TP Airflow

Exercice 1 : Installer et lancer Airflow

- Lancer un service jupyter-notebook avec le port 5000 d'ouvert
- Installez Apache Airflow (apache-airflow) en utilisant pip

> Ouvrez un terminal linux puis Lancer la commande `pip install apache-airflow`

> Lancer la commande : `sudo apt update && sudo apt install -y tzdata` puis entrer les valeurs correspondantes pour spécifier la localisation (`8` puis `37`)

- se placer dans le dossier `work` et lancer dans un terminal le scheduler

> Ouvrez un terminal linux et lancer la commande : `airflow scheduler` et entrer `y` pour initialiser la base de données (ne pas l'arrêter)

- Lancer dans un autre terminal le webserver sur le port `5000`

> Ouvrez un autre terminal linux puis lancer la commande : `airflow webserver --port 5000` (ne pas l'arrêter)

- Créer un dossier `dags` dans votre répértoire

> Ouvrez un autre terminal puis lancer la commande : `mkdir dags`

- Changer dans la configuration airflow

`dags_folder = /home/onyxia/work/dags load_examples = False`

> Lancer la commande : `nano ../airflow/airflow.cfg` puis modifier les lignes correspondantes. Pour enregistrer `Crtl+x`, puis appuyer sur `y` pour accepter et enfin `Entrée` pour sortir. Ensuite, arrêter le scheduler (`Crtl+c`). Lancer ensuite la commande `airflow db reset` et entrer `y`. Relancer le scheduler.

- Lancer la commande suivante pour créer un utilisateur :

```
airflow users  create --role Admin --username admin --email admin --
firstname admin --lastname admin --password airflow1234!
```

> Entrer la commande précédente pour générer un utilisateur sur airflow

- Connecter vous ensuite à airflow sur l'url exposé sur le port 5000

> Connecter vous à Airflow sur la seconde URL exposée avec les identifiants précédents `(admin/airflow1234!)`.

Exercice 2 : Lancer un dag

- Ecrire un dag qui lance deux task python sucessives qui print "OK"

```
import pendulum
from airflow.decorators import dag, task


@task()
def etape1():
    print('OK')

@task()
def etape2():
    print('OK')

@dag("test_dag",
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def test():
    a = etape1()
    b = etape2()

    a >> b

test()
```

> Créer un fichier `dag.py` dans le dossier `dags` et copier le contenu du dag dans ce fichier. Si le dag n'apparaît pas il se peut que jupyter génère un fichier en doublon ce qui entraîne une erreur d'import. Pour le supprimer `rm -rf /home/onyxia/work/dags/.ipynb_checkpoints`

Lancer ce dag manuellement dans l'interface utilisateur puis trouver dans les logs la valeur affichée.

> Pour lancer le DAG manuellement, cliquer sur le DAG et cliquer sur le triangle bleu en haut à droite de la page. Pour consuler les logs : `Cliquer sur le carré vert de l'étape 1 -> Logs`. La ligne s'affiche : `[2023-11-22, 19:40:03 UTC] {logging_mixin.py:154} INFO - OK`

> A noter : si une page d'erreur s'affiche, retourner à l'adresse `https://<url>/home`. Elle permet de vérifier si un DAG est en erreur lors de l'import.

Exercice 3 : Planification d'un DAG

Modifiez le DAG pour planifier l'exécution quotidienne toutes les 10 minutes (le site `crontab.guru` pourra vous aider).

> Modifier le dag en y ajoutant

```
@dag("test_dag",
    schedule="*/10 * * * *", # <--- every 10 minutes
```

```
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
```

Exercice 4 : Utilisation d'un opérateur

Ajoutez une troisième tâche au DAG qui exécute une commande Bash : `date` (avec BashOperator).

```
import pendulum
from airflow.decorators import dag, task
from airflow.operators.bash import BashOperator # <--- import du provider

@task()
def etape1():
    print('OK')

@task()
def etape2():
    print('OK')

@dag("test_dag",
    schedule="*/10 * * * *",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def test():
    a = etape1()
    b = etape2()
    c = BashOperator(task_id="bash_task", bash_command="date")  # <---
Ajout du BashOperator qui lance la commande linux `date`
    a >> b >> c # <--- Ajout de la dépendance de la task

test()
```

Relancer le dag manuellement.

Exercice 5 : Les XComs

Modifiez le DAG pour que la première tache retourne la valeur `1` et que la seconde tâche utilise le résultat de la première tâche en tant que paramètre et lui ajoute `10`. Relancer le dag manuellement.

```
import pendulum
from airflow.decorators import dag, task
from airflow.operators.bash import BashOperator

@task()
def etape1():
```

```
    return 1 # <---- etape 1 retourne 1

@task()
def etape2(number: int): # <-- etape 2 prend en paramètre number qui est
de type int
    print(number + 10) # <--- affiche number + 10

@dag("test_dag",
    schedule="*/10 * * * *",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def test():
    a = etape1()
    b = etape2(number=a) # <--- Ajout du retour de l'étape 1 à l'étape 2
    c = BashOperator(task_id="bash_task", bash_command="date")
    a >> b >> c

test()
```

Trouver la valeur du XCom de la première tâche dans l'interface Airflow.

Exercice 6 : Variabiliser les DAGs

Ajouter une variable airflow dans l'UI dont le nom est `numero` et sa valeur est `1`. Faite retourner la valeur de la variable dans la première tache du DAG (documentation).

> Aller dans l'onglet `Admin > Variables > +` puis ajouter dans `Key` : `number` et `Val` : `1`

```
import pendulum
from airflow.decorators import dag, task
from airflow.operators.bash import BashOperator
from aiflow.models import Variable # <---- import de Variable

@task()
def etape1():
    return int(Variable.get("number")) # <---- retourne la valeur de la
variable number et la convertit en entier

@task()
def etape2(number: int):
    print(number + 10)

@dag("test_dag",
    schedule="*/10 * * * *",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def test():
```

```
    a = etape1()
    b = etape2(number=a)
    c = BashOperator(task_id="bash_task", bash_command="date")
    a >> b >> c

test()
```

Où faut t'il placer la récupération de la valeur de la variable ? Pourquoi ?

Note : consulter les best pratices Airflow dans la documentation

> Il faut placer la récupération de la variable à l'intérieur de la `task` python, car les variables sont stockées en base de données et le contenu des tasks ne sont lancés qu'à la lecture du DAG (cf. https://airflow.apache.org/docs/apache-airflow/stable/best-practices.html#airflow-variables)

Exercice 7 : Ajouter une connexion et l'utiliser

1. Installer le provider amazon S3 `pip install 'apache-airflow[amazon]'` et modifier dans le fichier `airflow.cfg` le paramètre `test_connection = Enabled`

> Comme précédemment, `nano ../airflow/airflow.cfg` puis changer le paramètre.

2. Relancer le webserver et le scheduler

3. Ajouter une connexion au stockage S3 de la plateforme appelée (`minio`) (documentation). *Attention : le test de connexion ne fonctionne pas avec minio.*

> Lancer la commande `env | grep AWS` et remplir les variable dans les champs correspondants (`$VARIABLE`) et sauvegarder. Si la connexion ne fonctionne plus, aller dans le menu d'onyxia puis `Mon Compte` -> `Connexion au stockage` et copier les informations dans le menu de connexion renouvelevées dans `Admin` -> `Connexion` -> `+` :

> `Connection id` : `minio`

> `Connection type` : `Amazon Web Services`

> `AWS Access Key ID` : `$AWS_ACCESS_KEY_ID` `AWS Secret Access Key ID` : `$AWS_SECRET_ACCESS_KEY`

> `Extra` : `{"endpoint_url": "https://$AWS_S3_ENDPOINT", "aws_session_token": "$AWS_SESSION_TOKEN"}`

3. Ajouter une étape dans votre DAG qui utilise l'operator S3 pour lister les objets de votre bucket (documentation). Observer le XComs.

```
import pendulum
from airflow.decorators import dag, task
from airflow.operators.bash import BashOperator
from airflow.models import Variable
from airflow.providers.amazon.aws.operators.s3 import S3ListOperator  # <-
--- import de l'operator S3 List
```

```
@task()
def etape1():
    return int(Variable.get("number"))

@task()
def etape2(number: int):
    print(number + 10)

@dag("test_dag",
    schedule="*/10 * * * *",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def test():
    a = etape1()
    b = etape2(number=a)
    c = BashOperator(task_id="bash_task", bash_command="date")

    s3_file = S3ListOperator(  # <---- Ajoute la task S3ListOperator
        task_id='list_s3_files',
        bucket='tgameiro', # <---- Utiliser votre bucket personnel
(visible sur l'URL : https://user-<user>-)
        aws_conn_id='minio'
            )

    a >> b >> c >> s3_file # <---- Ajout de la dépendance


test()
```

Exercice 8 : Lancer des requêtes SQL

1. Installer le provider postgreSQL `pip install apache-airflow-providers-postgres` et rédémarrer le webserver
2. Créer un service PostgreSQL dans le SSPCloud
3. Ajouter une connexion à la base de données

> Ajouter la connexion avec les informations indiqué dans les notes du service PostgreSQL. Vous pouvez ensuite tester la connexion pour vérifier qu'elle fonctionne.

3. Utiliser l'ExecuteQueryOperator (documentation) qui lance la commande `select 0`

```
import pendulum
from airflow.decorators import dag, task
from airflow.operators.bash import BashOperator
from airflow.models import Variable
from airflow.providers.amazon.aws.operators.s3 import S3ListOperator
```

```
from airflow.providers.common.sql.operators.sql import
SQLExecuteQueryOperator  # <---- import de l'operator SQL

@task()
def etape1():
    return int(Variable.get("number"))

@task()
def etape2(number: int):
    print(number + 10)

@dag("test_dag",
    schedule="*/10 * * * *",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def test():
    a = etape1()
    b = etape2(number=a)
    c = BashOperator(task_id="bash_task", bash_command="date")

    s3_file = S3ListOperator(
        task_id='list_s3_files',
        bucket='tgameiro',
        aws_conn_id='minio'
            )

    execute_query = SQLExecuteQueryOperator( # <---- Ajoute la task
SQLExecuteQueryOperator
        task_id="execute_query",
        sql="SELECT 0",
        split_statements=True,
        return_last=True,
    )

    a >> b >> c >> s3_file >> execute_query # <---- Ajout de la dépendance

test()
```

Exercice 9 : Créer des conditions dans vos DAG

Créer une `task branch` qui lance la tâche BashOperator si nous sommes le week-end, sinon lance la tâche qui liste les fichiers sur S3.

```
@task.branch
def branching():
    import datetime
    week = datetime.datetime.today().weekday()
    if week > 5:
```

```
            return "bash_task"
        return "list_s3_files"
```

Appliquer les dépendances pour les faire correspondre au schéma suivant

 href

```
import pendulum
from airflow.decorators import dag, task
from airflow.operators.bash import BashOperator
from airflow.models import Variable
from airflow.providers.amazon.aws.operators.s3 import S3ListOperator
from airflow.providers.common.sql.operators.sql import
SQLExecuteQueryOperator

@task()
def etape1():
    return int(Variable.get("number"))

@task()
def etape2(number: int):
    print(number + 10)

@task.branch
def branching():
    import datetime
    week = datetime.datetime.today().weekday()
    if week > 5:
        return "bash_task"
    return "list_s3_files"

@dag("test_dag",
    schedule="*/10 * * * *",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def test():
    a = etape1()
    b = etape2(number=a)
    c = BashOperator(task_id="bash_task", bash_command="date")
    d = branching()
    s3_file = S3ListOperator(
        task_id='list_s3_files',
        bucket='tgameiro',
        aws_conn_id='minio'
            )

    a >> b >> d >> s3_file  # <---- Ajout de la dépendance
    a >> b >> d >> c  # <---- Ajout de la dépendance

test()
```

Exercice 10 : Créer une alimentation de données

Faire un DAG qui :

- Lance une requête SQL pour créer une table avec un schéma de la table souhaité
- Liste les fichiers sur S3 dans votre bucket
- Dans une task python, lit la liste des fichiers de la première étape et selectionne le fichier
- Importer les données dans la base de données

> Créer un fichier `user.csv` en local et exporter le sur S3. `mc cp ./user.csv <user onyxia>/user.csv` avec le contenu suivant :

```
0,thierry,gameiro
1,jean,dujardin
2,eric,dupont
```

Ajouter dans la variable airflow `user` la valeur de votre utilsateur onyxia.

Appliquer le DAG suivant :

```
import pendulum
from airflow.decorators import dag, task
from airflow.operators.bash import BashOperator
from airflow.providers.amazon.aws.transfers.s3_to_sql import
S3ToSqlOperator  # <---- import de l'operator S3 Transfert
from airflow.providers.amazon.aws.operators.s3 import S3ListOperator
from airflow.providers.common.sql.operators.sql import
SQLExecuteQueryOperator

@task
def prepare_data(files): # <--- Tache python qui sélectionne le fichier
    for key in files:
        if "user" in key:
            return key


def parse_csv_to_list(filepath):
    import csv

    with open(filepath, newline="") as file:
        return list(csv.reader(file))


@dag("test_dag",
    schedule="*/10 * * * *",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
```

```
)
def test():

    create_table = SQLExecuteQueryOperator(
        task_id="create_table",
        conn_id="postgres",
        sql=["CREATE TABLE IF NOT EXISTS utilisateur (id INT PRIMARY KEY
NOT NULL, nom VARCHAR(100), prenom VARCHAR(100))", "TRUNCATE
utilisateur"], # <--- Truncate afin d'éviter si on relance la task
retourne une erreur d'ID dupliqué
        split_statements=True,
        return_last=True,
    )


    s3_file = S3ListOperator(
            task_id='list_s3_files',
            bucket="{{ var.value.get('user') }}", # <--- Récupère la
variable airflow "user" sans faire un appel en base de données en
utilisant les template jinja
            aws_conn_id='minio'
            )

    prepare = prepare_data(files=s3_file.output) # <--- pour récupérer le
XCom depuis un operator : attribut output


    transfer_s3_to_sql = S3ToSqlOperator( # <---- Transfert S3 to SQL :
https://airflow.apache.org/docs/apache-airflow-providers-
amazon/stable/_api/airflow/providers/amazon/aws/transfers/s3_to_sql/index.
html#airflow.providers.amazon.aws.transfers.s3_to_sql.S3ToSqlOperator
        task_id="transfer_s3_to_sql",
        s3_bucket="{{ var.value.get('user') }}", # <--- Récupère la
variable airflow "user" sans faire un appel en base de données en
utilisant les template jinja
        s3_key=prepare,
        sql_conn_id="postgres",
        aws_conn_id="minio",
        table="utilisateur",
        column_list=["id", "nom", "prenom"],
        parser=parse_csv_to_list,
    )

    create_table >> s3_file >> prepare  >> transfer_s3_to_sql # <----
Ajout de la dépendance


test()
```