Airflow

1. Introduction et installation

Introduction

Airflow est un outil d'automatisation et d'orchestration développé en Python par les équipes de AirBNB dans les années 2010. Aujourd'hui, [Airflow](https://airflow.apache.org/) est développé par la Fondation Apache.

Open Source, Airflow est un outil très populaire avec plus de 22 000 stars sur [Github](https://github.com/apache/airflow/). On peut aussi voir qu'Airflow a été adopté par de nombreuses [entreprises](https://stackshare.io/airflow) comme Slack ou Walmart.

Airflow propose de nombreuses fonctionnalités pour mettre en place des pipelines d'actions, pour les automatiser, pour les surveiller... De plus, Airflow permet de gérer un ensemble d'utilisateurs en leur attribuant différents rôles. Enfin, à l'heure où les entreprises se tournent de plus en plus vers le Cloud, Airflow permet une intégration simple de ces technologies.

Installation

Dans ce cours, nous allons utiliser la version Docker de Airflow pour simplifier l'installation.

 Si vous préférez utiliser une version classique de Airflow, vous pouvez utiliser les instructions données dans le cours d'**Introduction à Airflow**. Il y aura très peu de différences entre l'utilisation de Docker et l'utilisation d'une installation classique. L'utilisation de Docker permet de simplifier l'installation.

Les instructions d'installation sont reprises de la [documentation officielle d'Airflow](https://airflow.apache.org/docs/apache-airflow/stable/start/docker.html).

Nous allons devoir télécharger un fichier docker-compose pour lancer les différents containers correspondant aux différents composants de Airflow.

Exécutez la commande suivante pour télécharger le fichier docker-compose.yml

wget <https://dst-de.s3.eu-west-3.amazonaws.com/airflow_avance_fr/docker-compose/docker-compose.yaml>

Airflow nécessite la création de volumes pour simplifier l'échanges de fichier depuis la machine hôte vers les containers. On va donc créer 3 dossiers: dags, logs, plugins.

Exécutez la commande suivante pour créer ces dossiers

mkdir ./dags ./logs ./plugins

Enfin, on doit créer un fichier de variables pour Docker-Compose.

Exécutez la commande suivante

echo -e "AIRFLOW\_UID=$(id -u)\nAIRFLOW\_GID=0" > .env

Si vous avez fait le cours *Kubernetes*, il est important d'arrêtez les conteneurs liés à ce cours, sinon la machine virtuelle est lente.

Vous pouvez vérifier le contenu du fichier .env avec la commande cat .env. Maintenant que la configuration est prête, on peut initialiser Airflow.

Exécutez la commande suivante

docker-compose up airflow-init

Cette opération peut prendre un peu de temps. Lorsque le message start\_airflow-init\_1 exited with code 0 s'affiche, l'initialisation est terminée et on peut lancer les containers.

Exécutez la commande suivante pour lancer les containers

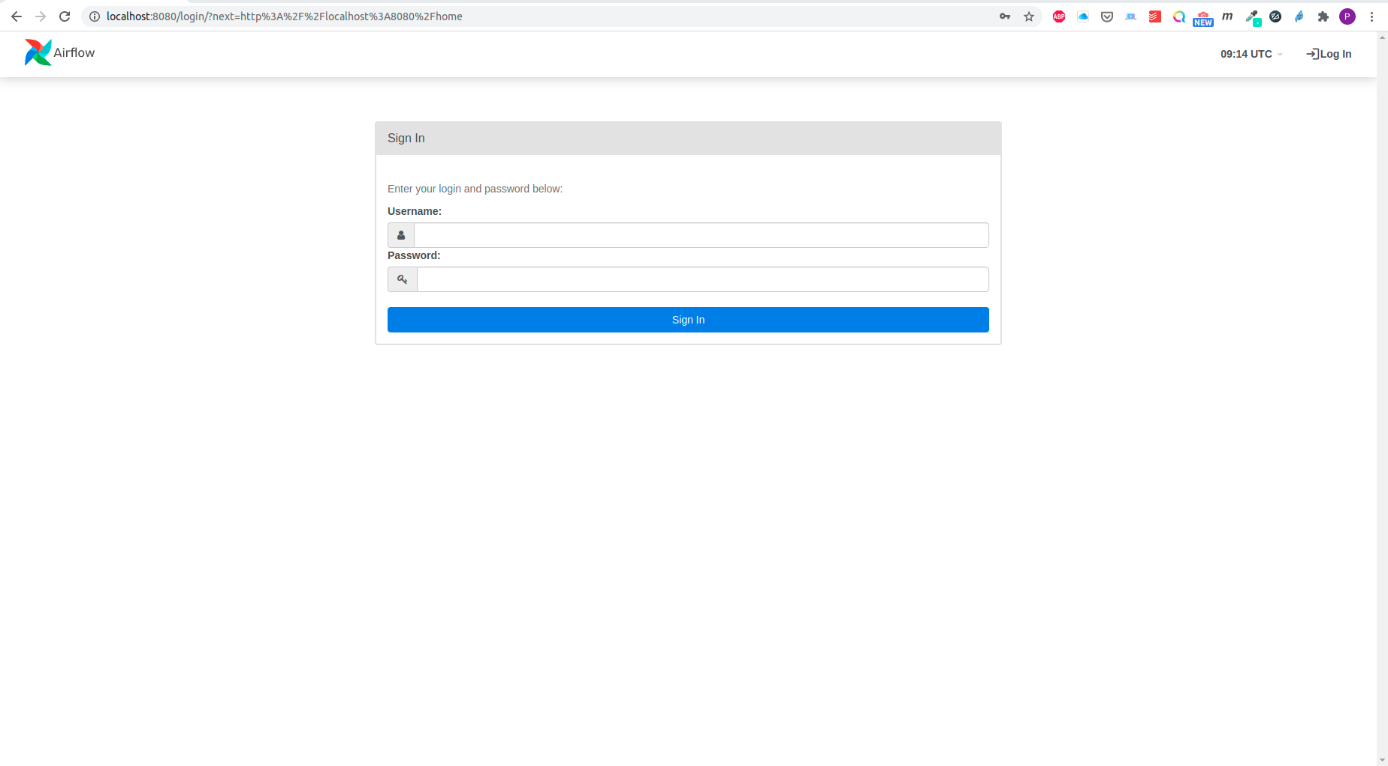
docker-compose up -d

Après quelques instants, on peut vérifier que 6 containers fonctionnent bien en utilisant la commande docker container ls.

Pour vérifier qu'Airflow fonctionne correctement, on peut essayer d'afficher l'interface graphique: elle est disponible sur le port 8080 de la machine virtuelle. On peut donc essayer d'y accéder en ouvrant dans un navigateur l'adresse IP de la machine virtuelle et le port 8080. Pour rappel, on peut voir l'adresse IP de la machine virtuelle l'encart Connect, 53.132.17.34 par exemple.

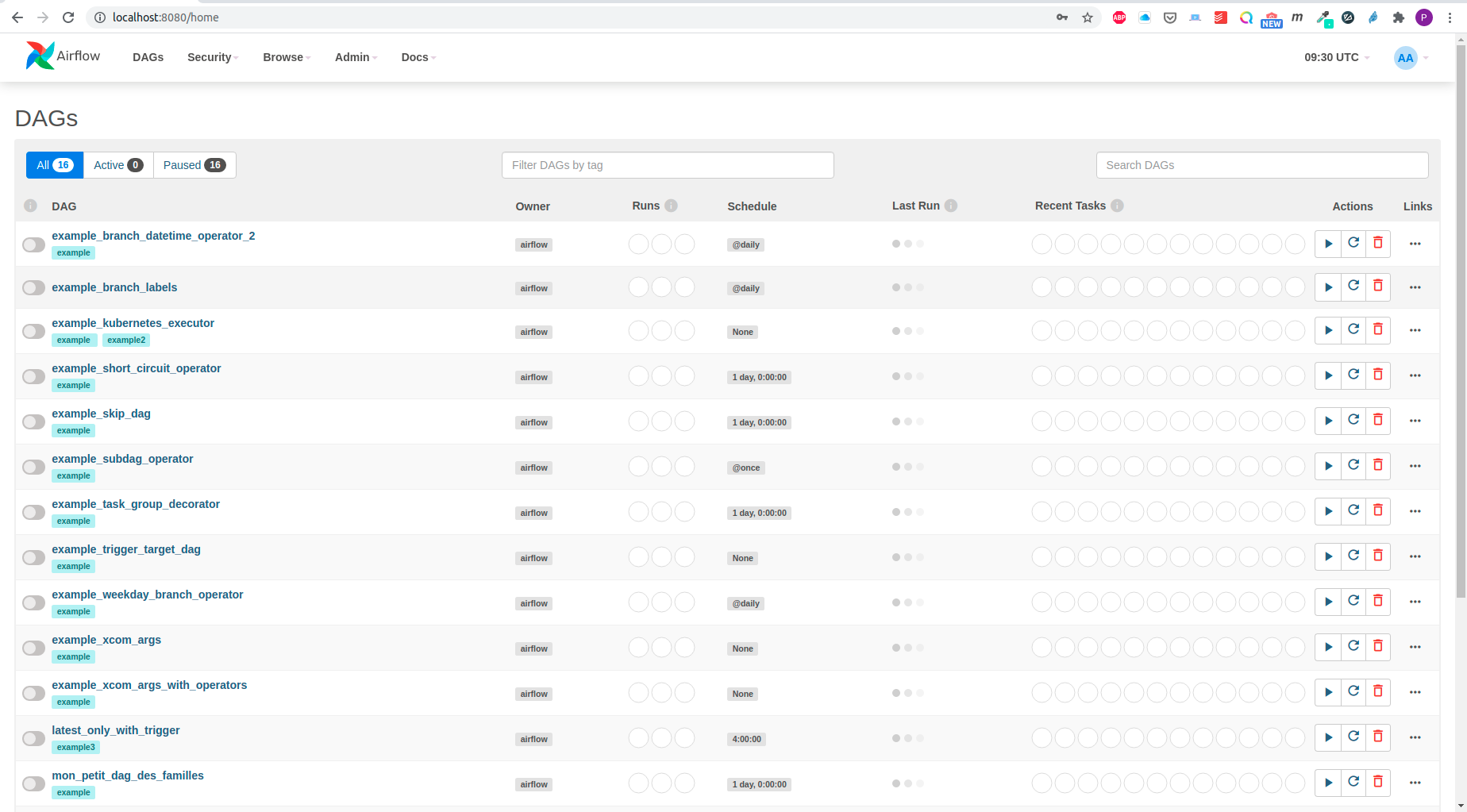
Ouvrez dans votre navigateur l'adresse 53.132.17.34:8080 en remplaçant 53.132.17.34 par l'adresse de votre machine

Vous devriez voir l'interface suivante:



Vous pouvez vous connecter en utilisant le nom d'utilisateur airflow et le mot de passe airflow.

L'interface devrait alors ressembler à ceci:



Nous reviendrons en détails sur l'utilisation de cette interface dans la prochaine partie du cours. Airflow met aussi à notre disposition un script pour utiliser l'interface en ligne de commande de Airflow mais aussi se connecter aux containers via un shell bash ou un shell Python.

Téléchargez le script suivant

wget https://dst-de.s3.eu-west-3.amazonaws.com/airflow\_avance\_fr/docker-compose/airflow.sh

chmod +x airflow.sh

On peut utiliser ce script de 3 manière différentes:

* ./airflow.sh ... suivi des commandes classiques de l'interface en ligne de commande de Airflow
* ./airflow.sh python pour ouvrir un shell Python
* ./airflow.sh bash pour ouvrir un shell bash

Exécutez la commande suivante pour afficher les commandes principales de l'interface en ligne de commande de Airflow

./airflow.sh –help

Notez qu'avec une utilisation classique de Airflow, l'interface en ligne de commande s'utilise simplement avec la commande airflow. On pourra donc remplacer cette commande par airflow --help.

Exécutez la commande suivante pour ouvrir un shell bash

./airflow.sh bash

On peut se déplacer à l'intérieur du container et voir notamment les volumes créés plus tôt. On peut aussi exécuter des commandes via l'interface en ligne de commande de Airflow (./airflow.sh --help au lieu de airflow --help par exemple). On pourra quitter cette interface avec la commande exit.

### Conclusion

Dans cette partie, nous avons introduit Airflow et installé cet outil via Docker. Dans la prochaine partie, nous allons voir les principes de bases de Airflow ainsi que notre premier workflow.

Annexe :

Docker compose yaml contenu :

# Licensed to the Apache Software Foundation (ASF) under one

# or more contributor license agreements. See the NOTICE file

# distributed with this work for additional information

# regarding copyright ownership. The ASF licenses this file

# to you under the Apache License, Version 2.0 (the

# "License"); you may not use this file except in compliance

# with the License. You may obtain a copy of the License at

#

# http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing,

# software distributed under the License is distributed on an

# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY

# KIND, either express or implied. See the License for the

# specific language governing permissions and limitations

# under the License.

#

# Basic Airflow cluster configuration for CeleryExecutor with Redis and PostgreSQL.

#

# WARNING: This configuration is for local development. Do not use it in a production deployment.

#

# This configuration supports basic configuration using environment variables or an .env file

# The following variables are supported:

#

# AIRFLOW\_IMAGE\_NAME - Docker image name used to run Airflow.

# Default: apache/airflow:master-python3.8

# AIRFLOW\_UID - User ID in Airflow containers

# Default: 50000

# AIRFLOW\_GID - Group ID in Airflow containers

# Default: 50000

#

# Those configurations are useful mostly in case of standalone testing/running Airflow in test/try-out mode

#

# \_AIRFLOW\_WWW\_USER\_USERNAME - Username for the administrator account (if requested).

# Default: airflow

# \_AIRFLOW\_WWW\_USER\_PASSWORD - Password for the administrator account (if requested).

# Default: airflow

# \_PIP\_ADDITIONAL\_REQUIREMENTS - Additional PIP requirements to add when starting all containers.

# Default: ''

#

# Feel free to modify this file to suit your needs.

---

version: "3"

x-airflow-common: &airflow-common

image: ${AIRFLOW\_IMAGE\_NAME:-apache/airflow:2.1.1}

environment: &airflow-common-env

AIRFLOW\_\_CORE\_\_EXECUTOR: CeleryExecutor

AIRFLOW\_\_CORE\_\_SQL\_ALCHEMY\_CONN: postgresql+psycopg2://airflow:airflow@postgres/airflow

AIRFLOW\_\_CELERY\_\_RESULT\_BACKEND: db+postgresql://airflow:airflow@postgres/airflow

AIRFLOW\_\_CELERY\_\_BROKER\_URL: redis://:@redis:6379/0

AIRFLOW\_\_CORE\_\_FERNET\_KEY: ""

AIRFLOW\_\_CORE\_\_DAGS\_ARE\_PAUSED\_AT\_CREATION: "true"

AIRFLOW\_\_CORE\_\_LOAD\_EXAMPLES: "true"

AIRFLOW\_\_API\_\_AUTH\_BACKEND: "airflow.api.auth.backend.basic\_auth"

AIRFLOW\_\_SMTP\_\_SMTP\_HOST: "smtp.gmail.com"

AIRFLOW\_\_SMTP\_\_SMTP\_PORT: 587

AIRFLOW\_\_SMTP\_\_SMTP\_USER: "de.airflow@gmail.com"

AIRFLOW\_\_SMTP\_\_SMTP\_PASSWORD: "cfsrvkongsobheta"

AIRFLOW\_\_SMTP\_\_SMTP\_MAIL\_FROM: "de.airflow@gmail.com"

\_PIP\_ADDITIONAL\_REQUIREMENTS: ${\_PIP\_ADDITIONAL\_REQUIREMENTS:- pymongo}

volumes:

- ./dags:/opt/airflow/dags

- ./logs:/opt/airflow/logs

- ./plugins:/opt/airflow/plugins

user: "${AIRFLOW\_UID:-50000}:${AIRFLOW\_GID:-50000}"

depends\_on:

redis:

condition: service\_healthy

postgres:

condition: service\_healthy

services:

postgres:

image: postgres:13

environment:

POSTGRES\_USER: airflow

POSTGRES\_PASSWORD: airflow

POSTGRES\_DB: airflow

volumes:

- postgres-db-volume:/var/lib/postgresql/data

healthcheck:

test: ["CMD", "pg\_isready", "-U", "airflow"]

interval: 5s

retries: 5

restart: always

redis:

image: redis:latest

ports:

- 6379:6379

healthcheck:

test: ["CMD", "redis-cli", "ping"]

interval: 5s

timeout: 30s

retries: 50

restart: always

airflow-webserver:

<<: \*airflow-common

command: webserver

ports:

- 8080:8080

healthcheck:

test: ["CMD", "curl", "--fail", "http://localhost:8080/health"]

interval: 10s

timeout: 10s

retries: 5

restart: always

airflow-scheduler:

<<: \*airflow-common

command: scheduler

healthcheck:

test:

[

"CMD-SHELL",

'airflow jobs check --job-type SchedulerJob --hostname "$${HOSTNAME}"',

]

interval: 10s

timeout: 10s

retries: 5

restart: always

airflow-worker:

<<: \*airflow-common

command: celery worker

healthcheck:

test:

- "CMD-SHELL"

- 'celery --app airflow.executors.celery\_executor.app inspect ping -d "celery@$${HOSTNAME}"'

interval: 10s

timeout: 10s

retries: 5

restart: always

airflow-init:

<<: \*airflow-common

command: version

environment:

<<: \*airflow-common-env

\_AIRFLOW\_DB\_UPGRADE: "true"

\_AIRFLOW\_WWW\_USER\_CREATE: "true"

\_AIRFLOW\_WWW\_USER\_USERNAME: ${\_AIRFLOW\_WWW\_USER\_USERNAME:-airflow}

\_AIRFLOW\_WWW\_USER\_PASSWORD: ${\_AIRFLOW\_WWW\_USER\_PASSWORD:-airflow}

flower:

<<: \*airflow-common

command: celery flower

ports:

- 5555:5555

healthcheck:

test: ["CMD", "curl", "--fail", "http://localhost:5555/"]

interval: 10s

timeout: 10s

retries: 5

restart: always

volumes:

postgres-db-volume:

Dockerfile:

FROM jenkins/jenkins:2.346.3-jdk11

USER root

RUN apt-get update && apt-get install -y lsb-release

RUN curl -fsSLo /usr/share/keyrings/docker-archive-keyring.asc \

https://download.docker.com/linux/debian/gpg

RUN echo "deb [arch=$(dpkg --print-architecture) \

signed-by=/usr/share/keyrings/docker-archive-keyring.asc] \

https://download.docker.com/linux/debian \

$(lsb\_release -cs) stable" > /etc/apt/sources.list.d/docker.list

RUN apt-get update && apt-get install -y docker-ce-cli

USER jenkins

RUN jenkins-plugin-cli --plugins "blueocean:1.25.6 docker-workflow:1.29"

Nom du fichier.py

import dash

import dash\_core\_components as dcc

import dash\_html\_components as html

import pandas as pd

import plotly.express as px

from dash.dependencies import Output,Input

df = px.data.gapminder()

df\_1 = df[df['year'] == 2002]

external\_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(\_\_name\_\_, external\_stylesheets=external\_stylesheets,suppress\_callback\_exceptions=True)

app.layout = html.Div([

dcc.Location(id='url', refresh=False),

html.Div(id = 'page-content')

])

index\_page = html.Div([

html.H1('première Application Multipages', style={'color' : 'aquamarine', 'textAlign': 'center'}),

html.Button(dcc.Link('Espérance de vie par PIB', href='/page-1')),

html.Br(),

html.Button(dcc.Link('carte du monde', href='/page-2'))

], style={'alignItems': 'center'})

# Page 1

layout\_1 = html.Div([

html.H1('API Dash', style={'textAlign': 'center', 'color': 'mediumturquoise'}),

html.Div(dcc.Graph(id='page-1-graph')),

html.Div(dcc.Slider(id = 'page-1-slider',

min = df['year'].min(),

max = df['year'].max(),

marks={str(year): str(year) for year in df['year'].unique()},

step = None)),

html.Button(dcc.Link('Revenir à la page de garde', href='/'))

], style = {'background' : 'beige'})

@app.callback(Output(component\_id='page-1-graph', component\_property='figure'),

[Input(component\_id='page-1-slider', component\_property='value')])

def update\_graph(filter\_year):

df\_2 = df[df["year"] == filter\_year]

# Création de la figure plotly

fig = px.scatter(df\_2, x="gdpPercap",

y = "lifeExp",

color="continent",

size="pop")

return fig

# Page 2

layout\_2 = html.Div([

html.H1('Page 2', style={'textAlign': 'center', 'color': 'mediumturquoise'}),

html.Div(dcc.Dropdown(id = 'page-2-dropdown',

options= [{'label': 'life expandency', 'value': 'lifeExp'},

{'label': 'population', 'value': 'pop'}],

value= 'lifeExp'

)),

html.Div(dcc.Graph(id='page-2-graph')),

html.Button(dcc.Link('Revenir à la page de garde', href='/'))

], style = {'background' : 'beige'})

@app.callback(Output(component\_id='page-2-graph', component\_property='figure'),

[Input(component\_id='page-2-dropdown', component\_property='value')])

def update\_graph\_1(indicator):

# Création de la figure plotly

fig = px.scatter\_geo(df\_1, locations="iso\_alpha", color=indicator,

hover\_name="country", size="pop",

projection="natural earth")

return fig

# Mise à jour de l'index

@app.callback(dash.dependencies.Output('page-content', 'children'),

[dash.dependencies.Input('url', 'pathname')])

def display\_page(pathname):

if pathname == '/page-1':

return layout\_1

elif pathname == '/page-2':

return layout\_2

else:

return index\_page

if \_\_name\_\_ == '\_\_main\_\_':

app.run\_server(debug=True,host="0.0.0.0")

## 2. Premiers DAGs

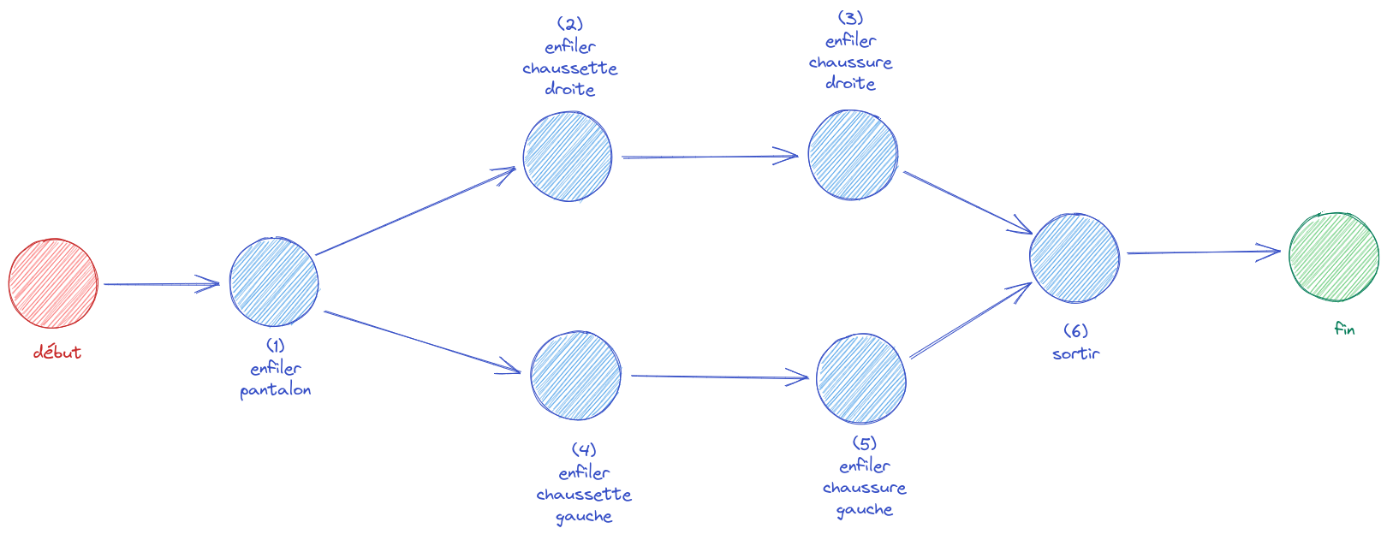
Dans cette partie, nous allons introduire les notions de DAG, de Task, de Job, d'Operator et de Run. Nous allons pouvoir créer notre premier DAG et l'enregistrer dans Airflow puis l'exécuter. Enfin nous verrons comment fonctionne l'interface

### Concepts

Airflow est un outil qui permet d'organiser un ensemble de tâches en leur donnant des dépendances, c'est-à-dire des conditions sur l'exécution de ces tâches. Pour définir ces dépendances, on utilise un DAG, diminutif de Directed Acyclic Graph, ou, en français, Graphe Dirigé Acyclique.

  Les graphes sont des outils mathématiques issus de la théorie des graphes. On y retrouve un ensemble de points appelés des nœuds représentant différentes entités et des relations, appelées arrêtes, entre ces nœuds. Des exemples très communs de graphe sont les plans de [métro](https://www.plandeparis.info/plans-de-metro/plan-de-metro-bonne-definition.gif): les nœuds sont les stations tandis que les liaisons entre les stations sont représentées par les arêtes.

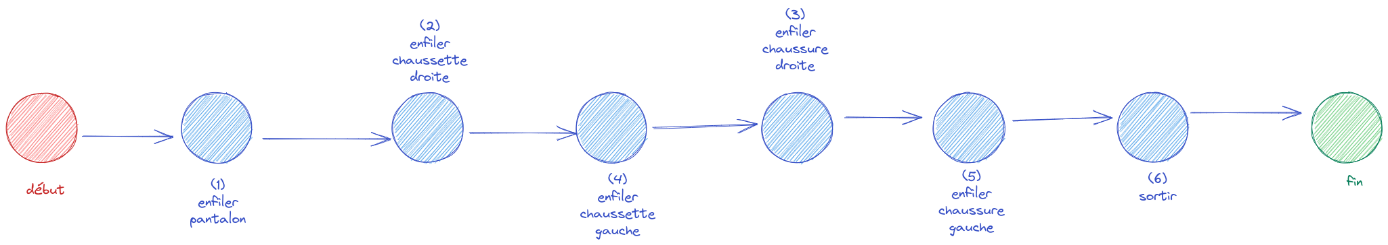
Un DAG est un graphe dirigé, c'est-à-dire que les arêtes ne vont que dans un sens, et acyclique, c'est-à-dire qu'on ne peut pas passer deux fois au même endroit lorsqu'on se déplace sur le graphe en suivant le sens des arêtes.



Dans l'exemple ci-dessous, on comprend bien la dépendance des différentes tâches: l'accomplissement de la dernière tâche (6) repose sur la réalisation des tâches (3) et (5). La tâche (3) dépend de la tâche (2) alors que la tâche (5) dépend de la tâche (4). Enfin, les tâches (2) et (4) ne peuvent se faire sans que la tâche (1) ne soit réalisée.

On peut donc voir que dans ce DAG, il faut que la tâche (1) doit être finalisée avant que les tâches (2) et (4) ne s'exécutent. Ces deux tâches ne dépendent pas l'une de l'autre mais sont nécessaires pour les tâches (3) et (5) respectivement. Une fois que ces tâches sont exécutées on pourra exécuter la dernière tâche.

Notez que le DAG pourrait tout à fait être utilisé pour la même opération.



Les tâches sont les mêmes mais leurs dépendances ne sont plus les mêmes. Ainsi, si la tâche (2) échoue, tous les tâches suivantes n'auront pas lieu. Dans le premier cas, si la tache (2) échoue, elle n'empêche pas la réalisation des autres tâches.

On voit donc l'importance de la définition du DAG et de sa topologie.

Dans Airflow, un DAG est donc constitué d'un ensemble de Task arrangées d'une certaine manière. Une Task représente une unité de travail théorique: on peut oser la comparaison avec une fonction Python, qui tant qu'elle n'est pas appelée ne représente que un ensemble d'actions potentielles.

Une Task est définie grâce à un Operator. Les Operator sont des objets qui permettent d'utiliser des outils différents. Airflow propose un ensemble d'Operator prédéfini parmi lesquels on retrouve les BashOperator ou les PythonOperator qui permettent l'exécution d'une tâche Bash ou Python respectivement. On trouve beaucoup d'autres Operator qui permettent d'utiliser d'autres outils comme des bases de données (MySqlOperator, PostgresOperator, ...), des outils comme Docker (DockerOperator), ou des outils de communication (EmailOperator, SlackOperator, ...), ...

La définition d'une Task implique donc l'utilisation d'un Operator.

Pour l'instant, nous n'avons abordé que la définition du travail à effectuer. Lorsque l'on définit un DAG, on peut définir la fréquence à laquelle celui-ci s'exécute. On peut choisir d'avoir un DAG qui est exécuté de manière récurrente ou simplement une fois.

Lorsqu'on déclenche (en anglais trigger) un DAG, il s'exécute une première fois. On parle alors d'un DAG run. Si on a défini une fréquence de répétition pour le DAG, alors il se ré-exécutera à un intervalle de temps donné. Lorsqu'un DAG est lancé, chacune des Task de ce DAG est lancée en suivant l'ordre défini par le DAG. Pour chaque tâche lancée, on crée une instance de cette tâche, en anglais une Task instance.

### Définition d'un DAG

Pour définir un DAG, on utilise un script Python.

Ouvrir un fichier de script appelé my\_first\_dag.py

Pour définir le DAG, nous allons utiliser les lignes suivantes:

from airflow import DAG

from airflow.utils.dates import days\_ago

my\_dag = DAG(

dag\_id='my\_very\_first\_dag',

description='My first DAG created with DataScientest',

tags=['tutorial', 'datascientest'],

schedule\_interval=None,

default\_args={

'owner': 'airflow',

'start\_date': days\_ago(2),

}

)

On voit que l'on doit donner un dag\_id à notre DAG. On décide ici d'ajouter aussi une description et des tags, c'est-à-dire des étiquettes qui permettent d'organiser les DAGS. On doit préciser le propriétaire de ce DAG ainsi qu'une start\_date qu'on définit comme étant deux jours avant aujourd'hui. Enfin, on peut préciser un schedule\_interval, c'est-à-dire un intervalle de temps entre deux exécutions.

Collez ces lignes dans le fichier my\_first\_dag.py

### Définition d'une tâche

Nous allons définir une tâche unique très simple qui affiche l'heure, accompagnée d'un message dans la sortie standard. Nous utiliserons donc un PythonOperator. Les lignes suivantes permettent de définir une telle Task.

from airflow.operators.python import PythonOperator

import datetime

# definition of the function to execute

def print\_date\_and\_hello():

print(datetime.datetime.now())

print('Hello from Airflow')

my\_task = PythonOperator(

task\_id='my\_very\_first\_task',

python\_callable=print\_date\_and\_hello,

dag=my\_dag

)

Les exigences minimales pour définir une Task avec le PythonOperator sont de lui fournir un task\_id qui servira d'identifiant à la Task et un python\_callable, c'est-à-dire une fonction Python qui sera exécutée. On passe aussi le dag auquel cette tâche appartient.

Collez ces lignes à la suite du fichier my\_first\_dag.py

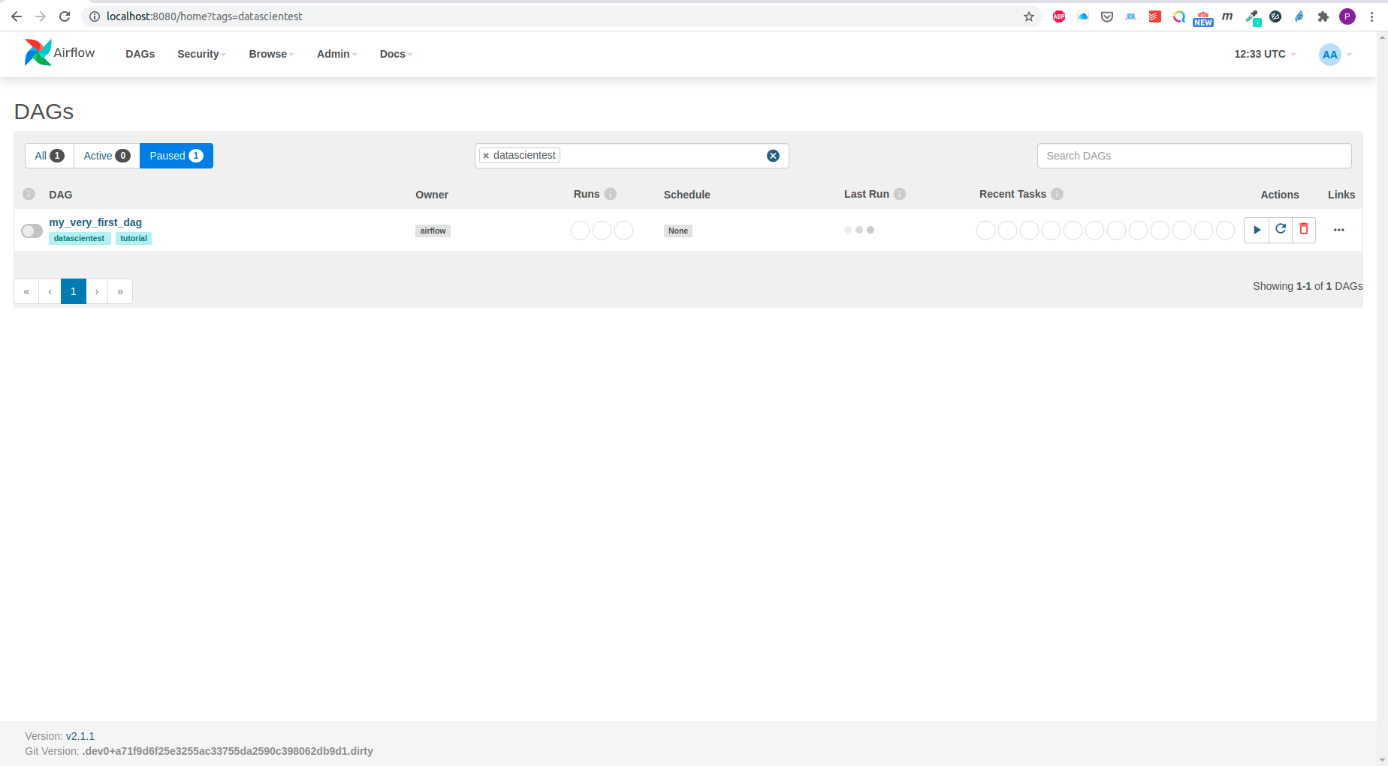
Enregistrement du DAG

Si on veut qu'Airflow prenne en compte ce DAG, il nous faut l'enregistrer dans Airflow, c'est-à-dire coller le fichier dans le dossiers dags et attendre que Airflow ait pris en compte ce nouveau DAG.

Copiez/Collez le fichier my\_first\_dag.py dans le dossier dags

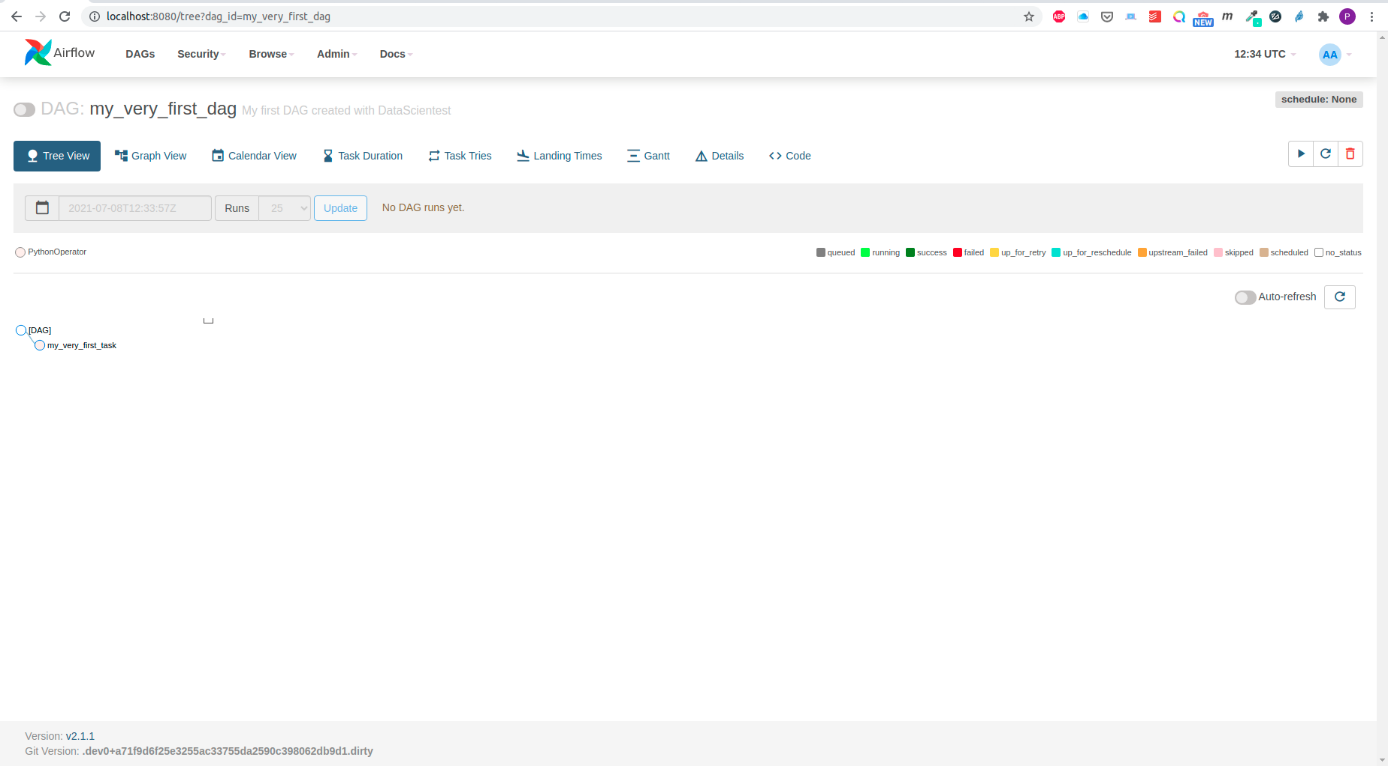
On peut se rendre à présent dans l'interface graphique de Airflow. Pour rappel, vous pouvez y accéder en utilisant l'adresse IP de la machine virtuelle (disponible dans l'encart Connect) et en y ajoutant :8080: par exemple: 53.132.17.34:8080.

Dans l'ongles DAGs, on peut rechercher notre DAG en utilisant le champ Filter DAGs by tag et en rentrant datascientest.



Selectionnez le DAG my\_very\_fist\_dag.py

On arrive sur cette interface:



Explorez les différents onglets Tree View, Graph View, Details et Code

Tree View et Graph View permettent d'afficher le DAG tandis que Details donne les informations générales sur le DAG et Code renvoie le code utilisé pour la définition du DAG. Les autres onglets sont vides tant qu'on n'a pas exécuté le DAG.

1. Enlevez la pause au dessus du bouton Tree View
2. Déclenchez l'exécution du DAG en utilisant le bouton

On peut voir que le DAG s'exécute une première fois. Comme nous n'avons pas spécifié la fréquence à laquelle celui-ci doit se répéter, on ne voit pas de deuxième occurence.

Ré-exécutez une deuxième fois le DAG avec le bouton

Dans la Calendar View, on peut à présent voir que deux Run ont eu lieu avec le statut Success. Dans l'onglet Task Duration, on peut voir la durée de notre tâche lors des deux Run.

Ré-exécutez encore deux fois le DAG

On peut voir dans l'onglet Task Tries le nombre de tentatives au fil du temps.

1. Remettez le DAG sur pause
2. Lancez une nouvelle exécution du DAG
3. Au bout de quelques secondes, enlevez la pause

On remarque qu'avant de relancer le DAG, le Run apparait en vert clair dans la Tree View. Dans l'onglet Landing Times on peut voir que le Landing Time a évolué au fil du temps: on représente ici l'écart entre le moment auquel le DAG a été lancé par rapport au moment où il aurait du être lancé. Ici, il ne pouvait pas être lancé car le DAG était en pause.

Rendez vous dans l'onglet Browse puis DAG Runs

On peut voir l'état des différentes exécutions de nos différents DAGS.

Rendez vous dans l'onglet Browse puis Jobs

Nous reviendrons sur la notion de Job plus tard mais nous pouvons déja voir qu'il y a un Job par exécution de DAG ainsi qu'un SchedulerJob.

Rendez vous dans l'onglet Browse puis Task Instances

On peut voir les différentes exécutions des tâches. Dans la dernière colonne, on peut voir un lien vers les logs correspondant aux instances de Task

Affichez les logs d'une Task Instance

\*\*\* Reading local file: /opt/airflow/logs/my\_very\_first\_dag/my\_very\_first\_task/2021-07-08T12:41:20.372453+00:00/1.log

[2021-07-08 12:43:20,352] {taskinstance.py:896} INFO - Dependencies all met for <TaskInstance: my\_very\_first\_dag.my\_very\_first\_task 2021-07-08T12:41:20.372453+00:00 [queued]>

[2021-07-08 12:43:20,372] {taskinstance.py:896} INFO - Dependencies all met for <TaskInstance: my\_very\_first\_dag.my\_very\_first\_task 2021-07-08T12:41:20.372453+00:00 [queued]>

[2021-07-08 12:43:20,372] {taskinstance.py:1087} INFO -

--------------------------------------------------------------------------------

[2021-07-08 12:43:20,372] {taskinstance.py:1088} INFO - Starting attempt 1 of 1

[2021-07-08 12:43:20,372] {taskinstance.py:1089} INFO -

--------------------------------------------------------------------------------

[2021-07-08 12:43:20,384] {taskinstance.py:1107} INFO - Executing <Task(PythonOperator): my\_very\_first\_task> on 2021-07-08T12:41:20.372453+00:00

[2021-07-08 12:43:20,392] {standard\_task\_runner.py:76} INFO - Running: ['\*\*\*', 'tasks', 'run', 'my\_very\_first\_dag', 'my\_very\_first\_task', '2021-07-08T12:41:20.372453+00:00', '--job-id', '2', '--pool', 'default\_pool', '--raw', '--subdir', 'DAGS\_FOLDER/my\_first\_dag.py', '--cfg-path', '/tmp/tmpw991522o', '--error-file', '/tmp/tmpbdgybcgb']

[2021-07-08 12:43:20,392] {standard\_task\_runner.py:77} INFO - Job 2: Subtask my\_very\_first\_task

[2021-07-08 12:43:20,388] {standard\_task\_runner.py:52} INFO - Started process 14044 to run task

[2021-07-08 12:43:20,441] {logging\_mixin.py:104} INFO - Running <TaskInstance: my\_very\_first\_dag.my\_very\_first\_task 2021-07-08T12:41:20.372453+00:00 [running]> on host f29a05494c5b

[2021-07-08 12:43:20,496] {taskinstance.py:1302} INFO - Exporting the following env vars:

AIRFLOW\_CTX\_DAG\_OWNER=\*\*\*

AIRFLOW\_CTX\_DAG\_ID=my\_very\_first\_dag

AIRFLOW\_CTX\_TASK\_ID=my\_very\_first\_task

AIRFLOW\_CTX\_EXECUTION\_DATE=2021-07-08T12:41:20.372453+00:00

AIRFLOW\_CTX\_DAG\_RUN\_ID=manual\_\_2021-07-08T12:41:20.372453+00:00

[2021-07-08 12:43:20,497] {logging\_mixin.py:104} INFO - 2021-07-08 12:43:20.497129

[2021-07-08 12:43:20,497] {logging\_mixin.py:104} INFO - Hello from Airflow

[2021-07-08 12:43:20,497] {python.py:151} INFO - Done. Returned value was: None

[2021-07-08 12:43:20,509] {taskinstance.py:1211} INFO - Marking task as SUCCESS. dag\_id=my\_very\_first\_dag, task\_id=my\_very\_first\_task, execution\_date=20210708T124120, start\_date=20210708T124320, end\_date=20210708T124320

[2021-07-08 12:43:20,535] {taskinstance.py:1265} INFO - 0 downstream tasks scheduled from follow-on schedule check

[2021-07-08 12:43:20,566] {local\_task\_job.py:151} INFO - Task exited with return code 0

On retrouve les données affichées lors de l'exécution de la fonction définie dans le PythonOperator. On peut aussi afficher les détails de cette instance en cliquant sur le Task Id de la tâche choisie.

Nous parlerons des autres menus dans la suite.

DAG avec deux tâches

Dans cette partie nous allons créer un nouveau DAG avec une deuxième tâche.

1. Créez un nouveau DAG appelé my\_second\_dag
2. Créez une tâche second\_dag\_task1 qui imprime le message hello from task 1
3. Créez une tâche second\_dag\_task2 qui imprime le message hello from task 2
4. Enregistrez ce DAG dans Airflow

Ce second DAG ne sera pas relancé (schedule\_interval=None)

from airflow import DAG

from airflow.utils.dates import days\_ago

from airflow.operators.python import PythonOperator

my\_dag = DAG(

dag\_id='my\_second\_dag',

description='A new DAG with two tasks',

tags=['tutorial', 'datascientest'],

schedule\_interval=None,

default\_args={

'owner': 'airflow',

'start\_date': days\_ago(2),

}

)

def print\_task1():

print('hello from task 1')

def print\_task2():

print('hello from task 2')

task1 = PythonOperator(

task\_id='second\_dag\_task1',

python\_callable=print\_task1,

dag=my\_dag

)

task2 = PythonOperator(

task\_id='second\_dag\_task2',

python\_callable=print\_task2,

dag=my\_dag

)

Une fois que ce DAG est ajouté, on peut se rendre dans l'interface graphique et regarder la Graph View. On voit que les tâches sont indépendantes.

Enlevez la pause et déclenchez l'exécution du DAG plusieurs fois

Si on se rend dans les différents onglets, on peut voir qu'à présent, on observe deux courbes au lieu d'une: on a en fait une courbe par Task.

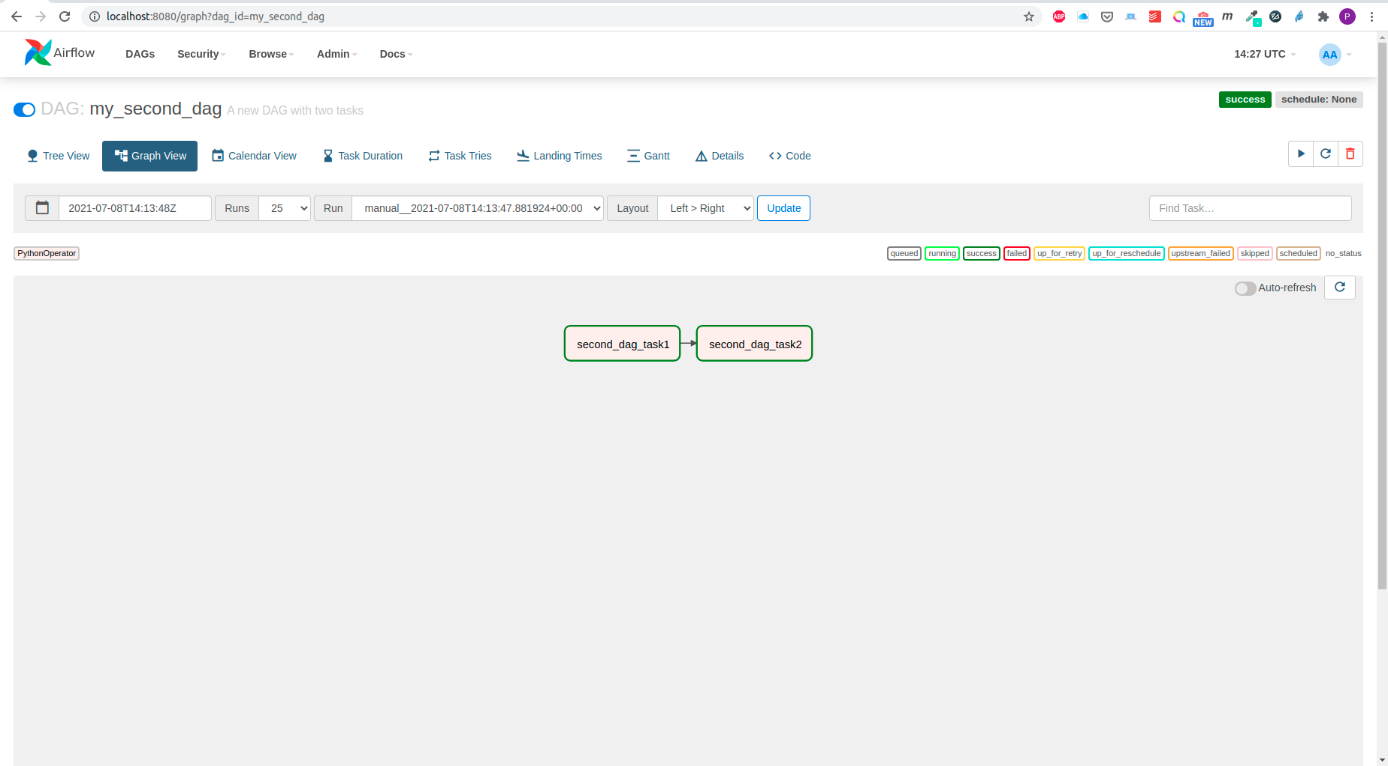
Création de dépendances

Pour créer des dépendances, on peut utiliser l'opérateur >>.

Dans le DAG précédent, ajoutez la ligne task1 >> task2 (en changeant éventuellement les noms)

Cette ligne signifie que la Task task2 dépend de la réalisation de task1.

Mettez à jour ce DAG dans Airflow et affichez la Graph View



On peut voir que l'opérateur >> représente bien la fléche de la vue graphe.

Déclenchez l'exécution du DAG plusieurs fois

Dans l'onglet Gant, on voit bien que l'exécution des tâches se fait de manière successive.

Dans le deuxième DAG, modifiez la première tâche de manière à ce qu'elle dure 30 secondes au moins

# replace only the print\_task1 function

import time

def print\_task1():

time.sleep(30)

print('hello from task 1')

Mettez à jour le DAG dans Airflow, relancez-le et rendez vous sur l'onglet Graph View

On peut voir que la première tâche prend bien un certain temps avant de s'exécuter et que la deuxième est donc bien mise en attente.

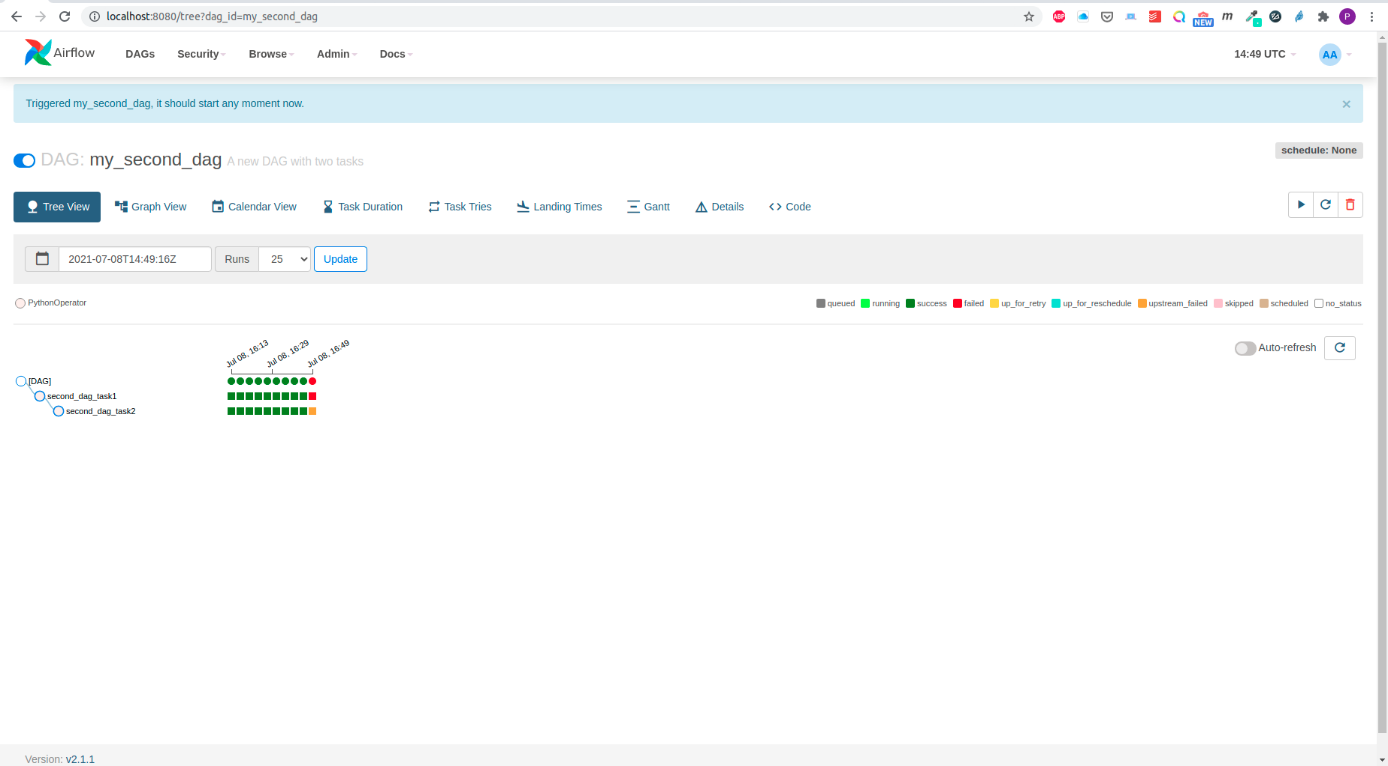
Dans le deuxième DAG, modifiez la première tâche de manière à ce qu'elle provoque une exception de type TypeError

def print\_task1():

raise TypeError('This will not work')

print('hello from task 1')

Exécutez ce DAG



On remarque que cette fois-ci le DAG n'a pas fonctionné. Son statut est failed. La première Task n'a pas fonctionné (failed) et la deuxième n'a même pas pu être lancée à cause de sa dépendance sur la première (upstream\_failed).

Exercice d'application

Dans cette partie, nous allons mettre en place les principes vus précédemment. Pour commencer, nous allons créer une fonction Python:

def print\_text(text):

print(text)

Cette fonction prend en entrée une chaîne de caractères et l'imprime dans la sortie standard. Pour passer des arguments à une fonction avec le PythonOperator, on peut préciser l'argument op\_kwargs lors de la définition de la tâche. Par exemple:

task1 = PythonOperator(

task\_id='first\_task',

python\_callable=print\_text,

op\_kwargs= {

'text': 'Enfiler pantalon'

}

)

L'objectif de cet exercice est de recréer le DAG présenté dans la première partie de cet exercice em utilisant la fonction définie juste avant.

from airflow import DAG

from airflow.operators.python import PythonOperator

from airflow.utils.dates import days\_ago

my\_dag = DAG(

dag\_id='my\_dag\_of\_the\_morning',

description='My DAG to know what to do in the morning',

tags=['tutorial', 'datascientest'],

schedule\_interval=None,

default\_args={

'owner': 'airflow',

'start\_date': days\_ago(2),

}

)

def print\_text(text):

print(text)

texts = [

'Enfiler pantalon',

'Enfiler chaussette droite',

'Enfiler chaussure droite',

'Enfiler chaussette gauche',

'Enfiler chaussure gauche',

'Sortir'

]

ids = [

'pantalon',

'chaussette\_droite',

'chaussure\_droite',

'chaussette\_gauche',

'chaussure\_gauche',

'sortir'

]

task1, task2, task3, task4, task5, task6 = [

PythonOperator(

dag=my\_dag,

task\_id=i,

python\_callable=print\_text,

op\_kwargs={

'text': t

}

) for t, i in zip(texts, ids)

]

task1 >> task2

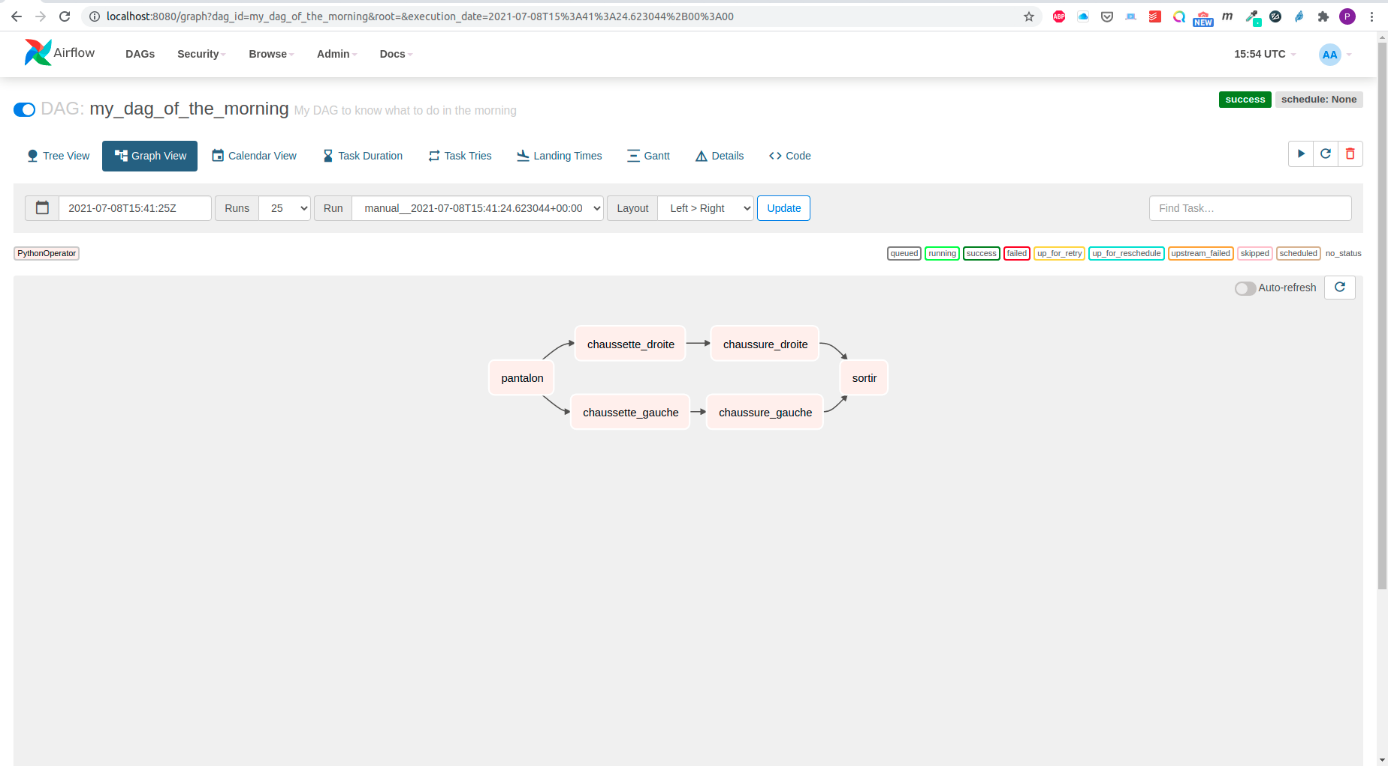
task1 >> task4

task2 >> task3

task4 >> task5

task3 >> task6

task5 >> task6



On peut noter qu'on aurait pu simplifier l'écriture des dépendances en faisant:

task1 >> [task2, task4]

task2 >> task3

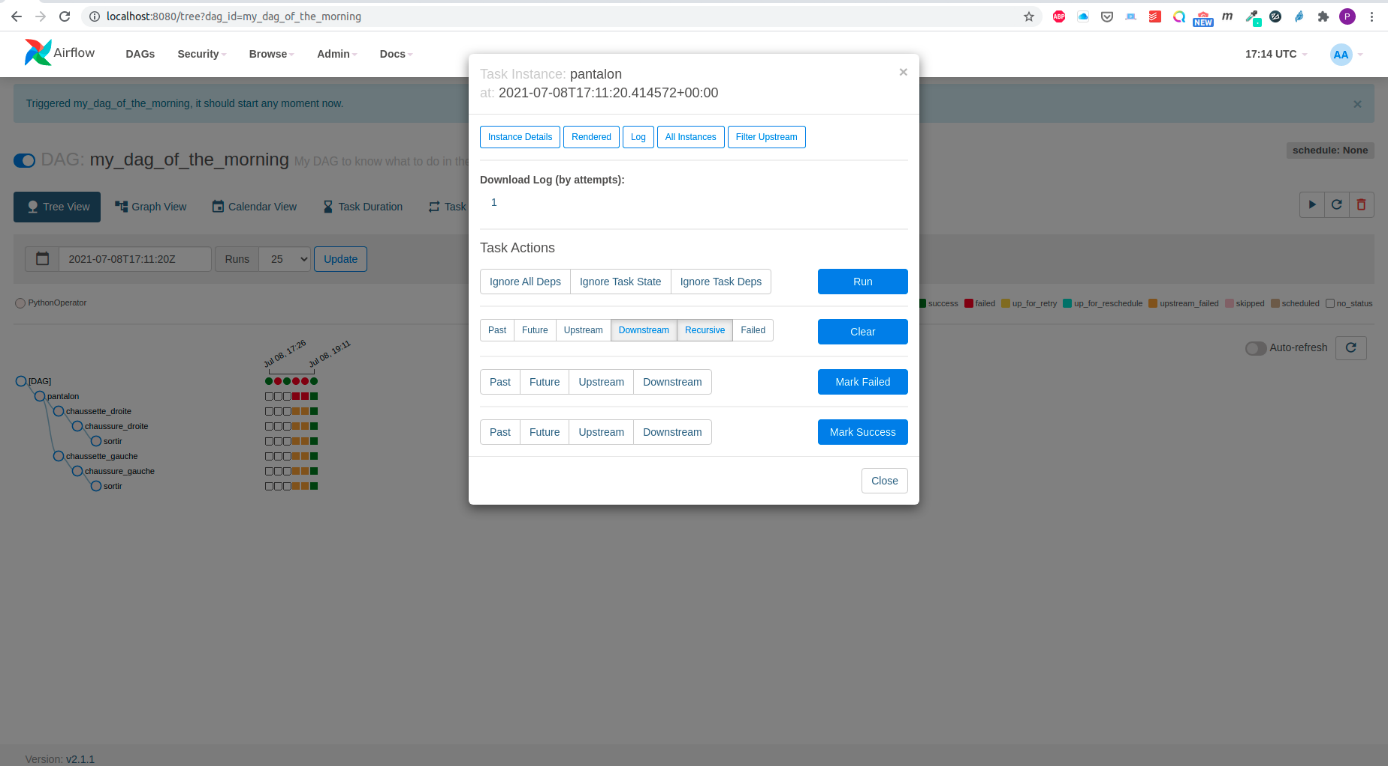
task4 >> task5

[task3, task5] >> task6

Exécutez le DAG

La vue Gantt est très intéressante puisqu'on voit bien les tâches qui ont lieu en même temps et celles qui doivent attendre qu'une autre tâche soit terminée pour commencer.

puis l'onglet Tree View, cliquez sur une des tâches (carré vert)



On peut aussi accéder aux détails de la Task instance depuis ce menu.

Vérifiez que le texte a bien été imprimé dans les logs

### Conclusion

Dans cette partie, nous avons abordé les concepts basiques de Airflow. Nous avons vu la notion de DAG qui est une succession organisée de Task, elles-mêmes implémentées à partir d'Operator. Nous avons aussi vu comment enregistrer un DAG et le lancer depuis l'interface graphique. Dans la suite, nous verrons d'autres options pour créer des workflows complexes.

## 3. DAGs avancés

Dans la partie précédente, nous avons vu comment créer un DAG simple. Nous allons voir dans cette partie comment rendre ce DAG récurrent et comment gérer les conditions, entre autres, et comment relancer les Task qui ont échoué.

### DAGs récurrents

Dans la partie précédente, nous avons choisi de créer un DAG avec un schedule\_interval fixé à la valeur None. Ce DAG ne s'exécutait donc qu'une fois. En réalité, cet argument peut prendre les mêmes valeurs qu'un Job CRON (cf cours Linux et Bash).

Pour rappel, les Jobs CRON peuvent être orchestrés avec un string (un [outil](https://crontab.guru/) pour s'entraîner à former des expressions CRON). Ce string est composé de 5 champs: minutes, heures, jour du mois, mois jour de la semaine. Pour chacun des champs, on peut indiquer qu'il peut prendre:

* n'importe quelle valeur avec \*
* les nombres de a à b avec a-b
* les valeurs a, b, c avec a,b,c
* la mention de / permet d'éviter certaines valeurs intermédiaires

Ainsi, par exemple, 23 0-20/2 \* \* 1,6-7 signifie toutes les deux heures à Xh23 entre minuit et 20h les lundis, samedis et dimanches.

1. Créez un DAG contenant une tâche qui imprimera l'heure et le jour
2. Paramétrez ce DAG de manière à ce qu'il s'exécute toutes les minutes

from airflow import DAG

from airflow.utils.dates import days\_ago

from airflow.operators.python import PythonOperator

import datetime

my\_dag = DAG(

dag\_id='minutely\_dag',

description='My DAG that\'s triggered every minute',

tags=['tutorial', 'datascientest'],

schedule\_interval='\* \* \* \* \*',

default\_args={

'owner': 'airflow',

'start\_date': days\_ago(0, minute=1),

}

)

def print\_date():

print(datetime.datetime.now())

my\_task = PythonOperator(

task\_id='print\_date\_task',

dag=my\_dag,

python\_callable=print\_date

)

Une fois que ce DAG est créé, nous allons pouvoir enlever la pause du DAG. On remarque que le nombre d'exécutions du DAG explose: Airflow tente en effet de rattraper les exécutions qu'il n'a pas pu faire depuis la start\_date car il était en pause. En se rendant dans l'onglet Details, on peut voir la start\_date: elle correspond à 00:01 ce jour-ci.

Pour éviter ce comportement, on peut modifier l'argument catchup lors de la définition du DAG et le mettre à la valeur False:

Renommez le DAG en ajoutant l'argument catchup=False, ajoutez-le à Airflow et enlevez la pause

from airflow import DAG

from airflow.utils.dates import days\_ago

from airflow.operators.python import PythonOperator

import datetime

my\_dag = DAG(

dag\_id='minutely\_dag\_without\_catchup',

description='My DAG that\'s triggered every minute',

tags=['tutorial', 'datascientest'],

schedule\_interval='\* \* \* \* \*',

default\_args={

'owner': 'airflow',

'start\_date': days\_ago(0, minute=1),

},

catchup=False

)

def print\_date():

print(datetime.datetime.now())

my\_task = PythonOperator(

task\_id='print\_date\_task',

dag=my\_dag,

python\_callable=print\_date

)

Nous allons à présent définir un DAG dont l'exécution prend plus de temps que le schedule\_interval prévu. Notons que l'on peut définir le schedule\_interval avec un objet de type timedelta.

Créez ce DAG et exécutez-le

from airflow import DAG

from airflow.utils.dates import days\_ago

from airflow.operators.python import PythonOperator

import datetime

import time

my\_dag = DAG(

dag\_id='too\_long\_dag',

description='My DAG that\'s triggered every 10 seconds',

tags=['tutorial', 'datascientest'],

schedule\_interval=datetime.timedelta(seconds=10),

default\_args={

'owner': 'airflow',

'start\_date': days\_ago(0, minute=1),

},

catchup=False

)

def sleep\_20\_seconds():

time.sleep(20)

my\_task = PythonOperator(

task\_id='sleep\_20\_seconds',

dag=my\_dag,

python\_callable=sleep\_20\_seconds

)

On peut voir que le DAG prend au moins 20 secondes pour tourner alors qu'il se lance toutes les 10 secondes. Ce comportement ne pose pas de problème à Airflow: on a deux exécutions qui ont lieu en même temps. Il faut toutefois noter qu'un trop grand nombre de tâches exécutées parallélement pourrait incapaciter la machine.

Les DAGs récurrents ne sont donc pas très difficiles à créer.

### Workflows conditionnels

Dans cette partie, nous allons voir comment définir des conditions en cas d'échecs, de réussite des différentes tâches d'un DAGs. Pour cela, nous allons utiliser les fonctions suivantes:

def successful\_task():

print('success')

def failed\_task():

raise Exception('This task did not work!')

import random

def random\_fail\_task():

random.seed()

if random.random() < .9:

raise Exception('This task randomly failed')

La première fonction fonctionnera toujours, la deuxième sera toujours un échec et la dernière sera en échec dans 90% des cas.

Nous allons tout d'abord créer un DAG qui va utiliser 3 tâches:

import random

from airflow import DAG

from airflow.utils.dates import days\_ago

from airflow.operators.python import PythonOperator

my\_dag = DAG(

dag\_id='fork1\_dag',

tags=['tutorial', 'datascientest'],

schedule\_interval=None,

default\_args={

'owner': 'airflow',

'start\_date': days\_ago(0, minute=1)

},

catchup=False

)

def successful\_task():

print('success')

def failed\_task():

raise Exception('This task did not work!')

def random\_fail\_task():

random.seed()

if random.random() < .9:

raise Exception('This task randomly failed')

task1 = PythonOperator(

task\_id='task1',

python\_callable=successful\_task,

dag=my\_dag

)

task2 = PythonOperator(

task\_id='task2',

python\_callable=failed\_task,

dag=my\_dag

)

task3 = PythonOperator(

task\_id='task3',

python\_callable=successful\_task,

dag=my\_dag

)

[task1, task2] >> task3

Créez et déclenchez ce DAG

Le DAG ne peut pas s'exécuter correctement: la tâche task2 va forcément échouer et empêcher l'exécution de la tâche task3. En fait, ce comportement est dû à la valeur par défaut de l'argument trigger\_rule. Cet argument permet en effet de spécifier les conditions de l'exécution d'une tâche:

all\_success: (valeur par défaut) toutes les tâches parentes doivent avoir réussi

all\_done: toutes les tâches parentes ont été exécutées (peu importe le statut)

one\_success: au moins une tâche parente a réussi

one\_failed: au moins une tâche parente a échoué

none\_failed: toutes les tâches parentes ont réussi ou ont été passées.

...

Cet argument peut être passé dans le default\_args du DAG ou dans la définition des différentes tâches.

Modifiez la tâche task3 en passant à cet argument la valeur "all\_done" et relancez le DAG

task3 = PythonOperator(

task\_id='task3',

python\_callable=successful\_task,

dag=my\_dag,

trigger\_rule='all\_done'

)

On peut voir, qu'à présent, la tâche task3 est bien exécutée.

En utilisant l'argument trigger\_rule, créez un DAG qui commence par une tâche pouvant échouer de manière aléatoire. Si la tâche échoue, une tâche task2 doit être exécutée et si elle réussie, une tâche task3 doit être exécutée. Dans tous les cas, une tâche task4 doit être lancée après. On pourra prendre un schedule\_interval de 10 secondes pour pouvoir voir les différents comportements

import random

from airflow import DAG

from airflow.utils.dates import days\_ago

from airflow.operators.python import PythonOperator

import datetime

my\_dag = DAG(

dag\_id='fork1\_dag',

tags=['tutorial', 'datascientest'],

schedule\_interval=datetime.timedelta(seconds=10),

default\_args={

'owner': 'airflow',

'start\_date': days\_ago(0, minute=1)

},

catchup=False

)

def successful\_task():

print('success')

def failed\_task():

raise Exception('This task did not work!')

def random\_fail\_task():

random.seed()

a = random.random()

print(a)

if a < .9:

raise Exception('This task randomly failed')

task1 = PythonOperator(

task\_id='task1',

python\_callable=random\_fail\_task,

dag=my\_dag

)

task2 = PythonOperator(

task\_id='task2',

python\_callable=successful\_task,

dag=my\_dag,

trigger\_rule='all\_failed'

)

task3 = PythonOperator(

task\_id='task3',

python\_callable=successful\_task,

dag=my\_dag,

trigger\_rule='all\_success'

)

task4 = PythonOperator(

task\_id='task4',

python\_callable=successful\_task,

dag=my\_dag,

trigger\_rule='all\_done'

)

task1 >> [task2, task3]

[task2, task3] >> task4

Dans certains cas, une tâche est si importante qu'on ne peut pas simplement la laisser échouer. On peut demander à Airflow de relancer la tâche un certain nombre de fois, en précisant un temps d'attente entre deux passages. Pour définir ce comportement, on peut utiliser l'argument retries pour définir le nombre de nouvelles tentatives et retry\_delay, qui prend un objet de type datetime.timedelta, pour définir le délai entre deux tentatives, lors de la définition de la tâche.

Créez un DAG déclenché à la main avec une seule tâche qui échoue nécessairement. Airflow devra relancer la tâche 5 fois avec un interval de temps de 30 secondes. Une fois le DAG créé, ajoutez le à Airflow et lancez-le

from airflow import DAG

from airflow.utils.dates import days\_ago

from airflow.operators.python import PythonOperator

import datetime

my\_dag = DAG(

dag\_id='retries\_dag',

description='My DAG that will try but fail',

tags=['tutorial', 'datascientest'],

schedule\_interval=None,

default\_args={

'owner': 'airflow',

'start\_date': days\_ago(0, minute=1),

},

catchup=False

)

def failed\_task():

raise Exception('This task did not work!')

task1 = PythonOperator(

task\_id="my\_failed\_task",

python\_callable=failed\_task,

retries=5,

retry\_delay=datetime.timedelta(seconds=30),

dag=my\_dag

)

On constate qu'effectivement, la tâche a le statut up\_for\_retry et est bien lancée plusieurs fois. Airflow embarque aussi une option pour envoyer des mails en cas de nouvelle tentative ou en cas d'échec d'une tâche. Il suffit de remplir les arguments suivants:

* email avec une liste d'adresses e-mail destinataires
* email\_on\_retry (ou email\_on\_failure avec un booléen)

Pour que ces e-mails soient envoyés, il faut définir des variables d'environnement. Elles ont déja été définie dans le fichier docker-compose:

AIRFLOW\_\_SMTP\_\_SMTP\_HOST=smtp.gmail.com

AIRFLOW\_\_SMTP\_\_SMTP\_PORT=587

AIRFLOW\_\_SMTP\_\_SMTP\_USER=de.airflow@gmail.com

AIRFLOW\_\_SMTP\_\_SMTP\_PASSWORD=Airflow123

[AIRFLOW\_\_SMTP\_\_SMTP\_MAIL\_FROM=de.airflow@gmail.com](mailto:AIRFLOW__SMTP__SMTP_MAIL_FROM=de.airflow@gmail.com)

 Il vous est demandé de faire attention à l'utilisation de l'adresse e-mail de.airflow@gmail.com. Nous avons fait le choix de créer l'adresse e-mail pour vous simplifier l'exercice. Merci de ne pas l'utiliser pour des activités malveillantes ou de modifier les paramètres du compte par égard pour les autres apprenants.

Modifiez le DAG précédent pour que Airflow envoie un mail en cas de nouvelle tentative ou en cas d'échec de la tâche my\_failed\_task et en précisant votre adresse e-mail

task1 = PythonOperator(

task\_id="my\_failed\_task",

python\_callable=failed\_task,

retries=5,

retry\_delay=datetime.timedelta(seconds=30),

dag=my\_dag,

email\_on\_retry=True,

email=['john.doe@datascientest.com']

)

Nous avons vu dans cette partie comment programmer des DAG pour qu'ils s'exécutent à intervalles réguliers. Nous avons aussi vu comment gérer les échecs et réussites de nos DAG. Dans la suite, nous allons voir d'autres options pour adapter l'utilisation de Airflow à différents scenarii.

## 4. SubDags, Sensors et Xcom

### SubDags

Lorsque l'on automatise des processus, on peut avoir un ensemble de tâches qui reviennent plusieurs fois dans différents processus. Si un DAG est associé à chaque processus, on risque de se retrouver avec des ensembles de tâches répétées dans différents DAGs. Pour permettre de simplifier ces architectures, Airflow permet de lancer un DAG depuis un autre DAG. On parle alors de subdag.

Pour déclencher l'exécution d'un subdag dans un DAG, on peut utiliser l'opérateur SubDagOperator.

Pour créer un SubDag, il faut définir une fonction qui permet de retourner un DAG:

Dans un fichier my\_subdag.py, collez les lignes suivantes

from airflow import DAG

from airflow.operators.bash import BashOperator

from airflow.utils.dates import days\_ago

def create\_sub\_dag(dag\_id, schedule\_interval, start\_date):

my\_sub\_dag = DAG(

dag\_id=dag\_id,

schedule\_interval=schedule\_interval,

default\_args={

'start\_date': days\_ago(0)

}

)

task1 = BashOperator(

bash\_command="echo subdag task 1",

task\_id="my\_sub\_dag\_task1",

dag=my\_sub\_dag

)

task2 = BashOperator(

bash\_command="echo subdag task 2",

task\_id="my\_sub\_dag\_task2",

dag=my\_sub\_dag

)

task1 >> task2

return my\_sub\_dag

Dans un autre fichier parent\_dag.py, collez les lignes suivantes

from airflow import DAG

from airflow.utils.dates import days\_ago

from airflow.operators.subdag import SubDagOperator

from airflow.operators.bash import BashOperator

# importing DAG generating function

from my\_subdag import create\_sub\_dag

my\_parent\_dag = DAG(

dag\_id="parent\_to\_subdag",

schedule\_interval=None,

default\_args={

'owner': 'airflow',

'start\_date': days\_ago(0, 1)

}

)

task1 = SubDagOperator(

task\_id="my\_subdag",

subdag=create\_sub\_dag(

dag\_id=my\_parent\_dag.dag\_id + '.' + 'my\_subdag',

schedule\_interval=my\_parent\_dag.schedule\_interval,

start\_date=my\_parent\_dag.start\_date),

dag=my\_parent\_dag

)

task2 = BashOperator(

task\_id="bash\_task",

bash\_command="echo hello world from parent",

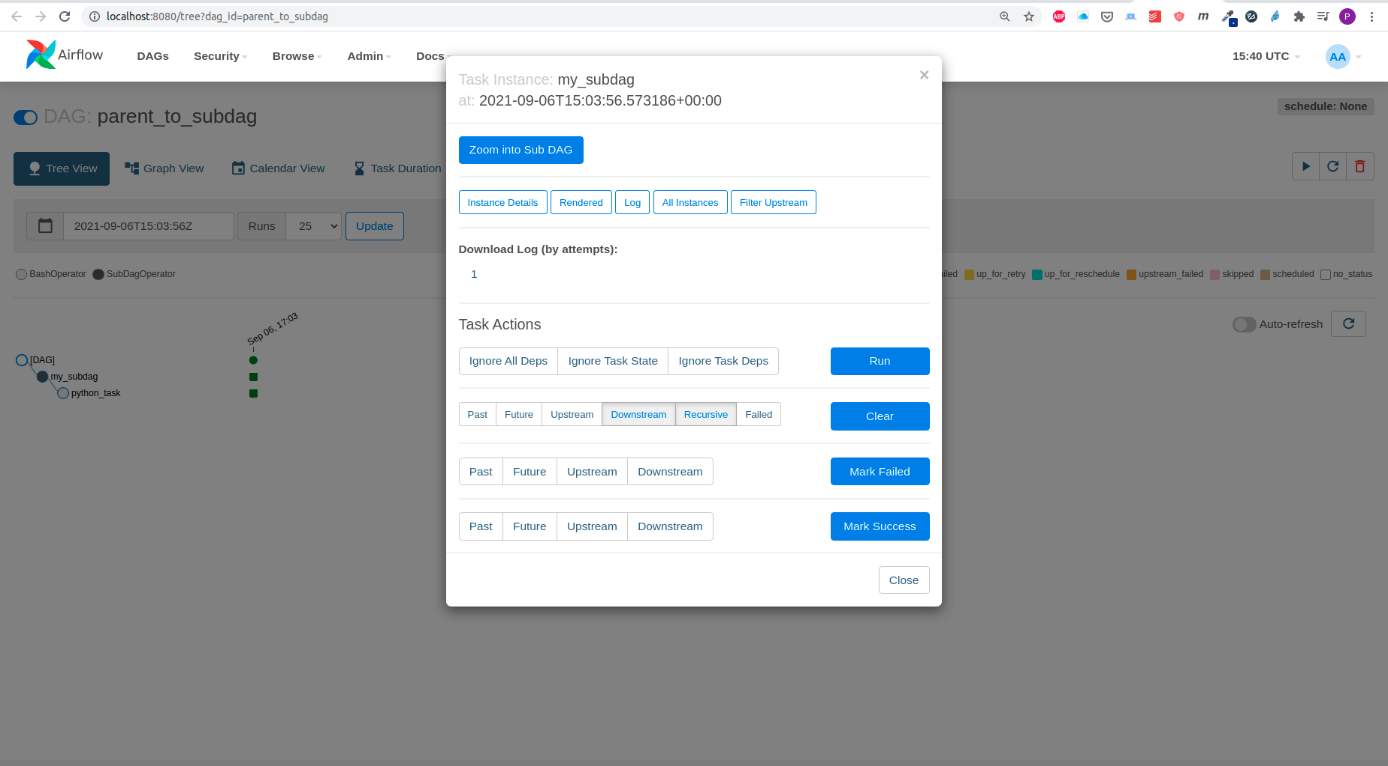
dag=my\_parent\_dag

)

task1 >> task2

Copiez/collez ces deux fichiers dans le dossiers dags et déclenchez le DAG parent\_to\_subdag

Sur l'interface, si on clique sur le DAG parent, on peut voir que le subDAG est accessible via le bouton Zoom into SubDAG.



Xcoms

Les Xcoms permettent à des données d'êtres transmises d'une tâche à une autre. On peut, par exemple, vouloir calculer une statistique sur une base de données (moyenne, médiane, ...) lors d'une tâche et transmettre cette valeur à une tâche suivante qui devra prendre en compte cette valeur pour décider des actions suivantes.

Pour créer des valeurs accessibles aux autres tâches du DAG, on utilise un simple return dans l'opérateur Airflow:

Créez ce DAG et exécutez-le

from airflow import DAG

from airflow.operators.python import PythonOperator

from airflow.utils.dates import days\_ago

import random

my\_dag = DAG(

dag\_id='simple\_xcom\_dag',

schedule\_interval=None,

start\_date=days\_ago(0)

)

def function\_with\_return():

return random.uniform(a=0, b=1)

my\_task = PythonOperator(

task\_id='python\_task',

dag=my\_dag,

python\_callable=function\_with\_return

)

Une fois que le DAG est lancé, on peut se rendre dans l'onglet Admin > XCom pour voir la valeur de la valeur retournée par la tâche. Toutefois, la valeur n'est pas nommée et donc difficilement réutilisable.

Pour nommer les XComs, on peut utiliser l'objet task\_instance. Cet objet est passé comme variable d'une fonction appelée par le PythonOperator. On peut alors utiliser la méthode push\_xcom de la task\_instance pour lui donner un nom.

Modifiez la fonction function\_with\_return avec les lignes suivantes:

def function\_with\_return(task\_instance):

task\_instance.xcom\_push(

key="my\_xcom\_value",

value=random.uniform(a=0, b=1)

)

Exécutez le DAG

On peut à présent voir la variable dans le même onglet que précédemment mais elle est nommée à présent. Pour accéder à un XCom, on peut utiliser la méthode xcom\_pull de l'objet task\_instance. On doit spécifier la clef (key) pour désigner la valeur que l'on veut retrouver mais il faut aussi spécifier un argument task\_ids qui prend une liste d'ids de tâches: en effet, plusieurs tâches peuvent créer un un XCom avec le même nom

Ajouter une deuxième tâche au DAG précédent qui affiche le contenu de la variable produite dans la première tâche.

def read\_data\_from\_xcom(task\_instance):

print(

task\_instance.xcom\_pull(

key="my\_xcom\_value",

task\_ids=['python\_task']

)

)

my\_task2 = PythonOperator(

task\_id="read\_Xcom\_value",

dag=my\_dag,

python\_callable=read\_data\_from\_xcom

)

my\_task >> my\_task2

On voit donc comment on peut faire pour passer des données d'une tâche à une autre. Notez que les données échangées par les XComs sont stockées dans la base de données interne d'Airflow: les valeurs d'un XCom doivent être limitées à 48KB. De plus, ces valeurs doivent être sérialisables: on ne peut pas mettre n'importe quel objet dans un XCom.

Transformez le DAG précédent de manière à ce qu'il enregistre un XCom dont la valeur est un dictionnaire avec plusieurs clefs

def function\_with\_return(task\_instance):

task\_instance.xcom\_push(

key="my\_xcom\_value",

value={

"hello": "world",

"bonjour": "le monde"

}

)

Notez que les XComs sont définies et accessibles à l'intérieur d'un même DAG. Pour définir des données accessibles à tous les DAGs, on peut utiliser des Variables (onglet Admin > Variables). Une fois une variable créée, on peut y accéder en utilisant:

from airflow.models import Variable

my\_variable\_value = Variable.get(key="my\_variable")

Pour définir une variable, on peut simplement utiliser Variable.set(key="key", value="value") ou en utilisant l'interface graphique.

Sensors et Connections

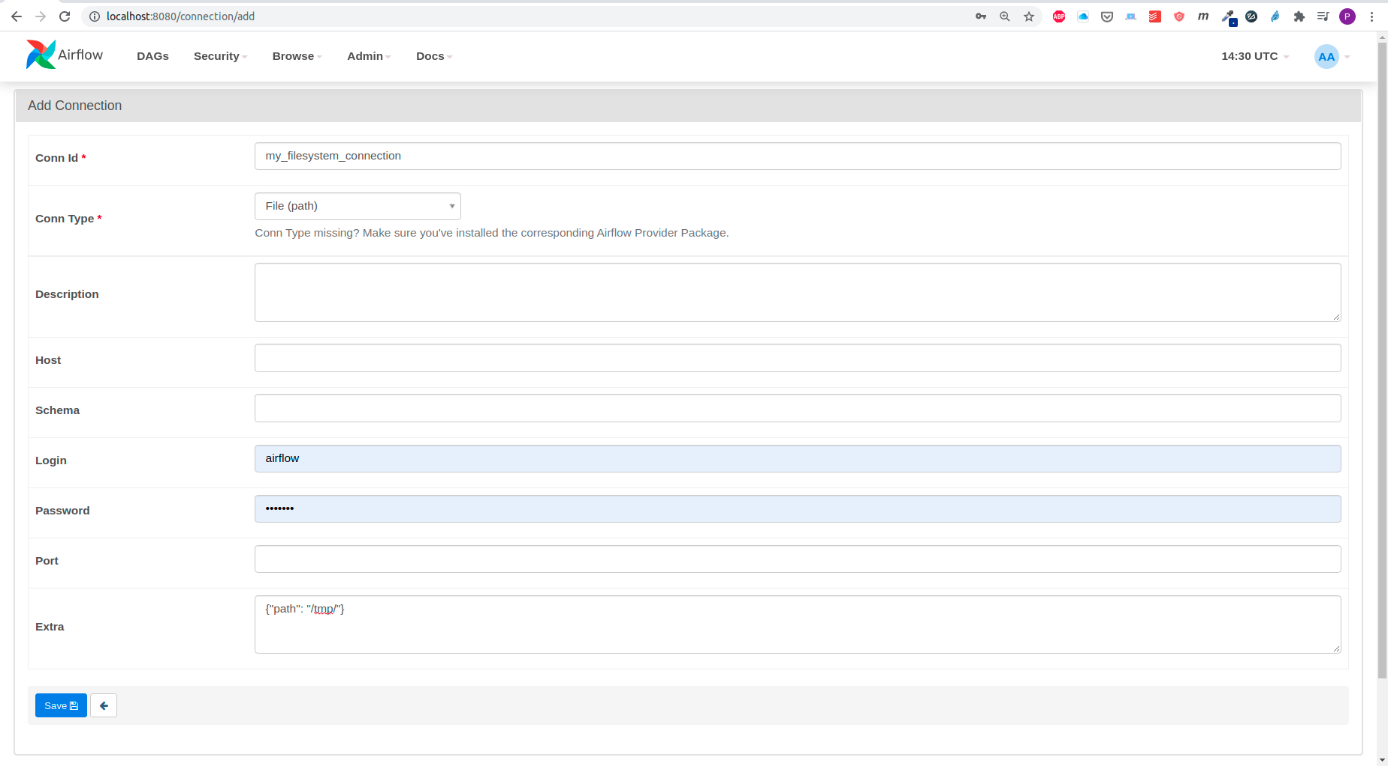
Pour certains processus, on peut vouloir créer des conditions basées sur la présence d'un fichier, de la disponibilité d'une base de données etc... On peut aussi vouloir vérifier qu'un fichier a bien été créé.

Pour effectuer ce genre de vérification, on peut utiliser un Sensor. Dans notre exemple, nous allons utiliser un FileSensor qui permet de vérifier qu'un fichier existe bien dans un répertoire donné. Pour cela, nous devons créer une Connection, c'est-à-dire un objet qui contient des informations de connexion relatives à un système de stockage par exemple.

Pour créer une Connection, il faut se rendre dans l'onglet Admin > Connection et cliquer sur le bouton en forme de signe +.

Créez une Connection avec les paramètres suivants

* Conn Id: identifiant de la Connection prend la valeur my\_filesystem\_connection
* Conn Type: type de Connection prend la valeur File (path)
* Login prend la valeur airflow
* Password prend la valeur airflow
* Extra doit contenir la valeur {"path": "/tmp/"}



Créez un DAG avec le FileSensor suivant

from airflow import DAG

from airflow.utils.dates import days\_ago

from airflow.sensors.filesystem import FileSensor

from airflow.operators.bash import BashOperator

my\_dag = DAG(

dag\_id='sensor\_dag',

schedule\_interval=None,

start\_date=days\_ago(0)

)

my\_sensor = FileSensor(

task\_id="check\_file",

fs\_conn\_id="my\_filesystem\_connection",

filepath="my\_file.txt",

poke\_interval=30,

dag=my\_dag,

timeout=5 \* 30,

mode='reschedule'

)

my\_task = BashOperator(

task\_id="print\_file\_content",

bash\_command="cat /tmp/my\_file.txt",

dag=my\_dag

)

my\_task >> my\_sensor

Notez l'utilisation des arguments poke\_interval qui définit le temps entre deux tentatives pour trouver le fichier, fs\_conn\_id pour définir la Connection à utiliser, timeout qui définit le temps au bout duquel la tâche est considérée comme un échec. Enfin le dernier argument mode dans lequel on passe la valeur "reschedule". Ce mode permet de ne pas prolonger l'exécution de la tâche entre deux tentatives de recherche du fichier: en effet, par défaut, la tâche continue tant que le fichier n'est pas trouvé ou que le timeout n'est pas atteint.

Exécutez le DAG

Lors de l'exécution de ce DAG, on voit que la tâche obtient le statut UP\_FOR\_RESCHEDULE. En allant dans les logs de la tâche, on peut voir que ses tentatives pour retrouver le fichier sont infructueuses.

Exécutez la commande docker exec -it airflow\_airflow-worker\_1 bash pour accéder au container qui fait tourner le script

Une fois connecté, nous allons pouvoir créer un fichier.

Exécutez la commande suivante

echo "hello world" >> /tmp/my\_file.txt

À présent, le fichier est trouvable et la tâche devrait réussir.

Les Sensors peuvent être très intéressants conditionner l'exécution d'un DAG à la présence/existence de données.

5. Concepts avancés

Dans cette partie, nous allons aborder rapidement quelques concepts supplémentaires de Airflow.

Gestion des utilisateurs

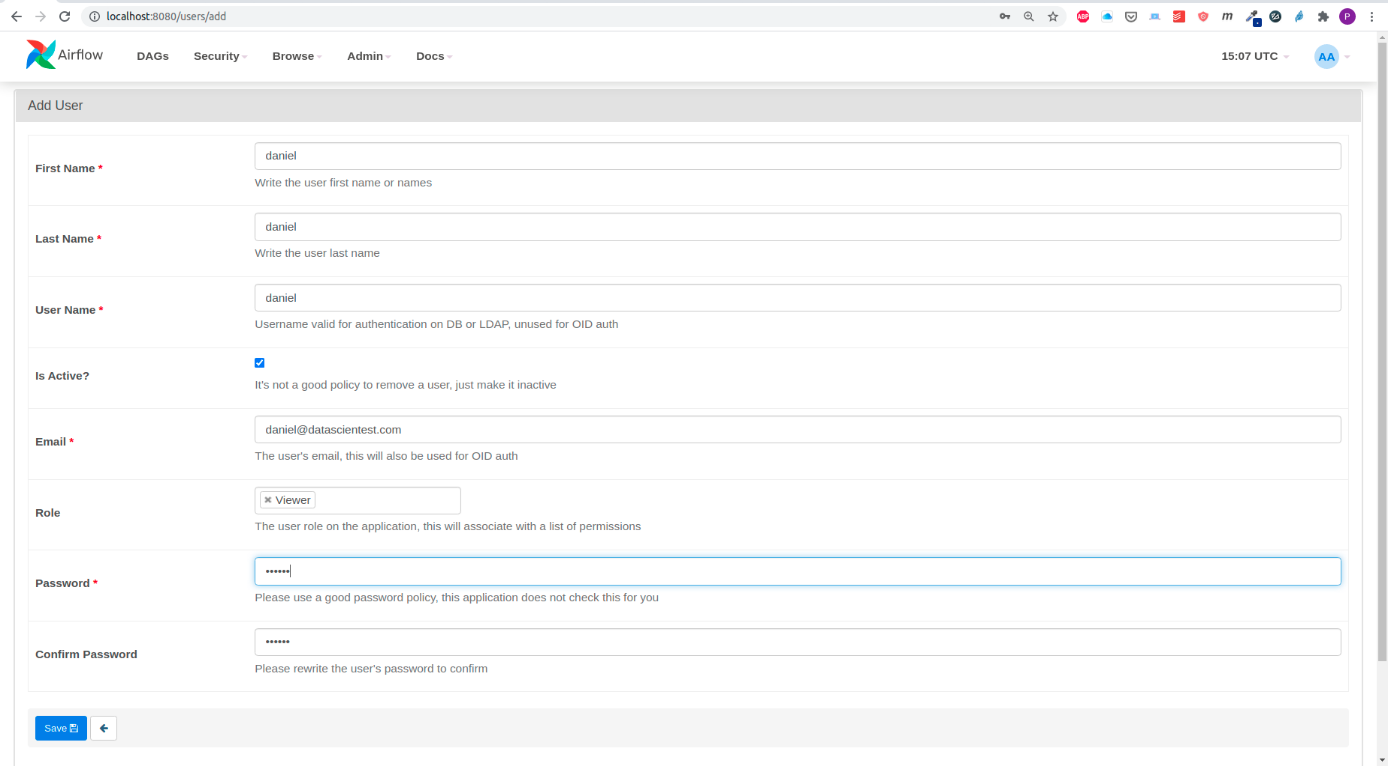
Le compte que l'on utilise depuis le début de ce cours est un compte avec rôle d'administrateur. En réalité, Airflow propose un système de rôle pour différencier les droits des utilisateurs.

Dans l'onglet Security, on peut accéder à la liste des utilisateurs, à la liste des rôles mais aussi à la liste des différents menus et des permissions associées à ces menus.

Rendez vous dans l'onglet List Roles

Les rôles définissent des ensembles de droits: on parle de Role Based Access Control. On peut alors associer un utilisateur à un rôle.

* Rendez vous dans l'onglet Security > List Users
* Créez un nouvel utilisateur avec les informations suivantes (on a pris daniel comme mot de passe)



Déconnectez vous en cliquant sur AA en haut à droite puis reconnectez vous en tant que daniel

On voit à présent que nos capacités sont très limitées: on peut voir les DAGs mais on ne peut pas agir sur eux.

Airflow offre un contrôle des utilisateurs très fin et très intéressant lorsqu'on doit mettre un serveur Airflow à disposition d'une organisation toute entière.

Documentation

Airflow permet de documenter les tâches ainsi que les DAGs de manière plus simple que simplement l'utilisation du code Python et des commentaires.

Enregistrez le DAG suivant

from airflow import DAG

from airflow.utils.dates import days\_ago

from airflow.operators.python\_operator import PythonOperator

import time

my\_dag = DAG(

dag\_id="documented\_dag",

doc\_md="""# Documented DAG

This `DAG` is documented and the next line is a quote:

> Airflow is nice

This DAG has been made:

\* by Paul

\* with documentation

\* with caution

""",

start\_date=days\_ago(0),

schedule\_interval=None

)

def sleep\_1\_sec():

time.sleep(1)

task1 = PythonOperator(

task\_id="sleep1",

python\_callable=sleep\_1\_sec,

doc\_md="""# Task1

Task that is used to sleep for 1 sec""",

dag=my\_dag

)

task2 = PythonOperator(

task\_id="sleep2",

python\_callable=sleep\_1\_sec,

doc="""Task 3

It has an ugly description.

""",

dag=my\_dag

)

Une fois que ce DAG est enregistré, on peut voir la documentation sur le DAG directement dans le menu du DAG (DAG Docs). On peut aussi voir la documentation des tâches en se rendant dans la Graph View et en cliquant sur une tâche puis sur Instance Details. Avec doc\_md, la documentation est traitée comme du markdown et est donc mise en forme.

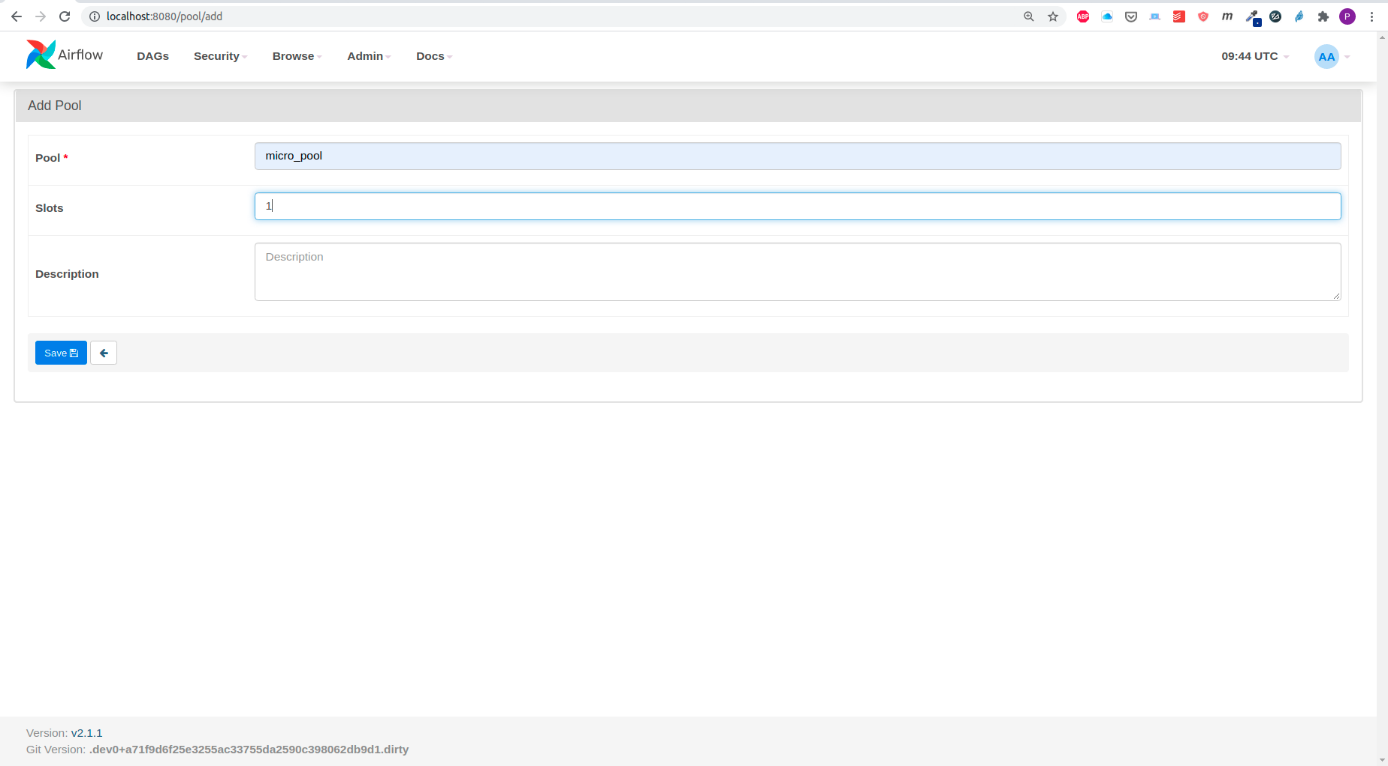
Documenter les tâches ainsi que les DAG peut être très intéressant dans une organisation avec de nombreux utilisateurs de Airflow et de nombreux processus gérés par Airflow.

Environnement d'exécution et Pools

Dans ce cours, nous avons exécuté les tâches directement dans l'environnement local. L'utilisation d'Executor permet de changer l'environnement d'exécution: on peut par exemple exécuter des tâches en utilisant Kubernetes ou d'autres gestionnaires de resources.

Les Pools sont utilisés pour limiter l'exécution de tâches concurrentes: dans certains cas, on préfére que les tâches soient exécutées séquentiellement plutôt que parallélement. Un Pool permet de définir un nombre de slot c'est à dire de tâche qui peuvent s'exercer parallélement.

En utilisant l'onglet Admin > Pools, créez un Pool appelé micro\_pool contenant un seul slot.



Créez un DAG avec le code suivant

from airflow import DAG

from airflow.operators.python import PythonOperator

from airflow.utils.dates import days\_ago

import time

def wait\_1\_minute():

time.sleep(60)

my\_dag = DAG(

dag\_id="concurrent\_dag",

schedule\_interval=None,

start\_date=days\_ago(0),

default\_args={

'pool': 'micro\_pool'

}

)

task1 = PythonOperator(

task\_id="wait1",

python\_callable=wait\_1\_minute,

dag=my\_dag

)

task2 = PythonOperator(

task\_id="wait2",

python\_callable=wait\_1\_minute,

dag=my\_dag

)

task3 = PythonOperator(

task\_id="wait3",

python\_callable=wait\_1\_minute,

dag=my\_dag

)

Dans ce DAG, les 3 tâches peuvent être exécutées de manière parallèle. Toutefois, nous avons utilisé un Pool avec un seul slot: dans ce Pool, une seule tâche peut être exécutée à la fois. Ce DAG devrait donc prendre environ 3 minutes pour s'exécuter.

Déclenchez le DAG en faisant attention au temps d'exécution et au statut des tâches

Nous allons à présent supprimer cette assignation des tâche au Pool micro\_pool.

Commentez les lignes correspondant à l'argument default\_args dans la définition du DAG, mettez à jour le code du DAG dans Airflow et relancez le DAG

Cette fois-ci, l'exécution prend une minute puisque le Pool par défaut contient 128 slots. Les trois tâches peuvent s'exécuter de manière parallèle.

### Plugins

Les Plugins Airflow sont des processus qui permettent de relier Airflow à d'autres outils. Par exemple, si l'organisation a l'habitude de surveiller des processus en utilisant Prometheus, on peut utiliser un Plugin qui renverra des informations sur le statut du cluster Airflow et le statut des tâches en cours. On peut aussi facilement relier Airflow à certaines bases de données. On peut voir une liste de certains Plugins sur ce dépot [Github](https://github.com/airflow-plugins/).

### API et clients

Dans ce cours, nous avons surtout vu comment utiliser l'interface graphique de Airflow. Il existe un client en ligne de commande ainsi qu'une API REST. Ces clients ne sont pas dans le périmètre de ce cours mais vous pouvez retrouvez la documentation sur l'API REST dans l'onglet Docs et la documentation sur l'interface en ligne de commande [ici](https://airflow.apache.org/docs/apache-airflow/stable/usage-cli.html)

## 6. Conclusion et évaluation

### Conclusion

Dans ce cours, nous avons vu comment on pouvait atomiser un processus en différentes petites tâches qui peuvent communiquer entre elles, s'enchaînent et peuvent être conditionnées sur leur réussite ou échec. Nous avons vu comment Airflow apportait une plateforme de gestion des processus, des utilisateurs, des droits de ces utilisateurs.

Airflow est un outil très utilisé en Data Engineering aujourd'hui, notamment pour gérer les ETL et ELT récurrents ou non.

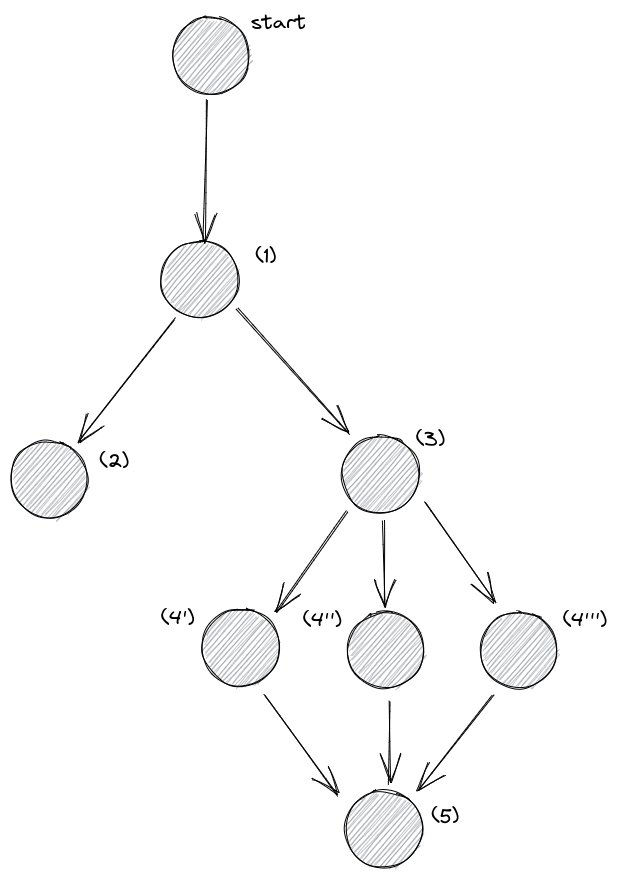
### Bonnes pratiques

Dans ce paragraphe, nous allons voir quelques bonnes pratiques dans l'utilisation d'Airflow:

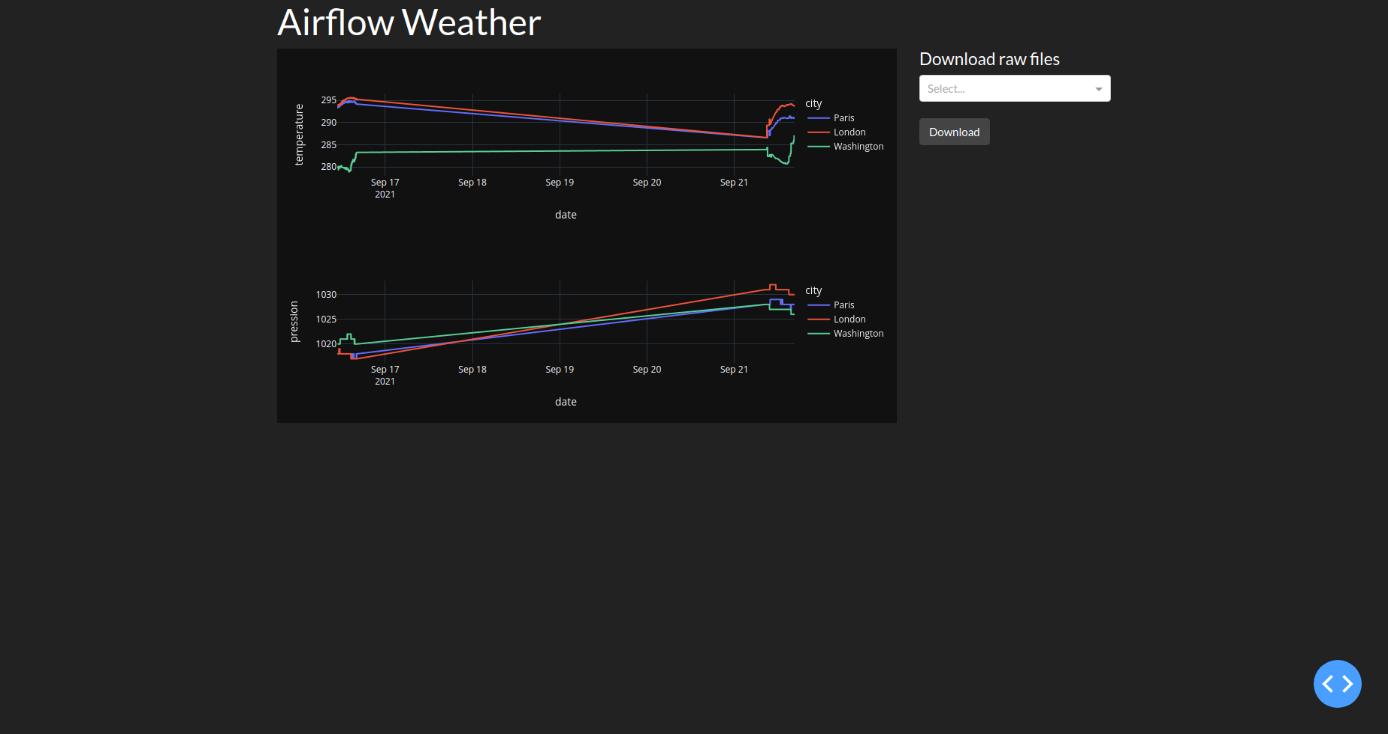
* réduire au maximum la taille des tâches pour qu'elles puissent tourner ou re-tourner de manière indépendantes
* éviter de faire tourner du code en dehors de la définition des tâches et des DAGs dans le dossier dags
* éviter d'utiliser Airflow pour des tâches sur de trop gros volumes de données: préférer lancer des tâches sur des moteurs de calcul dédiés (Spark, Hadoop, SQL, ...)
* contrairement à ce que nous avons fait dans le cours, préférer un start\_date statique
* passer la plupart des arguments communs aux différentes tâches dans l'argument default\_args dans la définition du DAG
* éviter de supprimer des tâches dans un DAG pour conserver l'historique d'exécution de ces tâches: préférer construire un nouveau DAG
* implémenter des tâches de vérifications avec des Sensors notamment après des insertions
* bien documenter les tâches et les DAG

### Évaluation

Pour l'évaluation de ce cours, nous allons construire un DAG qui permet de récupérer des informations depuis une API de données météo disponible en ligne, les stocke, les transforme et entraîne un algorithme dessus. Son architecture sera la suivante:



Ce DAG permet ainsi de nourrir un dashboard lancé dans un docker-compose.yml dédié et disponible sur le port 8050 de la machine. Ce DAG devra être exécuté toutes les minutes pour mettre à jour régulièrement le dashboard ainsi que le modèle de prédiction.



#### **Pré-requis**

Pour commencer l'évaluation, il faut télécharger un autre fichier docker-compose.yaml et créer des dossiers pour stocker les données:

  Attention ! Si vous souhaitez conserver le premier fichier docker-compose.yml utilisé dans le cours, vous pouvez utiliser un nouveau dossier. Il vous faudra, dans ce cas, refaire les étapes d'initialisation du premier cours !

  Si l'interface Airflow n'est pas fonctionnelle, c'est à cause des résidus du premier docker-compose.yml. Je vous invite à partir d'une machine vide, donc il faudra faire un reset. Avant de le faire, récupérez l'ensemble des fichiers qui vous sont importants.

Exécutez la commande suivante

# shutting down previous containers

docker-compose down

# deleting previous docker-compose

rm docker-compose.yaml

# downloading new docker-compose.yml file

wget https://dst-de.s3.eu-west-3.amazonaws.com/airflow\_avance\_fr/eval/docker-compose.yaml

# creating directories

mkdir clean\_data

mkdir raw\_files

echo -e "AIRFLOW\_UID=$(id -u)\nAIRFLOW\_GID=0" > .env

docker-compose up airflow-init

wget https://dst-de.s3.eu-west-3.amazonaws.com/airflow\_avance\_fr/eval/data.csv -O clean\_data/data.csv

echo '[]' >> raw\_files/null\_file.json

# starting docker-compose

docker-compose up -d

#### **(1) Récupération de données depuis l'API OpenWeatherMap**

[OpenWeatherMap](https://openweathermap.org/) est une API de données météo. Avec un plan gratuit, on peut requêter l'API jusqu'à 60 fois par minute, ce qui sera amplement suffisant pour notre cas d'usage.

Créez un compte sur OpenWeatherMap et connectez vous au compte

Une fois connecté, on peut se rendre dans le menu [My API keys](https://home.openweathermap.org/api_keys). Vous pourrez alors voir votre clef d'API par défaut.

Pour obtenir les données en temps réel de la météo quelque part dans le monde, on peut faire une requête sur l'API current dont la documentation est disponible [ici](https://openweathermap.org/current). Pour obtenir des données, on doit faire une requête HTTP de type GET à l'URL suivante, https://api.openweathermap.org/data/2.5/weather?q=city\_name&appid=API\_key, en remplaçant votre api\_key par la clef d'API obtenue à l'étape précédente et city\_name par le nom d'une ville.

Essayez la commande suivante en remplaçant api\_key par votre clef

curl -X GET <https://api.openweathermap.org/data/2.5/weather?q=paris&appid=API_key>

La première tâche (1) consiste donc en la récupération des données depuis OpenWeatherMap: on pourra faire plusieurs requêtes pour avoir les données sur plusieurs villes. Pour cela, on stockera une Variable nommée cities. Dans notre solution, nous utilisons ['paris', 'london', 'washington'] mais vous pouvez choisir d'autres villes.

Cette tâche doit stocker les données au format json dans un fichier dont le nom correspond à l'heure et la date de la récolte de donnée: 2021-01-01 00:00.json. Ce fichier devra être créé dans le dossier /app/raw\_files.

On pourra utiliser les librairies requests et json.

#### **(2) et (3) transformation des données**

Les tâches (2) et (3) consiste à lire le contenu du dossier /app/raw\_files et transformer les données au format csv.

On pourra utiliser la fonction suivante:

def transform\_data\_into\_csv(n\_files=None, filename='data.csv'):

parent\_folder = '/app/raw\_files'

files = sorted(os.listdir(parent\_folder), reverse=True)

if n\_files:

files = files[:n\_files]

dfs = []

for f in files:

with open(os.path.join(parent\_folder, f), 'r') as file:

data\_temp = json.load(file)

for data\_city in data\_temp:

dfs.append(

{

'temperature': data\_city['main']['temp'],

'city': data\_city['name'],

'pression': data\_city['main']['pressure'],

'date': f.split('.')[0]

}

)

df = pd.DataFrame(dfs)

print('\n', df.head(10))

df.to\_csv(os.path.join('/app/clean\_data', filename), index=False)

La tâche (2) devra prendre les 20 derniers fichiers, les concaténer et les transformer dans un fichier data.csv alors que la tâche (3) devra prendre en compte tous les fichiers dans le dossier et créer un fichier fulldata.csv.

Le premier fichier sera utilisé par le dashboard pour visualiser les dernières observations alors que le deuxième fichier sera utilisé dans la suite du DAG pour entraîner un algorithme.

#### **(4) et (5) entraînement de modèles et sélection du plus performant**

Les tâches (4'), (4'') et (4''') correspondent à l'entraînement de différents modèles de régression (respectivement LinearRegression, DecisionTreeRegressor, RandomForestRegressor). Une fois ces modèles entraînés et testés avec une méthode de validation croisée, on pourra utiliser un XCom pour transmettre la valeur du score de validation. La tâche (5) permettra de choisir le meilleur modèle, de le réentraîner sur toutes les données et de le sauvegarder.

Ces deux tâches pourraient classiquement être traitée par un script similaire à celui-ci:

import pandas as pd

from sklearn.model\_selection import cross\_val\_score

from sklearn.linear\_model import LinearRegression

from sklearn.tree import DecisionTreeRegressor

from joblib import dump

def compute\_model\_score(model, X, y):

# computing cross val

cross\_validation = cross\_val\_score(

model,

X,

y,

cv=3,

scoring='neg\_mean\_squared\_error')

model\_score = cross\_validation.mean()

return model\_score

def train\_and\_save\_model(model, X, y, path\_to\_model='./app/model.pckl'):

# training the model

model.fit(X, y)

# saving model

print(str(model), 'saved at ', path\_to\_model)

dump(model, path\_to\_model)

def prepare\_data(path\_to\_data='/app/clean\_data/fulldata.csv'):

# reading data

df = pd.read\_csv(path\_to\_data)

# ordering data according to city and date

df = df.sort\_values(['city', 'date'], ascending=True)

dfs = []

for c in df['city'].unique():

df\_temp = df[df['city'] == c]

# creating target

df\_temp.loc[:, 'target'] = df\_temp['temperature'].shift(1)

# creating features

for i in range(1, 10):

df\_temp.loc[:, 'temp\_m-{}'.format(i)

] = df\_temp['temperature'].shift(-i)

# deleting null values

df\_temp = df\_temp.dropna()

dfs.append(df\_temp)

# concatenating datasets

df\_final = pd.concat(

dfs,

axis=0,

ignore\_index=False

)

# deleting date variable

df\_final = df\_final.drop(['date'], axis=1)

# creating dummies for city variable

df\_final = pd.get\_dummies(df\_final)

features = df\_final.drop(['target'], axis=1)

target = df\_final['target']

return features, target

if \_\_name\_\_ == '\_\_main\_\_':

X, y = prepare\_data('./clean\_data/fulldata.csv')

score\_lr = compute\_model\_score(LinearRegression(), X, y)

score\_dt = compute\_model\_score(DecisionTreeRegressor(), X, y)

# using neg\_mean\_square\_error

if score\_lr < score\_dt:

train\_and\_save\_model(

LinearRegression(),

X,

y,

'/app/clean\_data/best\_model.pickle'

)

else:

train\_and\_save\_model(

DecisionTreeRegressor(),

X,

y,

'/app/clean\_data/best\_model.pickle'

)

On pourra réutiliser les fonctions utilisées dans ce script pour créer ces tâches.

#### **Quelques conseils et rappels**

* n'hésitez pas à tester vos fonctions en dehors de Airflow: vous perdrez beaucoup moins de temps à les utiliser en dehors de Airflow plutôt qu'à l'intérieur d'un DAG
* une fois que le code fonctionne, on peut implémenter une tâche pour que l'intégrer à un DAG: essayez de faire tourner le DAG avec une première tâche puis une seconde ...
* en suivant exactement le script utilisé dans les tâches (4) et (5), il faut avoir au moins 15 observations: lorsque vous avez construit la tâche (1) et qu'elle fonctionne correctement, laissez-la tourner une quinzaine de minutes seule.
* Le dashboard permet de concrétiser le cas d'utilisation mais c'est aussi un bon outil pour voir si le DAG fonctionne correctement: vous pouvez y voir les données traitées mais aussi les données brutes.
* la clef d'API est limitée en nombre de requêtes par minutes. Si jamais vous faîtes trop de requêtes en même temps, la clef sera indisponible pendant quelques instants.
* vous pouvez toujours vérifier que les fichiers sont bien créés en observant le contenu des dossiers clean\_data et raw\_files qui sont montés sur les dossiers /app/clean\_data et /app/raw\_files de docker-compose. (Si vous n'êtes pas familier avec docker-compose, considérer que depuis l'extérieur de Airflow, /app/clean\_data correspond au dossier clean\_data présent dans le répertoire de travail)
* merci de respecter le nom des Variable pour que la correction soit plus simple pour nous !

#### **Attendus**

Pour valider le module, il faut rendre un fichier Python contenant la définition compléte du DAG. Vous pouvez ajouter un fichier explicatif des différents choix que vous avez faits.