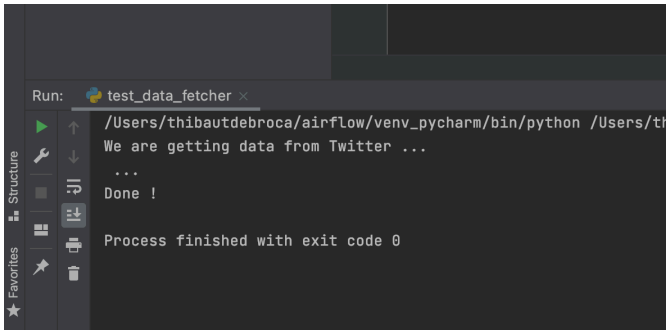
This will look like this:



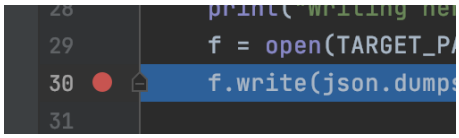
Right click on the file and launch:



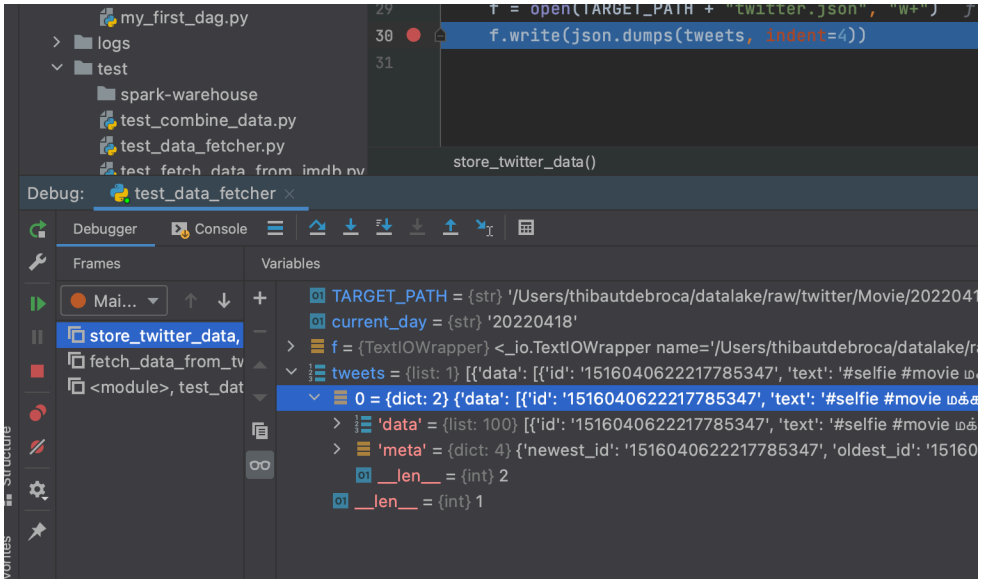
In the console, you should see:



The awesome thing is that you can add Debug breakpoint like this (click on the line number to add a breakpoint):



Which will permit you to see content inside variables:



Extremely useful for debugging.

8. Documentation

There are many many more things that we haven't talked about Airflow.

I strongly recommend to:

- read all documentation: https://airflow.apache.org/docs/apache-airflow/stable/index.html
- The best practices: https://airflow.apache.org/docs/apache-airflow/stable/best-practices.html

- Tips to debug a DAG: https://www.astronomer.io/blog/7-common-errors-to-check-when-debugging-airflow-dag/

NB: Link with GIT

In real production workflow, your code will be on GIT, and generally it's checkout automatically in the /dag folder of your airflow instance:

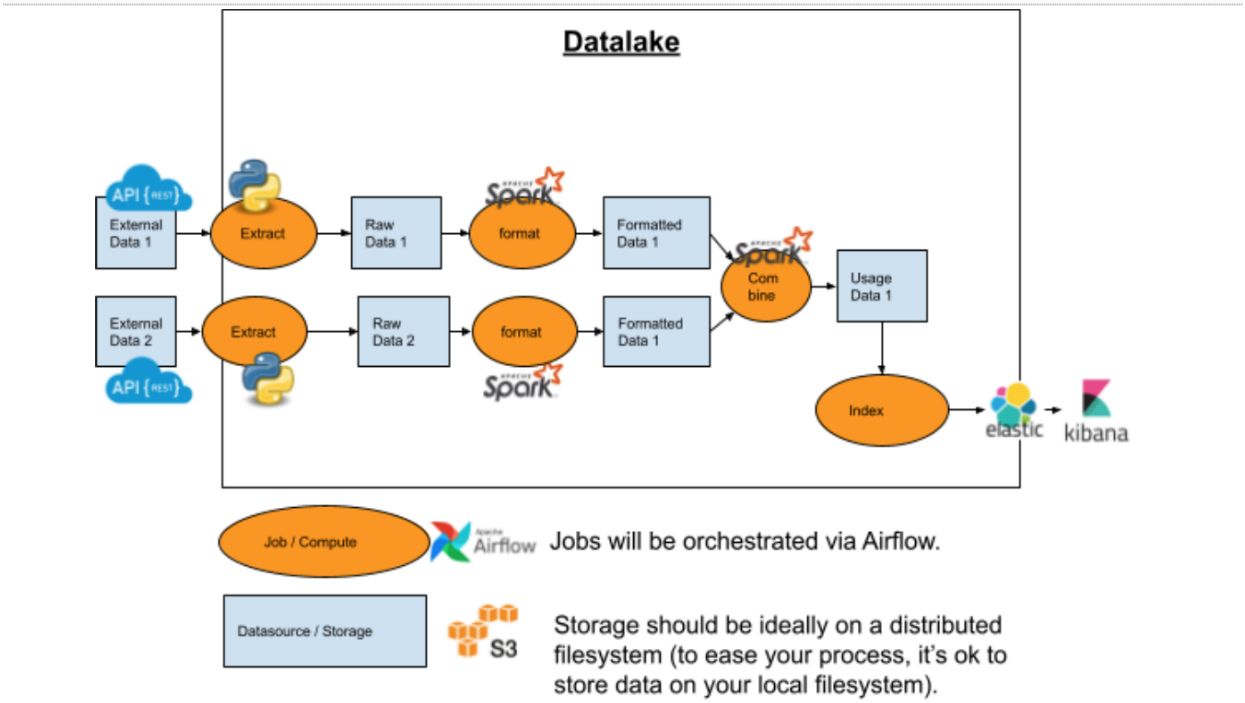
Episode IV: Big Data Project - Code skeleton

With all we have learnt before, let's start the Big Data Project.

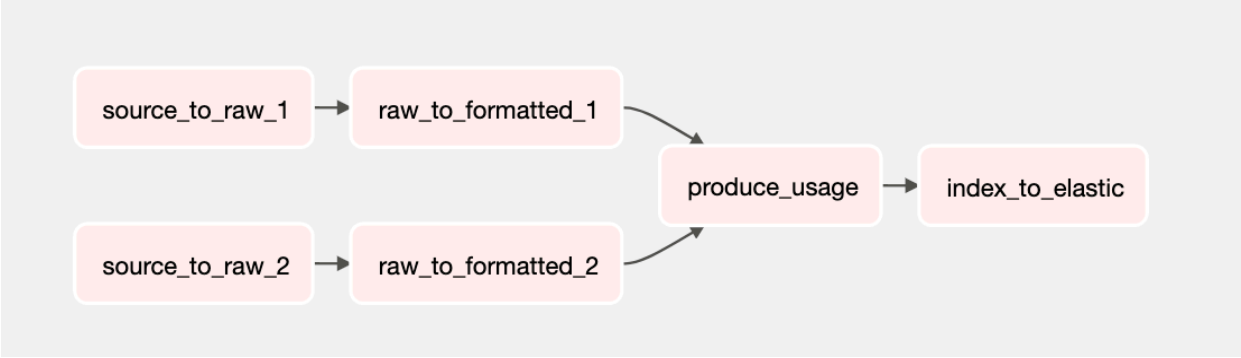
The Big Data project is located here.

1. Write a DAG that reflects architecture

As a reminder the architecture looks like this:



You should write a DAG will looks like this:



2. Prepare your files hierarchy / Data Lake structure

Your Datalake should respect a naming convention otherwise it will be a big mess.

Here is the naming convention you'll use:

Every path will be like this:

/{layer}/{group}/{TableName}/{date}/filename

- **layer** can be:
  - **raw** is the 1st step of the data, the data un-touched from the source system
  - **formatted** is the 2sd step of the data, the data prepared to be exploited by the Datalake / analytics system.
  - **usage** is the last step where
- **group**:
  - If in **raw** or formatted level, the group will be the name of the **source** system from which we have fetched the data (example: twitter, google, opendata …)
  - If in **usage** level, the group will be the name of the usage created (i.e.: recommandation, analytics …)
- **TableName**:
  - This folder is always the name of a data object: an object will a similar schema.
- **Date**:
  - The format should be YYYYMMDD, example 20221125 (25th November 2022)
- **Filename**:
  - Can be any name.

Full example:

/raw/
/raw/source1/
/raw/source1/DataEntity1/
/raw/source1/DataEntity1/20220416/
/raw/source1/DataEntity1/20220416/extract.json
/raw/source1/DataEntity2/
/raw/source1/DataEntity2/20220416/
/raw/source1/DataEntity2/20220416/extract.csv
/raw/source1/DataEntity2/20220417/
/raw/source1/DataEntity2/20220417/extract.csv
/raw/source2/DataEntity3/
/raw/source2/DataEntity3/20220416/
/raw/source2/DataEntity3/20220416/extract2.json

/formatted/
/formatted/source1/
/formatted/source1/DataEntity1/
/formatted/source1/DataEntity1/20220416/
/formatted/source1/DataEntity1/20220416/file1.parquet
/formatted/source1/DataEntity2/20220416/
/formatted/source1/DataEntity2/20220416/file2.parquet
/formatted/source2/DataEntity3/
/formatted/source2/DataEntity3/20220416/
/formatted/source2/DataEntity3/20220416/file3.parquet

/usage/
/usage/my_usage1/
/usage/my_usage1/DataEntityABC/
/usage/my_usage1/DataEntityABC/20220416/
/usage/my_usage1/DataEntityABC/20220416/fileABC1.parquet
/usage/my_usage1/DataEntityABC/20220417/
/usage/my_usage1/DataEntityABC/20220417/fileABC2.parquet

Inside a folder like /{layer}/{group}/{TableName}/
You should have only files with the same schema. That means if you want to load another Table / Data object from the same {group}, you should create another folder {TableName} inside the {group} folder.

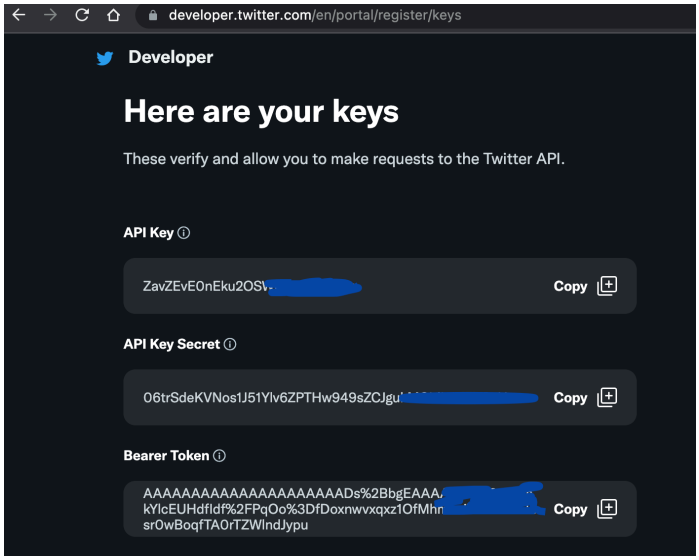
Episode V: Big Data Project - Extract source from Twitter APIs

In the Big Data project, you are free to get data from any source you judge interesting. Here we are getting data from Twitter. We use the API offered by Twitter. This is one example to fetch data in your Data Lake, but feel free to use the method you want, depending on your use case.

	In 2023, Elon Musk decided to shut down the Twitter public API. More info here. So you are not able anymore to do this “Episode V” paragraph. This stays here as an example of how to connect to an API that requires to get access Token.

1. Get Twitter Tokens credentials

Follow this prerequisites paragraph:
https://developer.twitter.com/en/docs/authentication/oauth-2-0/bearer-tokens#Prerequisites
At the end, you’ll get a page like this:



You now have the needed credentials to connect your code to Twitter API.

In your HOME folder of your computer, create the following file: twitter_keys.yaml
And insert the following content:

twitter_keys.yaml

```
search_tweets_v2:
  endpoint:  https://api.twitter.com/2/tweets/search/recent
  consumer_key: <YOUR_CONSUMER_KEY>
  consumer_secret: <YOUR_CONSUMER_SECRET_KEY>
  bearer_token: <YOUR_BEARER_TOKEN>
```

Replace the values: <YOUR_CONSUMER_KEY>, <YOUR_CONSUMER_SECRET_KEY>, <YOUR_CONSUMER_SECRET_KEY> by the ones you got above in your Twitter developer account.

2. Call Twitter API from your Python code

First install the library:

```
pip3 install searchtweets-v2
```

Note: Here is documentation to use the API:

- https://pypi.org/project/searchtweets-v2/
- https://github.com/twitterdev/search-tweets-python/tree/v2

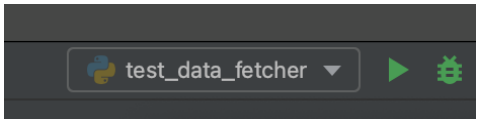
Now, in the file 'data_fetcher.py' previously created, copy this code:

lib/data_fetcher.py/

```
import json
from datetime import date
import os

from searchtweets import gen_request_parameters, load_credentials,
collect_results

HOME = os.path.expanduser('~')
DATALAKE_ROOT_FOLDER = HOME + "/datalake/"


def fetch_data_from_twitter(**kwargs):
    tweets = query_data_from_twitter()
    store_twitter_data(tweets)


def query_data_from_twitter():
    query = gen_request_parameters("#movie", None, results_per_call=100)
    print("We are getting data from Twitter ...", query)
```

```
    search_args = load_credentials("~/.twitter_keys.yaml",
yaml_key="search_tweets_v2", env_overwrite=False)
    return collect_results(query, max_tweets=100,
result_stream_args=search_args)


def store_twitter_data(tweets):
    current_day = date.today().strftime("%Y%m%d")
    TARGET_PATH = DATALAKE_ROOT_FOLDER + "raw/twitter/Movie/" + current_day +
"/"
    if not os.path.exists(TARGET_PATH):
        os.makedirs(TARGET_PATH)
    print("Writing here: ", TARGET_PATH)
    f = open(TARGET_PATH + "twitter.json", "w+")
    f.write(json.dumps(tweets, indent=4))
```

Launch the file test_data_fetcher, in order to test all work well in the code:



In your filesystem, you should see a new file into your Datalake:

```
thibautdebroca@Thibauts-MacBook-Pro:~$ ls -lr ~/datalake/raw/twitter/Movie/20220417/
total 64
-rw-r--r--  1 thibautdebroca  staff  32290 Apr 17 16:27 twitter.json
```

That's it you have loaded data from Twitter.

In Windows, if you have a problem like "cannot find twitter_keys.yaml"

You could use Env variables to pass the credentials
https://pypi.org/project/searchtweets-v2/#environment-variables

Click on your

And add your env variables like this:

You turn: You should adapt the code to handle pagination and fetch more data from the API than the 100 lines in the example.

	In a real Airflow in production, we should avoid treating data inside the machine/executors of Airflow. Indeed Airflow is only a scheduler, and the machines are generally small. If you want to treat massive data, you should do it on an external system (a batch system with huge machines). Airflow will only be responsible for calling this external system. In our Big Data Project, we are processing a huge amount of data, so we can do that from the Airflow machine. Of course, it's faster to do it like this.

3. Tune the query

In the code we have used a simple query '#movie' but you can tune your query with many things:

- Use "OR" "AND" logical conditions
- Use "is:retweet" to get only retweeted tweet
- Use "has:images" to get only tweet with images
- …

The full possibilities to build a query are here:
https://developer.twitter.com/en/docs/twitter-api/tweets/search/integrate/build-a-query

Episode VI: Big Data Project - Extract data from IMDB

Here we'll see another method of extracting data from the Internet.

In the default use case, we want to fetch data from IMDB to get ratings of movies.

There are many ways to fetch data from IMDB on the web. Hopefully, IMDB provide open data files that they refresh every day here: https://datasets.imdbws.com/

1. Extract one table data from IMDB

Similarly to the previous *data_fetcher.py*, create a file called *imdb_fetcher.py* and add this code:

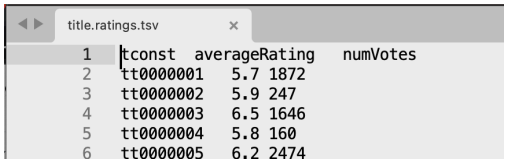
lib/imdb_fetcher.py

```
import os
from datetime import date

import requests

HOME = os.path.expanduser('~')
DATALAKE_ROOT_FOLDER = HOME + "/datalake/"


def fetch_data_from_imdb(**kwargs):
    current_day = date.today().strftime("%Y%m%d")
    TARGET_PATH = DATALAKE_ROOT_FOLDER + "raw/imdb/MovieRating/" + current_day
+ "/"
    if not os.path.exists(TARGET_PATH):
        os.makedirs(TARGET_PATH)

    url = 'https://datasets.imdbws.com/title.ratings.tsv.gz'
    r = requests.get(url, allow_redirects=True)
    open(TARGET_PATH + 'title.ratings.tsv.gz', 'wb').write(r.content)
```

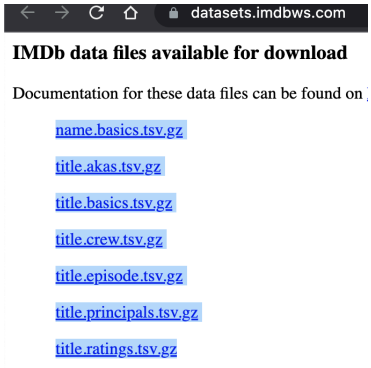
Run the code from another test file to check if it works well.

2. Extract all tables data from IMDB

In the previous code, we have loaded only the table rating. If we look at the file content:



We only have movie ID (tt0000001, tt0000002 …), so we would like to have at least the name of the Movie. When looking at data provided by IMDB (https://datasets.imdbws.com/), we can see :



Adapt the code before to factorize the function, so we can easily pass another URL name and a different DataEntity name.
Your function should look like this:

```
def fetch_data_from_imdb(url, data_entity_name, **kwargs):
    # TODO: Implement
```

- The result of `'https://datasets.imdbws.com/title.ratings.tsv.gz'` will go in `datalake/raw/imdb/MovieRating/$DATE/…`
- The result of `'https://datasets.imdbws.com/name.basics.tsv.gz'` will go in `datalake/raw/imdb/MovieName/$DATE/…`
- …

Episode VII: Big Data Project - Prepare formatted data

We now want to prepare data in a way they can be analyzed easily.

For that, we chose to convert our data in Parquet format. A classic format found in Datalake: https://databricks.com/glossary/what-is-parquet#:~:text=What%20is%20Parquet%3F,handle%20complex%20data%20in%20bulk. This is ‘column oriented’ data file format designed for efficient data storage and retrieval.

In order to convert the data, we’ll use a traditional Python library called pandas.
Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

Let’s install pandas:

```
pip3 install pandas
```

In order to use parquet, let's install pyarrow:

```
pip3 install pyarrow
```

1. Prepare IMDB data in formatted layer

Now create a file called *raw_to_fmt_imdb.py* inside the folder lib:

lib/raw_to_fmt_imdb.py

```
import os

import pandas as pd

HOME = os.path.expanduser('~')
DATALAKE_ROOT_FOLDER = HOME + "/datalake/"


def convert_raw_to_formatted(file_name, current_day):
   RATING_PATH = DATALAKE_ROOT_FOLDER + "raw/imdb/MovieRating/" + current_day
+ "/" + file_name
   FORMATTED_RATING_FOLDER = DATALAKE_ROOT_FOLDER +
"formatted/imdb/MovieRating/" + current_day + "/"
   if not os.path.exists(FORMATTED_RATING_FOLDER):
       os.makedirs(FORMATTED_RATING_FOLDER)
   df = pd.read_csv(RATING_PATH, sep='\t')
   parquet_file_name = file_name.replace(".tsv.gz", ".snappy.parquet")
   df.to_parquet(FORMATTED_RATING_FOLDER + parquet_file_name)
```

Read the code, understand what it does. Run the code, to verify it works well.

Check the size of both files. Explain why one is bigger than the other.
It appears that there is an advantage with the file with a bigger size. What is it ?

2. Prepare Twitter data in formatted layer

For twitter data, the structure is a little bit different than a simple CSV. let's have a look at the structure. Open 'twitter.json' file or read a preview. For that go in a Terminal, in the folder of 'twitter.json' and launch:

head -20 twitter.json

This will look to something like this:

```
head -20 twitter.json
[
  {
    "data": [
      {
        "id": "1515698191521439752",
        "text": "Went to @bletchleypark \nSo cool to go back in time. \nBase on the pic#movie
#theimitationgame\n#bletchleypark #UnitedKingdom #ENGLAND #WW2 https://t.co/TQAUpJ0a9I"
      },
      {
        "id": "1515698078199734279",
        "text": "the movie. \nWith an awesome synth soundtrack and 80s aesthetic this movie is highly
recommended to fans of \"It Follows\" and \"Nightmare on Elm Street\".\n\nThis is one of my favorite movies of the
last decade. \n\n#HorrorMovies #WritingCommunity #horror #movie https://t.co/SPdn9T3Amo"
      },
      {
        "id": "1515698020117061641",
        "text": "RT @ag_poll: Just a general opinion poll #movie question, w/no correct answer, which #film do
you think of when the word #Easter is mention\u2026"
      },
      {
        "id": "1515697971534442504",
        "text": "RT @LuanaMallorquin: I\u2019m making a short film for my final project at university , This is a
sneak peek into the concept\n\n#art #movie #anima\u2026"
      },
      {
```

You see the file is composed of a first array, with a first element containing an attribute ‘data’ which contains the elements we want to read.

Create a file *raw_to_fmt_twitter.py* and add the following code:

lib/raw_to_fmt_twitter.py

```
import os

import pandas as pd

HOME = os.path.expanduser('~')
DATALAKE_ROOT_FOLDER = HOME + "/datalake/"


def convert_raw_to_formatted(file_name, current_day):
    RATING_PATH = DATALAKE_ROOT_FOLDER + "raw/twitter/Movie/" + current_day +
"/" + file_name
    FORMATTED_RATING_FOLDER = DATALAKE_ROOT_FOLDER + "formatted/twitter/Movie/"
```

```
+ current_day + "/"
    if not os.path.exists(FORMATTED_RATING_FOLDER):
        os.makedirs(FORMATTED_RATING_FOLDER)
    df = pd.read_json(RATING_PATH)
    parquet_file_name = file_name.replace(".json", ".snappy.parquet")
    final_df = pd.DataFrame(data=df.data)
    final_df.to_parquet(FORMATTED_RATING_FOLDER + parquet_file_name)
```

Episode VIII: Big Data Project - Prepare combined data

Now that you are a professional of SQL, we'll try to use SQL in order to analyze all our data. (Of course, you are totally free to use any tool/language, but Spark is a very great tool).

(One of) The best tool(s) to compute data in parallel is Apache Spark. Spark provides many way to query data: You can query via DataFrame, or SQL.

Here is full documentation: https://spark.apache.org/docs/latest/

1. Install Apache Spark

Download the latest Apache Spark version:

https://spark.apache.org/downloads.html

Untar/Unzip Apache Spark distribution.

Then move it in /opt/spark-2.3.0 folder

```
mv spark-2.3.0-bin-hadoop2.7 /opt/spark-2.3.0
```

Make a symbolic link to /opt/spark:

```
ln -s /opt/spark-2.3.0 /opt/spark
```

More here:
https://towardsdatascience.com/how-to-use-pyspark-on-your-computer-9c7180075617

2. Use Spark to analyze your data and save the output

Create a file *combine_data.py* with the following content

lib/combine_data.py

```
import os
from pyspark.sql import SQLContext

HOME = os.path.expanduser('~')
DATALAKE_ROOT_FOLDER = HOME + "/datalake/"


def combine_data(current_day):
   RATING_PATH = DATALAKE_ROOT_FOLDER + "formatted/imdb/MovieRating/" +
current_day + "/"
   USAGE_OUTPUT_FOLDER_STATS = DATALAKE_ROOT_FOLDER +
"usage/movieAnalysis/MovieStatistics/" + current_day + "/"
   USAGE_OUTPUT_FOLDER_BEST = DATALAKE_ROOT_FOLDER +
"usage/movieAnalysis/MovieTop10/" + current_day + "/"
   if not os.path.exists(USAGE_OUTPUT_FOLDER_STATS):
       os.makedirs(USAGE_OUTPUT_FOLDER_STATS)
   if not os.path.exists(USAGE_OUTPUT_FOLDER_BEST):
       os.makedirs(USAGE_OUTPUT_FOLDER_BEST)

   from pyspark import SparkContext

   sc = SparkContext(appName="CombineData")
   sqlContext = SQLContext(sc)
   df_ratings = sqlContext.read.parquet(RATING_PATH)
   df_ratings.registerTempTable("ratings")

   # Check content of the DataFrame df_ratings:
   print(df_ratings.show())

   stats_df = sqlContext.sql("SELECT AVG(averageRating) AS avg_rating,"
                             "       MAX(averageRating) AS max_rating,"
                             "       MIN(averageRating) AS min_rating,"
                             "       COUNT(averageRating) AS count_rating"
                             "    FROM ratings LIMIT 10")
   top10_df = sqlContext.sql("SELECT tconst, averageRating"
                             "    FROM ratings"
                             "    WHERE numVotes > 50000 "
                             "    ORDER BY averageRating DESC"
                             "    LIMIT 10")

   # Check content of the DataFrame stats_df and save it:
   print(stats_df.show())
   stats_df.write.save(USAGE_OUTPUT_FOLDER_STATS + "res.snappy.parquet",
mode="overwrite")
```

```
    # Check content of the DataFrame top10_df  and save it:
    print(top10_df.show())
    stats_df.write.save(USAGE_OUTPUT_FOLDER_BEST + "res.snappy.parquet",
mode="overwrite")
```

3. (Bonus) Install a local Jupyter Notebook

I recommend you to install a Jupyter Notebook in your local machine. That will permit you to easily try many data fetches.

Chapter IX: Big Data Project - Ingest into EK

This part is well explained here:
https://docs.google.com/presentation/d/1UgdI0ELB47kmWrqpTbFZeI0pe439qKyKNrp1fBKqXeg/edit#slide=id.g7a356531a1_3_47
You can follow this part.

In order to easily link between Spark and ELK, one possibility is to use .collect() on your dataframe, which read all data into an array. But be careful, collect is a method that get all the data in the driver of the Spark which is generally a bad idea in a real big data environment, as it will load all the data into 1 machine. The good practice (a little bit harder than a collect), is to use the official connector:
https://www.elastic.co/guide/en/elasticsearch/hadoop/current/spark.html.

Episode X: Big Data Project - Wrap Up

You now have all tools to build a clean, strong data pipeline, from fetching data, preparing data, combining data.

You are now free and expert to do any nice pipeline and produce values from different data inputs. Enjoy and I hope to see great projects from you !