

Objectifs du TP

- Comprendre la structure d'un DAG Airflow.
- Manipuler différents opérateurs : PythonOperator, BashOperator, DummyOperator, etc.
- Créer un décorateur personnalisé pour simplifier la définition des tâches.
- Expérimenter avec l'interface Web pour déclencher et suivre des DAGs.

Prérequis

- Python 3.8 ou supérieur.
- Airflow installé (local ou Docker).
- Connaissances de base en Python.

1. Structure générale d'un DAG

Listing 1 – Syntaxe générale d'un DAG

```
1 from airflow import DAG
2 from airflow.operators.python import PythonOperator
3 from airflow.operators.bash import BashOperator
4 from airflow.operators.dummy import DummyOperator
5 from datetime import datetime
6
7 # Exemple de fonction Python
8 def hello_world():
9     print("Hello Airflow!")
10    return "Task executed successfully"
11
12 with DAG(
13     dag_id="mon_premier_dag",
14     start_date=datetime(2025, 10, 10),
15     schedule_interval=None,
16     catchup=False,
17     tags=["formation", "exemple"]
18 ) as dag:
19
20     # Tache de debut
21     start = DummyOperator(task_id="start")
22
23     # Tache Python
24     python_task = PythonOperator(
25         task_id="hello_python",
26         python_callable=hello_world
27     )
```

```

28     # Tache Bash
29     bash_task = BashOperator(
30         task_id="hello_bash",
31         bash_command="echo 'Hello from Bash Operator!'"
32     )
33
34
35     # Tache de fin
36     end = DummyOperator(task_id="end")
37
38     # Definir les dependances
39     start >> python_task >> bash_task >> end

```

2. Exemples d'opérateurs en détail

PythonOperator avec paramètres

```

1 def process_data(filename, output_path):
2     """Fonction qui traite des données"""
3     print(f"Traitement du fichier {filename}")
4     # Simulation de traitement
5     with open(output_path, 'w') as f:
6         f.write(f"Données traitées de {filename}")
7     return f"Fichier {filename} traité avec succès"
8
9 process_task = PythonOperator(
10     task_id="process_data",
11     python_callable=process_data,
12     op_kwargs={
13         'filename': 'data.csv',
14         'output_path': '/tmp/output.txt',
15     }
16 )

```

BashOperator avec variables d'environnement

```

1 bash_complex = BashOperator(
2     task_id="complex_bash_task",
3     bash_command="""
4     echo "Execution date: {{ ds }}"
5     echo "Task instance key: {{ ti_key }}"
6     ls -la /tmp/
7     """
8     env={
9         'CUSTOM_VAR': 'valeur_personnalisee'
10    }
11 )

```

BranchPythonOperator pour la logique conditionnelle

```
1 from airflow.operators.python import BranchPythonOperator
2
3 def choose_branch(**kwargs):
4     """Decide quelle branche exécuter"""
5     execution_date = kwargs['execution_date']
6     day_of_week = execution_date.weekday()
7
8     # Exécuter une branche différente selon le jour
9     if day_of_week < 5: # Lundi Vendredi
10         return "weekday_task"
11     else: # Weekend
12         return "weekend_task"
13
14 branch_task = BranchPythonOperator(
15     task_id="choose_branch",
16     python_callable=choose_branch
17 )
18
19 weekday_task = DummyOperator(task_id="weekday_task")
20 weekend_task = DummyOperator(task_id="weekend_task")
21
22 branch_task >> [weekday_task, weekend_task]
```

3. Décorateur personnalisé avancé

Listing 2 – Décorateur pour créer une tâche Python

```
1 from airflow.operators.python import PythonOperator
2 from functools import wraps
3
4 def airflow_task(task_id=None, dag=None, **kwargs):
5     """
6         D corateur pour cr er facilement des PythonOperator
7     """
8     def decorator(func):
9         @wraps(func)
10        def wrapper(*args, **inner_kwargs):
11            return func(*args, **inner_kwargs)
12
13        # Cr er la t che Airflow
14        final_task_id = task_id or func.__name__
15        python_task = PythonOperator(
16            task_id=final_task_id,
17            python_callable=func,
18            dag=dag,
19            **kwargs
20        )
21    return python_task
22
```

```

21     # Stocker la référence de la tâche
22     wrapper.airflow_task = python_task
23     return wrapper
24
25
26     return decorator
27
28 # Utilisation du décorateur
29 @airflow_task(task_id="traitement_personnalise")
30 def ma_fonction_complexe():
31     """Une fonction avec le décorateur personnalisé"""
32     import pandas as pd
33     # Simulation de traitement de données
34     data = {'col1': [1, 2, 3], 'col2': ['a', 'b', 'c']}
35     df = pd.DataFrame(data)
36     print(f"DataFrame créé avec {len(df)} lignes")
37     return "Traitement terminé"
38
39 # Dans votre DAG, vous pouvez utiliser :
40 # ma_tache = ma_fonction_complexe.airflow_task

```

4. Exemple complet avec gestion de fichiers

```

1 from airflow import DAG
2 from airflow.operators.python import PythonOperator
3 from airflow.operators.bash import BashOperator
4 from airflow.operators.dummy import DummyOperator
5 from datetime import datetime
6 import json
7
8 def create_sample_data():
9     """Crée des données d'exemple"""
10    data = {
11        "timestamp": datetime.now().isoformat(),
12        "values": [1, 2, 3, 4, 5],
13        "metadata": {"source": "airflow_tp", "version": "1.0"}
14    }
15    with open('/tmp/sample_data.json', 'w') as f:
16        json.dump(data, f, indent=2)
17    return "Données créées"
18
19 def process_data():
20     """Traite les données créées"""
21     with open('/tmp/sample_data.json', 'r') as f:
22         data = json.load(f)
23
24         # Calculer la moyenne
25         values = data['values']
26         average = sum(values) / len(values)

```

```
27     result = {
28         "average": average,
29         "count": len(values),
30         "processed_at": datetime.now().isoformat()
31     }
32
33
34     with open('/tmp/processed_result.json', 'w') as f:
35         json.dump(result, f, indent=2)
36
37     return f"Moyenne calcul e: {average}"
38
39 def cleanup():
40     """Nettoie les fichiers temporaires"""
41     import os
42     files_to_remove = ['/tmp/sample_data.json', '/tmp/processed_result.json']
43     for file_path in files_to_remove:
44         if os.path.exists(file_path):
45             os.remove(file_path)
46             print(f"Fichier supprim : {file_path}")
47     return "Nettoyage termin"
48
49 with DAG(
50     dag_id="pipeline_complet",
51     start_date=datetime(2025, 10, 10),
52     schedule_interval=None,
53     catchup=False,
54     default_args={"retries": 1}
55 ) as dag:
56
57     start = DummyOperator(task_id="start")
58
59     create_data = PythonOperator(
60         task_id="create_data",
61         python_callable=create_sample_data
62     )
63
64     process_data_task = PythonOperator(
65         task_id="process_data",
66         python_callable=process_data
67     )
68
69     list_files = BashOperator(
70         task_id="list_files",
71         bash_command="ls -la /tmp/ | grep -E '(json|txt)' || true"
72     )
73
74     cleanup_task = PythonOperator(
75         task_id="cleanup",
76         python_callable=cleanup
```

```
77 )  
78  
79     end = DummyOperator(task_id="end")  
80  
81     # D finir le workflow  
82     start >> create_data >> process_data_task >> list_files >>  
     cleanup_task >> end
```

5. CHALLENGES - À vous de jouer !

Challenge 1 : Pipeline de traitement de données

- Créez un DAG qui :
 - Génère un fichier CSV avec 100 lignes de données aléatoires
 - Lit le fichier CSV et calcule des statistiques (moyenne, min, max)
 - Écrit les résultats dans un fichier JSON
 - Envoie un résumé par email (simulation)
 - Nettoie les fichiers temporaires

Challenge 2 : Système de décision conditionnelle

- Implémentez un DAG avec BranchPythonOperator qui :
 - Vérifie l'heure d'exécution
 - Exécute un chemin différent pour les heures de bureau (9h-18h) vs heures creuses
 - Chaque branche doit avoir au moins 2 tâches spécifiques
 - Les branches doivent se rejoindre à la fin

Challenge 3 : Décorateur avancé

- Créez un décorateur qui :
 - Log automatiquement le début et la fin de chaque tâche
 - Mesure le temps d'exécution
 - Gère les exceptions et relance les tâches si nécessaire
 - Accepte des paramètres de configuration

Challenge 4 : Workflow complexe avec parallélisme

- Conception d'un DAG avec :
 - 3 branches parallèles de traitement
 - Chaque branche doit traiter un type de données différent
 - Synchronisation des branches avec un opérateur de regroupement
 - Agrégation des résultats finaux
 - Gestion des erreurs sur une seule branche sans arrêter les autres

6. Conseils pour les challenges

- **Planifiez** votre DAG avant de coder
- **Testez** chaque tâche individuellement
- **Utilisez les logs** pour debugger
- **Pensez à l'idempotence** : vos tâches doivent pouvoir être relancées
- **Validez** vos données entre les étapes
- **Gérez les erreurs** avec élégance

Listing 3 – Structure de base pour commencer

```

1 # Template de base pour vos challenges
2 from airflow import DAG
3 from airflow.operators.python import PythonOperator,
4     BranchPythonOperator
5 from airflow.operators.bash import BashOperator
6 from airflow.operators.dummy import DummyOperator
7 from datetime import datetime
8
9 with DAG(
10     dag_id="votre_challenge",
11     start_date=datetime(2025, 10, 10),
12     schedule_interval=None,
13     catchup=False
14 ) as dag:
15
16     # Votre code ici...
17     start = DummyOperator(task_id="start")
18
19     # vous de compl ter !
20
21     end = DummyOperator(task_id="end")
22
23     # D finissez vos d pendances
24     start >> [...] >> end

```

7. Pour aller plus loin

- Utilisez les **XCom** pour échanger des données entre tâches
- Explorez les **Capteurs** (Sensors) pour attendre des événements
- Expérimitez avec les **Variables et Connections** d'Airflow
- Créez des **plugins personnalisés** pour réutiliser votre logique
- Étudiez les **bonnes pratiques** de conception de DAGs