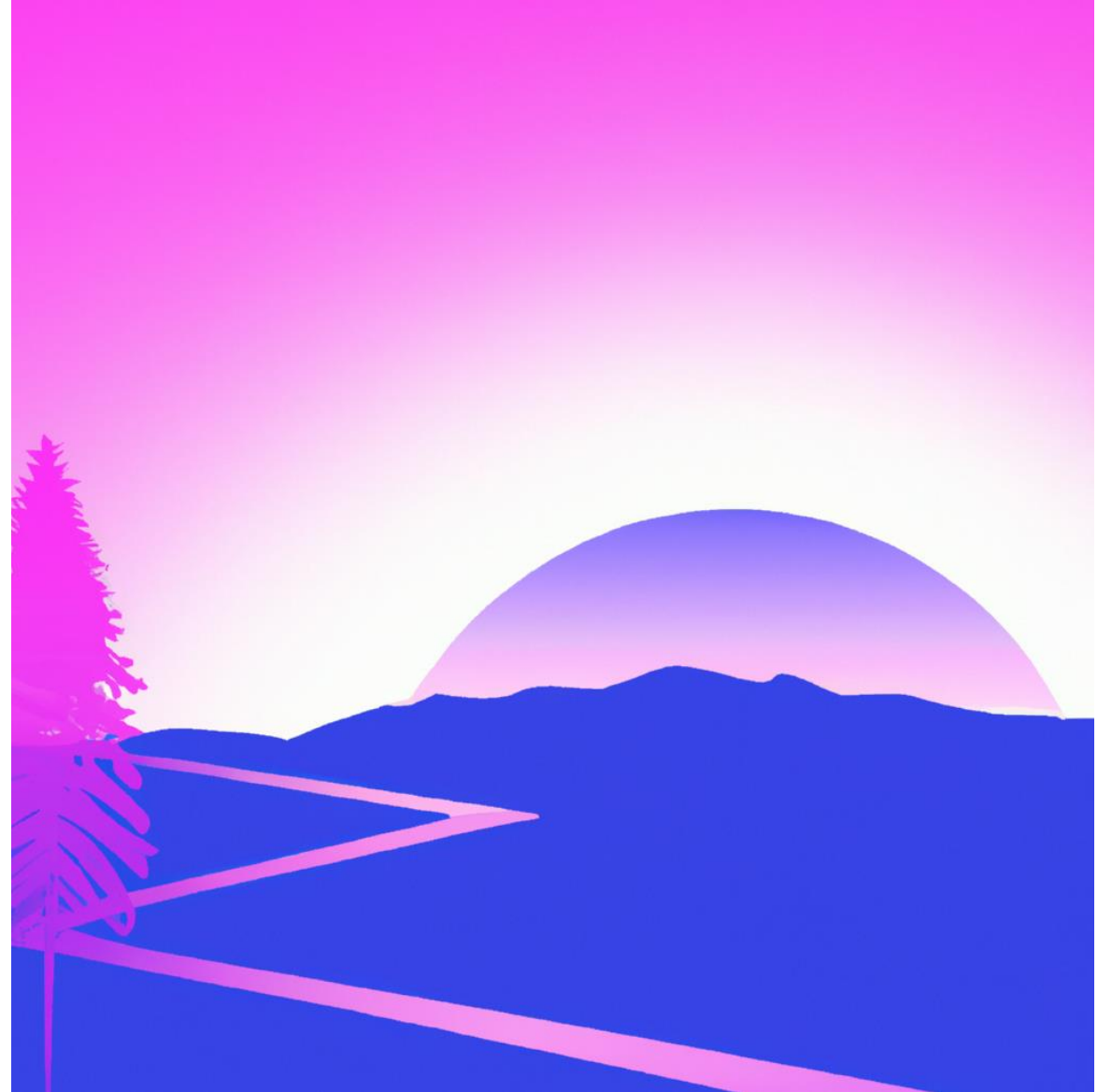


Airflow

Avancé

JUIN 2024



Objectifs de la formation

- Comprendre en profondeur les éléments clés d'Airflow : son architecture et ses composants.
- Être capable d'installer, configurer et démarrer Airflow
- Se familiariser avec les différentes vues de l'interface utilisateur d'Airflow.
- Configurer et utiliser des bases de données externes dans Airflow pour l'intégrer avec les sources de données.
- Appliquer les configurations d'ordonnancement dans Airflow, pour garantir une exécution fiable des tâches.
- Créer des workflows Airflow complexes avec des sous-workflows et gérer conditionnellement les tâches.
- Maîtriser des fonctionnalités avancées d'Airflow pour optimiser les ressources et respecter des SLAs précis.
- Intégrer des opérateurs différés et des poids de priorité pour une gestion efficace des flux de travail dans Airflow.
- Ordonner un pipeline de données réelles avec Airflow en automatisant et optimisant les tâches.
- Appliquer les bonnes pratiques d'Airflow pour optimiser la performance et la fiabilité des workflows en production.

Présentations

- Ce qui vous semble pertinent (rôle, expérience, attentes...).
- Votre éventuel rapport à Airflow dans votre travail.
- Votre background (pour orienter vers certains axes le discours et les échanges).



Programme



- 01** **Deck Airflow – Prise en main**
- 02** **Intégration de bases de données externes**
- 03** **Scheduling et planification avancée**
- 04** **Création et gestion de workflows complexes**
- 05** **Pipeline complet**
- 06** **Bonnes pratiques**

Bases Airflow

Vous disposez des slides de la formation Prise en main, car certains concepts sont réutilisés.
N'hésitez pas à vous y référer pendant les TP !



Notre repo

<https://github.com/BEESPE/airflow-avance-start>



01

Deck Airflow - Prise en main

Fondamentaux Airflow

Présentation d'Airflow et son architecture

- Apache Airflow : plateforme open-source permettant de développer, planifier et surveiller des flux de travail orientés par lots.
- Permet de construire des flux de travail interconnectant une grande variété de technologies.
- Une interface web aide à gérer l'état des flux de travail.
- Déployable de nombreuses façons, d'un simple processus à une configuration distribuée capable de prendre en charge des workflows plus complexes.



Fondamentaux Airflow

Présentation d'Airflow et son architecture

- **2014** L'équipe data de Airbnb commence à développer un outil pour répondre à leurs besoins croissants en matière d'orchestration de flux de travail. Ils avaient besoin d'une solution flexible et évolutive pour gérer leurs pipelines de données complexes, mais aucun des outils disponibles à l'époque ne répondait à leurs exigences.

Projet interne : Airflow

Objectif : créer une plateforme d'orchestration de workflows qui soit facile à utiliser, flexible et adaptée aux besoins de l'équipe data. Construction comme un projet open-source dès le début, permettant à d'autres organisations de bénéficier de leurs efforts et de contribuer à l'amélioration de la plateforme.
- **2015** Airflow ouvert au public. Rapidement, Airflow a attiré l'attention de la communauté des données et est devenu un outil populaire pour l'orchestration de workflows dans diverses entreprises et organisations.
- **2016** Airflow est remis à la Fondation Apache Software : formalisation de la gouvernance du projet, croissance continue en tant que projet open-source. Airflow a continué à évoluer avec une communauté active de contributeurs et d'utilisateurs (nouvelles fonctionnalités, corrections de bugs, améliorations de performances...)

Fondamentaux Airflow

Présentation d'Airflow et son architecture

- ➡ Aujourd'hui, Airflow est très utilisé dans de nombreuses entreprises pour l'orchestration de workflows de données, de pipelines ETL (Extract-Transform-Load)...
- ➡ La popularité d'Airflow continue de croître, et il est devenu un élément essentiel de l'écosystème des données pour de nombreuses organisations à travers le monde.

Fondamentaux Airflow

Présentation d'Airflow et son architecture

- Tous les workflows sont définis en code Python : « **workflow as code** »
- **Dynamique** Les pipelines Airflow sont configurés en tant que code Python, pour permettre une génération dynamique des pipelines.
- **Extensible** Le framework Airflow contient des opérateurs pour se connecter à différentes technologies. Les composants Airflow sont extensibles pour s'adapter facilement à l'environnement.
- **Flexible** La paramétrisation des workflows est intégrée en utilisant le moteur de templating Jinja.



Fondamentaux Airflow

Présentation d'Airflow et son architecture

```
from datetime import datetime
from airflow import DAG
from airflow.decorators import task
from airflow.operators.bash import BashOperator

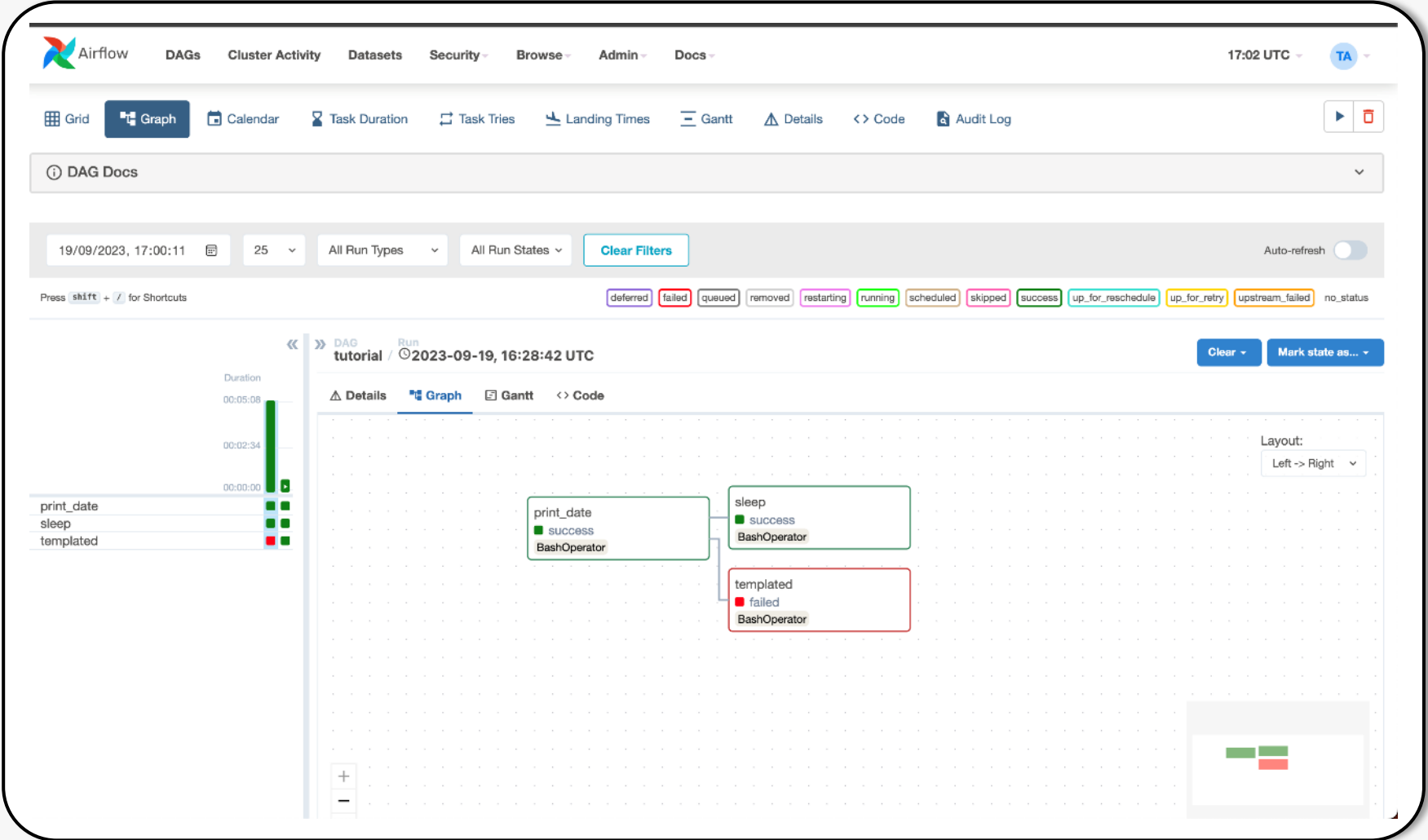
# A DAG represents a workflow, a collection of tasks
with DAG(dag_id="demo", start_date=datetime(2024, 3, 28), schedule="0 0 * * *") as dag:
    # Tasks are represented as operators
    hello = BashOperator(task_id="hello", bash_command="echo hello")

    @task()
    def airflow():
        print("airflow")

    # Set dependencies between tasks
    hello >> airflow()
```

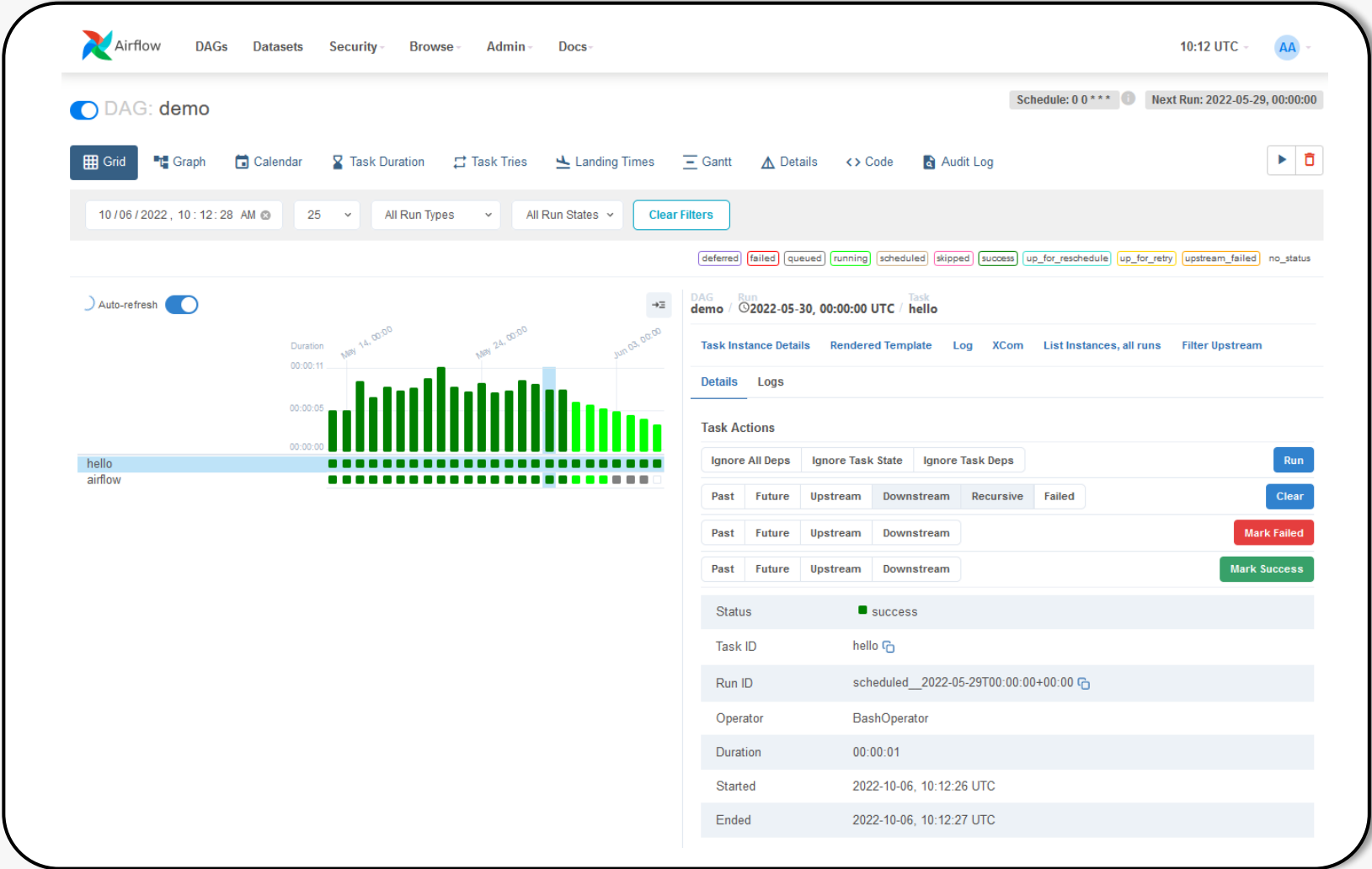
Fondamentaux Airflow

Présentation d’Airflow et son architecture



Fondamentaux Airflow

Présentation d’Airflow et son architecture



Fondamentaux Airflow

Présentation d'Airflow et son architecture

Pourquoi Airflow ?

- Si un workflow a un début et une fin clairs, et s'exécute à intervalles réguliers, il peut être programmé comme un DAG Airflow.
- Si vous préférez coder plutôt que cliquer.
- Les workflows peuvent être stockés dans un contrôle de version afin que vous puissiez revenir à des versions précédentes. En plus de l'extensibilité des composants (rappel) :
 - ➡ Les workflows peuvent être développés par plusieurs personnes simultanément.
 - ➡ Des tests peuvent être écrits pour valider la fonctionnalité.
 - ➡ Des sémantiques de planification et d'exécution riches vous permettent de définir facilement des pipelines complexes, s'exécutant à intervalles réguliers. Le backfilling vous permet de (re)exécuter des pipelines sur des données historiques après avoir apporté des modifications à votre logique. Et la possibilité de relancer des pipelines partiels après avoir résolu une erreur aide à maximiser l'efficacité.

Fondamentaux Airflow

Présentation d'Airflow et son architecture

Pourquoi Airflow ?

- Interface utilisateur d'Airflow -> vues détaillées de deux éléments :

- ➡ Pipelines

- ➡ Tâches

- = une vue d'ensemble des pipelines au fil du temps.

Depuis l'interface, on peut inspecter les journaux et gérer les tâches (par exemple, relancer une tâche en cas d'échec).

Open-source

Travail sur des composants développés, testés et utilisés par de nombreuses autres entreprises dans le monde.

Communauté active

Possibilité de trouver de nombreuses ressources utiles, sous forme de publications de blog, d'articles, de conférences, de livres... Canaux (Slack), listes de diffusion...

Airflow en tant que plateforme est vraiment personnalisable. En utilisant l'interface publique d'Airflow, on peut étendre et personnaliser presque tous les aspects d'Airflow.

Fondamentaux Airflow

Présentation d'Airflow et son architecture

Pourquoi ne pas utiliser Airflow ?

- Airflow a été conçu pour des workflows par **lots finis**.

Bien que l'interface de ligne de commande (CLI) et l'API REST permettent de déclencher des workflows, Airflow n'a pas été conçu pour des workflows événementiels s'exécutant indéfiniment. Airflow n'est pas une solution de streaming.

Cependant, un système de streaming tel que Apache Kafka peut être utilisé en collaboration avec Apache Airflow. Kafka peut être utilisé pour l'ingestion et le traitement en temps réel, les données d'événements étant écrites dans un emplacement de stockage (et Airflow démarre périodiquement un workflow pour traiter un lot de données).
- Si vous préférez cliquer plutôt que coder, Airflow n'est probablement pas la bonne solution. L'interface web vise à rendre la gestion des workflows aussi facile que possible et le framework Airflow est continuellement amélioré pour rendre l'expérience développeur aussi fluide que possible. Mais la philosophie d'Airflow est bien de définir les workflows comme du code, donc la **programmation** sera toujours **nécessaire**.

Fondamentaux Airflow

Présentation d'Airflow et son architecture

Composants requis

Une installation minimale d'Airflow comprend :

- **Un scheduler** Planificateur
Gère à la fois le déclenchement des workflows planifiés et la soumission des tâches à l'exécuteur pour les exécuter. L'**exécuteur** est une propriété de configuration du planificateur, et non un composant séparé, et s'exécute dans le processus du planificateur. Il existe plusieurs exécuteurs disponibles par défaut, et il est également possible d'en écrire un personnalisé.
- **Un serveur web** Offre une interface utilisateur pratique pour inspecter, déclencher et déboguer le comportement des DAG et des tâches.
- **Un dossier** Contenant des fichiers DAG, lu par le planificateur pour déterminer quelles tâches exécuter et quand les exécuter.
- **Une BDD** Une base de métadonnées, utilisée par les composants d'Airflow pour stocker l'état des workflows et des tâches. La configuration d'une base de données de métadonnées est décrite dans la configuration d'un Backend de Base de Données et est requise pour le bon fonctionnement d'Airflow.

Fondamentaux Airflow

Présentation d'Airflow et son architecture

Composants optionnels

Certains composants d'Airflow sont optionnels et peuvent permettre une meilleure extensibilité :

- **Worker**
Exécute les tâches qui lui sont confiées par le planificateur. Dans l'installation de base, le travailleur peut faire partie du planificateur et ne pas être un composant distinct. Il peut être exécuté en tant que processus en cours d'exécution continu dans le CeleryExecutor, ou en tant que POD dans le KubernetesExecutor.
- **Triggerer**
Exécute les tâches différées dans une boucle d'événements asyncio. Dans l'installation de base où les tâches différées ne sont pas utilisées, un déclencheur n'est pas nécessaire. Plus d'informations sur le report des tâches peuvent être trouvées dans les opérateurs et déclencheurs différables.

Fondamentaux Airflow

Présentation d'Airflow et son architecture

Composants optionnels

Certains composants d'Airflow sont optionnels et peuvent permettre une meilleure extensibilité :

- **Dag processor** Analyse les fichiers DAG et les sérialise dans la base de données de métadonnées. Par défaut, le processus du processeur de DAG fait partie du planificateur, mais il peut être exécuté en tant que composant distinct pour des raisons d'évolutivité et de sécurité. Si le processeur de DAG est présent, le planificateur n'a pas besoin de lire directement les fichiers DAG.
- **Plugins** Regroupés dans un dossier, les plugins sont un moyen d'étendre la fonctionnalité d'Airflow (similaire aux packages installés). Les plugins sont lus par le planificateur, le processeur de DAG, le déclencheur et le serveur web.

Fondamentaux Airflow

Présentation d'Airflow et son architecture

Insistons sur la distinction entre worker et executor

- Le **worker** dans Airflow est responsable de l'exécution des tâches assignées par le planificateur. Il exécute **effectivement** le code des tâches définies dans vos DAGs. Le travailleur peut être considéré comme un processus ou une instance qui traite activement les tâches. Dans un déploiement d'Airflow utilisant CeleryExecutor, par exemple, les travailleurs peuvent être des processus Celery qui exécutent des tâches en parallèle.
- L'**exécuteur** quant à lui est une **configuration** dans Airflow qui détermine comment les tâches sont exécutées. Il existe plusieurs types d'exécuteurs dans Airflow, dont les plus courants sont :
 - ➡ **SequentialExecutor** Exécute les tâches de manière séquentielle dans le même processus, principalement utilisé pour le développement et le débogage.
 - ➡ **LocalExecutor** Exécute les tâches en parallèle dans des processus distincts sur la même machine.
 - ➡ **CeleryExecutor** Utilise Celery pour exécuter les tâches en parallèle sur des travailleurs (processus Celery) distincts.
 - ➡ **KubernetesExecutor** Exécute les tâches dans des conteneurs sur un cluster Kubernetes.

Fondamentaux Airflow

Présentation d'Airflow et son architecture

Insistons sur la distinction entre worker et executor

En résumé, un travailleur est responsable de l'exécution effective des tâches, tandis que l'exécuteur détermine le mode d'exécution des tâches dans Airflow.

Question 1

Composants d'Airflow

Quel composant d'Airflow stocke les métadonnées sur l'état des workflows et des tâches ?

- ☐ Serveur Web
- ☐ Scheduler
- ☐ Worker
- ☐ Database
- ☐ Executor

Question 2

Composants d'Airflow

Quel composant d'Airflow est responsable de l'exécution effective des tâches ?

- ☐ Serveur Web
- ☐ Scheduler
- ☐ Worker
- ☐ Database
- ☐ Executor

Question 3

Composants d'Airflow

Quel composant d'Airflow est responsable de la planification et de l'exécution des tâches selon un horaire défini ?

- ☐ Serveur Web
- ☐ Scheduler
- ☐ Worker
- ☐ Database
- ☐ Executor

Question 4

Composants d'Airflow

Quel sous-composant d'Airflow détermine comment les tâches sont exécutées ?

- ☐ Serveur Web
- ☐ Scheduler
- ☐ Worker
- ☐ Database
- ☐ Executor

Fondamentaux Airflow

TP



Installation et configuration d'Airflow

Fondamentaux Airflow

Installation et configuration d'Airflow



Github

Si besoin, créer rapidement un compte Github. Le repo est disponible à l'adresse suivante.

`https://github.com/BEESPE/airflow-prise-en-main-start`

Le plus simple pour cette formation est d'importer dans votre compte Github un repo à partir du lien ci-dessus.



Fondamentaux Airflow

Installation et configuration d'Airflow



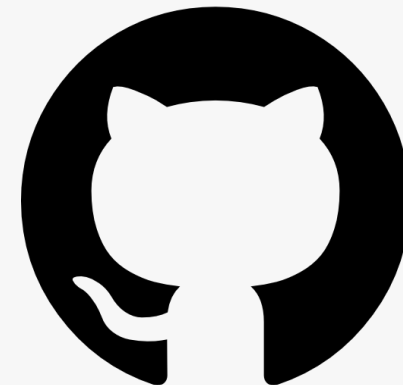
Gitpod

Pour les manipulations nous utiliserons Gitpod. Nous démarrerons notre machine Gitpod à partir de notre projet Github, en rajoutant `https://www.gitpod.io/#` devant l'URL de ce projet :

`https://www.gitpod.io/#https://github.com/<nom_utilisateur>/<projet>`

Si besoin, créer un compte Gitpod. Accepter les demandes de communication entre Gitpod et Github.

Puis, lancer le workspace. Vous pouvez utiliser la configuration par défaut (VSCode, Standard). Vous pouvez changer l'IDE si vous avez d'autres préférences.



Fondamentaux Airflow

Installation et configuration d'Airflow

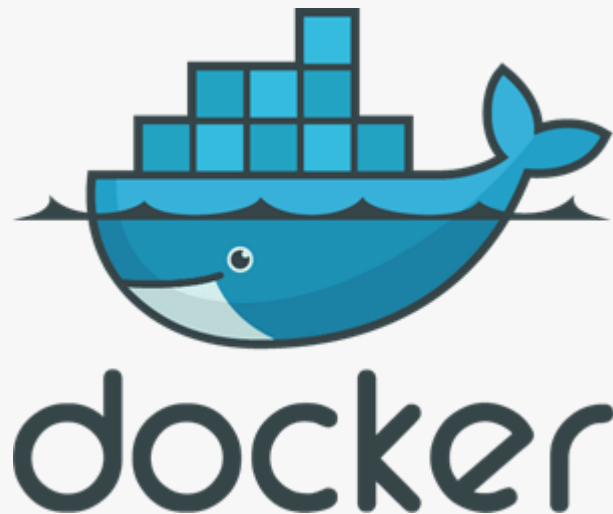


Docker

Récupérer (à la main ou en utilisant une commande `curl -Lf0`) le fichier `docker-compose.yaml` à l'adresse suivante :

<https://airflow.apache.org/docs/apache-airflow/2.8.3/docker-compose.yaml>

Bien entendu il est possible de personnaliser ce fichier, mais nous utiliserons celui-ci en expliquant les composants essentiels auxquels il fait référence.



Fondamentaux Airflow

Installation et configuration d'Airflow

Tour d'horizon des variables d'environnement/services

Postgres -

```
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: 10s
      retries: 5
      start_period: 5s
    restart: always
```

Base de données PostgreSQL utilisée comme backend pour stocker les métadonnées et la configuration d'Airflow :

- ➡ **Stockage des métadonnées des DAGs**
- ➡ **Suivi de l'historique d'exécution**
- ➡ **Gestion des connexions et des variables**
- ➡ **Planification et ordonnancement**
- ➡ **Journalisation et surveillance**

Fondamentaux Airflow

Installation et configuration d'Airflow

Tour d'horizon des variables d'environnement/services

Redis –

```
services:
  redis:
    image: redis:latest
    expose:
      - 6379
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 30s
      retries: 50
      start_period: 30s
    restart: always
```

Agent de messages (broker) pour la communication entre les différents composants d'Airflow, tels que le scheduler, les workers et le flower.

Système de stockage de données en mémoire de type clé-valeur, rapide et hautement extensible, qui est utilisé dans Airflow pour la gestion des files d'attente et la transmission des messages entre les différents processus.

Fondamentaux Airflow

Installation et configuration d'Airflow

Tour d'horizon des variables d'environnement/services

Le webservice –

```
services:
  airflow-webserver:
    <<: *airflow-common
    command: webserver
    ports:
      - "8080:8080"
    healthcheck:
      test: ["CMD", "curl", "--fail", "http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 5
      start_period: 30s
    restart: always
    depends_on:
      <<: *airflow-common-depends-on
      airflow-init:
        condition: service_completed_successfully
```

Interface utilisateur web d'Airflow.

Composant essentiel qui permet aux utilisateurs de visualiser, de gérer et d'interagir avec les workflows et les tâches Airflow via une interface conviviale accessible via un navigateur web.

Fondamentaux Airflow

Installation et configuration d'Airflow

Tour d'horizon des variables d'environnement/services

Le scheduler –

```
services:
  airflow-scheduler:
    <<: *airflow-common
    command: scheduler
    healthcheck:
      test: ["CMD", "curl", "--fail", "http://localhost:8974/health"]
      interval: 30s
      timeout: 10s
      retries: 5
      start_period: 30s
    restart: always
    depends_on:
      <<: *airflow-common-depends-on
      airflow-init:
        condition: service_completed_successfully
```

Composant essentiel responsable de la planification et de l'exécution des tâches selon un horaire défini.

Fondamentaux Airflow

Installation et configuration d'Airflow

Tour d'horizon des variables d'environnement/services

Le worker –

```
services:
  airflow-worker:
    <<: *airflow-common
    command: celery worker
    healthcheck:
      # yamllint disable rule:line-length
      test:
        - "CMD-SHELL"
        - 'celery --app airflow.providers.celery.executors.celery_executor.app inspect ping -d "celery@${HOSTNAME}"'
      interval: 30s
      timeout: 10s
      retries: 5
      start_period: 30s
    environment:
      <<: *airflow-common-env
      # Required to handle warm shutdown of the celery workers properly
      # See https://airflow.apache.org/docs/docker-stack/entrypoint.html#signal-propagation
      DUMB_INIT_SETSID: "0"
    restart: always
    depends_on:
      <<: *airflow-common-depends-on
      airflow-init:
        condition: service_completed_successfully
```

Composant optionnel qui est responsable de l'exécution réelle des tâches définies dans les workflows.

Fondamentaux Airflow

Installation et configuration d'Airflow

Tour d'horizon des variables d'environnement/services

Le triggerer –

```
services:
  airflow-triggerer:
    <<: *airflow-common
    command: triggerer
    healthcheck:
      test: ["CMD-SHELL", 'airflow jobs check --job-type TriggererJob --hostname "${HOSTNAME}"]
      interval: 30s
      timeout: 10s
      retries: 5
      start_period: 30s
    restart: always
    depends_on:
      <<: *airflow-common-depends-on
      airflow-init:
        condition: service_completed_successfully
```

Responsable de l'exécution des tâches différées dans une boucle d'événements asyncio.

Utilisé lorsque des tâches doivent être exécutées en attente d'un événement externe.

Fondamentaux Airflow

Installation et configuration d'Airflow

Tour d'horizon des variables d'environnement/services

init –

```
services:
  airflow-init:
    <<: *airflow-common
    entrypoint: /bin/bash
    # yamllint disable rule:line-length
    command:
      - -c
      - |
        if [[ -z "${AIRFLOW_UID}" ]]; then
          echo
          echo -e "\033[1;33mWARNING!!!: AIRFLOW_UID not set!\e[0m"
          echo "If you are on Linux, you SHOULD follow the instructions below to set "
          echo "AIRFLOW_UID environment variable, otherwise files will be owned by root."
          echo "For other operating systems you can get rid of the warning with manually crea
          echo "    See: https://airflow.apache.org/docs/apache-airflow/stable/howto/docker-c
          echo
        fi
        one_meg=1048576
        mem_available=$((($(getconf _PHYS_PAGES) * $(getconf PAGE_SIZE) / one_meg))
        cpus_available=$(grep -cE 'cpu[0-9]+' /proc/stat)
        disk_available=$(df / | tail -1 | awk '{print $4}')
        warning_resources="false"
        if (( mem_available < 4000 )) ; then
          echo
          echo -e "\033[1;33mWARNING!!!: Not enough memory available for Docker.\e[0m"
          echo "At least 4GB of memory required. You have $(numfmt --to iec $(mem_availabl
          echo
          warning_resources="true"
        fi
```

Responsable de l'initialisation de la base de données Airflow.

Lorsque vous démarrez ou configurez Airflow pour la première fois, vous devez exécuter le service "init" pour configurer et préparer la base de données nécessaire au fonctionnement d'Airflow.

Fondamentaux Airflow

Installation et configuration d'Airflow

Tour d'horizon des variables d'environnement/services

CLI –

```
services:

  airflow-cli:
    <<: *airflow-common
    profiles:
      - debug
    environment:
      <<: *airflow-common-env
      CONNECTION_CHECK_MAX_COUNT: "0"
      # Workaround for entrypoint issue. See: https://github.com/apache/airflow/issues/16252
    command:
      - bash
      - -c
      - airflow
```

Correspond à l'interface en ligne de commande fournie par Airflow pour interagir avec le système Airflow depuis le terminal.

Fondamentaux Airflow

Installation et configuration d'Airflow

Tour d'horizon des variables d'environnement/services

Flower –

```
services:
  # You can enable flower by adding "--profile flower" option e.g. docker-compose --profile flower up
  # or by explicitly targeted on the command line e.g. docker-compose up flower.
  # See: https://docs.docker.com/compose/profiles/
  flower:
    <<: *airflow-common
    command: celery flower
    profiles:
      - flower
    ports:
      - "5555:5555"
    healthcheck:
      test: ["CMD", "curl", "--fail", "http://localhost:5555/"]
      interval: 30s
      timeout: 10s
      retries: 5
      start_period: 30s
    restart: always
    depends_on:
      <<: *airflow-common-depends-on
      airflow-init:
        condition: service_completed_successfully
```

Outil de surveillance et de gestion des tâches Celery, lorsqu'il est utilisé comme moteur d'exécution dans Airflow (ce qui est très souvent le cas).

Fondamentaux Airflow

Installation et configuration d'Airflow

Tour d'horizon des variables d'environnement/services

Celery –

```
AIRFLOW__CORE__EXECUTOR: CeleryExecutor
```

Moteur d'exécution (optionnel) utilisé pour exécuter des tâches en parallèle et de manière distribuée.

Fondamentaux Airflow

Installation et configuration d'Airflow

Tour d'horizon des variables d'environnement/services

Volume postgres-db-volume –

```
volumes:  
  postgres-db-volume:
```

Volume Docker utilisé pour stocker les données de la base de données PostgreSQL utilisée par Airflow. Ce volume est monté sur le conteneur Docker exécutant le service de la base de données PostgreSQL afin de stocker de manière persistante les données de la base de données, telles que les métadonnées des tâches, des DAG (Directed Acyclic Graphs), des journaux, des connexions...

L'utilisation d'un volume Docker distinct pour la base de données permet de séparer les données de la base de données de l'image du conteneur Docker, ce qui facilite la sauvegarde, la restauration et la gestion des données de manière indépendante de la durée de vie du conteneur.

Fondamentaux Airflow

Installation et configuration d'Airflow



Initialisation de l'environnement

Avant de démarrer Airflow pour la première fois, vous devez préparer votre environnement, c'est-à-dire créer les fichiers, répertoires et initialiser la base de données nécessaires.

Sur Linux (que nous utilisons dans nos travaux pratiques), il faut définir votre identifiant d'utilisateur hôte, sinon, les fichiers créés dans les dossiers dags, logs et plugins seront créés avec l'utilisateur root. Puis, il faut les configurer pour le docker-compose. Nous allons donc exécuter les commandes suivantes :

```
mkdir -p ./dags ./logs ./plugins ./config  
echo -e "AIRFLOW_UID=$(id -u)" > .env
```

Fondamentaux Airflow

Installation et configuration d'Airflow



Initialisation de la base de données

Il faut désormais exécuter les migrations de base de données et créer le premier compte utilisateur. (vrai pour tous les systèmes d'exploitation). Pour ce faire, exécutez :

```
docker compose up airflow-init
```

Vous devriez voir des logs plus ou moins longs. Si tout se passe bien, ils se termineront par des lignes similaires à celles-ci :

```
airflow-init_1      | Admin user airflow created  
airflow-init_1      | 2.8.3  
start_airflow-init_1 exited with code 0
```

Fondamentaux Airflow

Installation et configuration d'Airflow

Identifiants du compte créé

Le compte créé à l'étape précédente a pour identifiant airflow et pour mot de passe airflow.

Les paramètres de sécurité par défaut d'Airflow incluent l'authentification, ce qui nécessite en particulier aux utilisateurs de se connecter pour accéder à l'interface web. Il s'agit d'une mesure de sécurité visant à empêcher l'accès non autorisé aux fonctionnalités et DAGs d'Airflow.

Lorsque vous accédez à l'interface web d'Airflow pour la première fois après avoir démarré le serveur Airflow, il vous demandera de vous connecter.

Fondamentaux Airflow

Installation et configuration d'Airflow

Nettoyer l'environnement (NE PAS EXECUTER CES COMMANDES)

Pour nettoyer l'environnement une fois l'utilisation terminée ou s'il y a un problème et qu'on souhaite repartir de zéro, le plus simple est de suivre ces deux étapes :

1. Exécuter la commande `docker compose down --volumes --remove-orphans` dans le répertoire où le fichier `docker-compose.yaml` a été téléchargé.
2. Supprimer le répertoire entier dans ce même dossier, avec la commande `rm -rf '<répertoire>'`.

Fondamentaux Airflow

Installation et configuration d'Airflow



Lancer Airflow

Il est maintenant possible de lancer l'ensemble des services. Exécuter la commande suivante :

```
docker compose up
```

Dans un deuxième terminal, vous pouvez vérifier l'état des conteneurs et vous assurer qu'aucun conteneur n'est dans un état anormal en exécutant la commande suivante :

```
docker ps
```

```
● gitpod /workspace/minimal-airflow-project (main) $ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS		NAMES		
331cda446325	apache/airflow:2.8.3	"/usr/bin/dumb-init ..."	3 minutes ago	Up 2 minutes (healthy)
0.0.0.0:8080->8080/tcp, :::8080->8080/tcp		minimal-airflow-project-airflow-webserver-1		
8ea78fee90ed	apache/airflow:2.8.3	"/usr/bin/dumb-init ..."	3 minutes ago	Up 2 minutes (healthy)
8080/tcp		minimal-airflow-project-airflow-triggerer-1		
8d423a4efdf8	apache/airflow:2.8.3	"/usr/bin/dumb-init ..."	3 minutes ago	Up 2 minutes (healthy)
8080/tcp		minimal-airflow-project-airflow-worker-1		
ac893b9e2279	apache/airflow:2.8.3	"/usr/bin/dumb-init ..."	3 minutes ago	Up 2 minutes (healthy)
8080/tcp		minimal-airflow-project-airflow-scheduler-1		
e1d979c8f310	postgres:13	"docker-entrypoint.s..."	18 minutes ago	Up 18 minutes (healthy)
5432/tcp		minimal-airflow-project-postgres-1		
595e8529c08d	redis:latest	"docker-entrypoint.s..."	18 minutes ago	Up 18 minutes (healthy)
6379/tcp		minimal-airflow-project-redis-1		

```
○ gitpod /workspace/minimal-airflow-project (main) $
```

Fondamentaux Airflow

Installation et configuration d'Airflow

Accéder à l'environnement

3 possibilités

- ➡ Lancer des commandes dans le CLI.
- ➡ Utiliser un explorateur pour parcourir l'interface web.
- ➡ Utiliser l'API REST.

Fondamentaux Airflow

Installation et configuration d'Airflow

Accéder à l'environnement

Possibilité 1 : Lancer des commandes dans le CLI

Il est possible d'interagir en ligne de commande, mais il faut le faire à l'intérieur d'un service défini de type `airflow-*`.

Par exemple, pour obtenir les informations à partir de la commande `airflow info`, il faut lancer :

```
docker compose run airflow-worker airflow info
```


Fondamentaux Airflow

Installation et configuration d'Airflow

Accéder à l'environnement

Possibilité 2 : Accéder à l'interface web

Une fois que le cluster a démarré, il est possible de se logger à l'interface web pour manipuler les DAGs.

Le webserver est disponible à l'adresse <http://localhost:8080>. Rappel : les identifiants par défaut sont airflow et airflow.

Nous nous concentrerons sur cette interface dans une partie dédiée.

Fondamentaux Airflow

Installation et configuration d'Airflow

Accéder à l'environnement

Possibilité 3 : Envoyer des requêtes à l'API REST

Il est possible d'utiliser les outils classiques pour envoyer des requêtes à l'API REST.

Par exemple (pour récupérer une liste de pools) :

```
ENDPOINT_URL=http://localhost:8080/  
curl -X GET \  
    --user "airflow:airflow" \  
    "${ENDPOINT_URL}/api/v1/pools"
```

Nous n'utiliserons pas cette API aujourd'hui.

Question 5

Installation et configuration d'Airflow

Quelles sont les différentes façons d'accéder à l'environnement Airflow pour gérer les workflows et les tâches ?

- ☐ API REST pour une intégration avec d'autres systèmes
- ☐ Connexion SSH à un serveur Airflow pour une gestion directe
- ☐ Interface en ligne de commande (CLI) via le terminal
- ☐ Interface utilisateur Web (UI) via un navigateur Web

Question 6

Celery

Qu'est-ce que Celery ?

- ☐ Un système de gestion de base de données relationnelle.
- ☐ Une bibliothèque Python.
- ☐ Un système de messagerie distribué et une file d'attente asynchrone.
- ☐ Un serveur web pour le déploiement d'applications.

Question 7

Celery

Comment Celery communique-t-il avec Apache Airflow ?

- ☐ À travers des appels HTTP REST.
- ☐ En écrivant directement dans la base de données d'Airflow.
- ☐ En utilisant des messages dans une file d'attente (ex. : RabbitMQ, Redis).
- ☐ En envoyant des signaux de système d'exploitation.

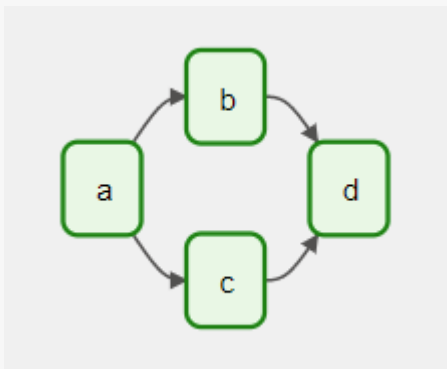
Fondamentaux Airflow

Création et exécution d'un DAG

Qu'est-ce qu'un DAG ?

Directed Acyclic Graph

Concept central d'Airflow, regroupant des tâches ensemble, organisées avec des dépendances et des relations pour définir comment elles doivent être exécutées.



Ce DAG définit quatre tâches - A, B, C et D - et dicte l'ordre dans lequel elles doivent être exécutées, ainsi que les dépendances entre elles. Il doit également spécifier la fréquence à laquelle il doit être exécuté, par exemple "toutes les 5 minutes à partir de demain" ou "tous les jours depuis le 1er janvier 2020".

Le DAG lui-même ne se soucie pas de ce qui se passe à l'intérieur des tâches ; il est simplement concerné par la manière de les exécuter - l'ordre d'exécution, le nombre de tentatives de répétition, s'il y a des délais d'expiration...

Fondamentaux Airflow

Création et exécution d'un DAG

Qu'est-ce qu'une tâche ?

Unité de base d'exécution dans Airflow.

Les tâches sont organisées en DAG (Directed Acyclic Graphs), puis des dépendances amont et aval sont définies entre elles afin d'exprimer l'ordre dans lequel elles doivent s'exécuter.

Trois types de base de tâches :

- ➡ Les opérateurs, des modèles de tâches prédéfinis que vous pouvez rapidement assembler pour construire la plupart des parties de vos DAG.
- ➡ Les capteurs, une sous-classe spéciale des opérateurs qui sont entièrement dédiés à l'attente d'un événement externe pour se produire.
- ➡ Les tâches décorées TaskFlow @task, qui sont des fonctions Python personnalisées empaquetées en tant que tâches.

En interne, ce sont toutes des sous-classes de **BaseOperator**, et les concepts de tâche et d'opérateur sont quelque peu interchangeables, mais il est utile de les considérer comme des concepts distincts - essentiellement, les opérateurs et les capteurs sont des modèles, et lorsque vous en appelez un dans un fichier DAG, vous créez une tâche.

Fondamentaux Airflow

Création et exécution d'un DAG

Relations entre les tâches :

Partie essentielle : définir comment les tâches sont liées les unes aux autres.

Dépendances dans Airflow : tâches « amont » (upstream) et « aval » (downstream). Qui précède/suit **directement**.

On déclare d'abord les tâches, puis leurs dépendances. 2 méthodes :

➡ Les opérateurs `>>` and `<<` (bitshift).

```
premiere_tache >> deuxieme_tache >> [troisième_tache, quatrieme_tache]
```

➡ Les méthodes explicites `set_upstream` et `set_downstream`.

```
first_task.set_downstream(second_task)  
third_task.set_upstream(second_task)
```


Fondamentaux Airflow

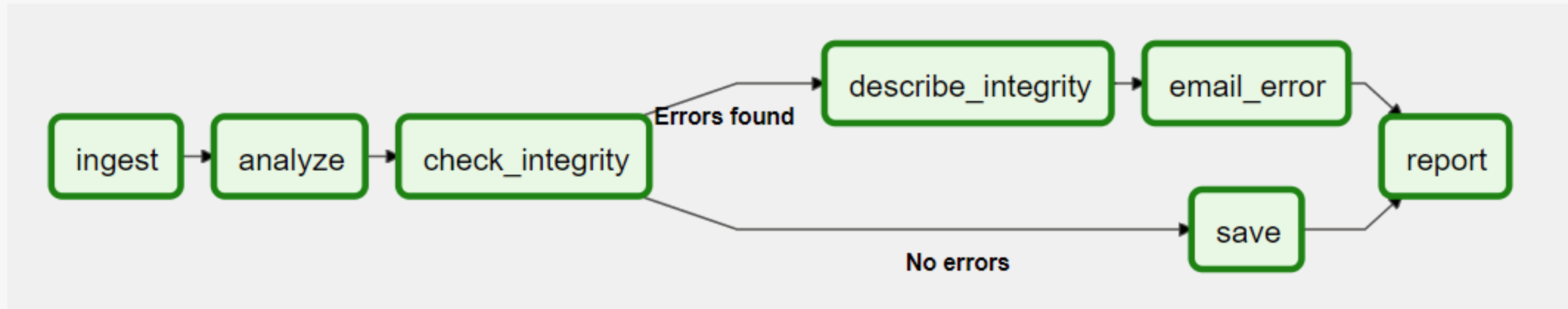
Création et exécution d'un DAG

Par défaut, une tâche s'exécute lorsque toutes ses tâches amont (parentes) ont réussi, mais il existe de nombreuses façons de modifier ce comportement pour ajouter des branchements, pour attendre uniquement certaines tâches amont, ou pour modifier le comportement en fonction de l'historique d'exécution actuel.

Nous n'en parlerons pas trop aujourd'hui, cela fait plus l'objet de la formation avancée.

Fondamentaux Airflow

Création et exécution d'un DAG



Fondamentaux Airflow

TP



Création et exécution d'un DAG

Fondamentaux Airflow

Création et exécution d'un DAG



Détail du premier DAG – import des modules

Un pipeline Airflow n'est qu'un script Python qui définit un objet DAG Airflow. Commençons par importer les bibliothèques dont nous aurons besoin :

```
import textwrap
from datetime import datetime, timedelta
from airflow.models.dag import DAG
from airflow.operators.bash import BashOperator
```

Fondamentaux Airflow

Création et exécution d'un DAG



Détail du premier DAG – arguments par défaut

Nous allons créer un DAG et quelques tâches, et nous avons le choix de passer explicitement un ensemble d'arguments au constructeur de chaque tâche (ce qui deviendrait redondant), ou (mieux !) nous pouvons définir un dictionnaire de paramètres par défaut que nous pouvons utiliser lors de la création des tâches.

Il est possible de remplacer ces arguments par défaut pour chaque tâche lors de l'initialisation de l'opérateur.

Nous pouvons définir différents ensembles d'arguments qui serviraient à différents usages (par exemple entre un environnement de production et de développement).

```
default_args={
    "depends_on_past": False,
    "email": ["airflow@example.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
    # 'wait_for_downstream': False,
    # 'sla': timedelta(hours=2),
    # 'execution_timeout': timedelta(seconds=300),
    # 'on_failure_callback': some_function, # or list of functions
    # 'on_success_callback': some_other_function, # or list of functions
    # 'on_retry_callback': another_function, # or list of functions
    # 'sla_miss_callback': yet_another_function, # or list of functions
    # 'trigger_rule': 'all_success'
},
```

Fondamentaux Airflow

Création et exécution d'un DAG



Détail du premier DAG – instantiation du DAG

Nous allons avoir besoin d'un objet DAG pour imbriquer nos tâches. Ici, nous passons une chaîne de caractères qui définit le `dag_id`, qui sert d'identifiant unique pour le DAG. Nous passons également le dictionnaire d'arguments par défaut que nous venons de définir et définissons une planification d'une journée pour le DAG.

```
with DAG(  
    "first-dag",  
    default_args=default_args,  
    description="Notre premier DAG",  
    schedule=timedelta(days=1),  
    start_date=datetime(2021, 1, 1),  
    catchup=False,  
    tags=["example"],  
) as dag:
```

Fondamentaux Airflow

Création et exécution d'un DAG

Détail du premier DAG – opérateurs

Un opérateur définit une unité de travail à accomplir pour Airflow. Utiliser des opérateurs est l'approche classique pour définir du travail dans Airflow. Pour certains cas d'utilisation, il est préférable d'utiliser l'API TaskFlow pour définir du travail dans un contexte « pythonique ».

Tous les opérateurs héritent de BaseOperator, qui inclut tous les arguments requis pour exécuter du travail dans Airflow. À partir de là, chaque opérateur inclut des arguments uniques pour le type de travail qu'il effectue. Certains des opérateurs les plus populaires sont PythonOperator, BashOperator et KubernetesPodOperator.

Ici, nous allons utiliser BashOperator pour exécuter quelques scripts bash.

Fondamentaux Airflow

Création et exécution d'un DAG



Détail du premier DAG – tâches

Pour utiliser un opérateur dans un DAG, vous devez l'instancier en tant que tâche. Les tâches déterminent comment exécuter le travail de votre opérateur dans le contexte d'un DAG.

Dans l'exemple suivant, nousinstancions le BashOperator comme deux tâches distinctes afin d'exécuter deux scripts bash différents. Le premier argument pour chaque instantiation, task_id, agit comme un identifiant unique pour la tâche.

```
t1 = BashOperator(  
    task_id="print_date",  
    bash_command="date",  
)  
  
t2 = BashOperator(  
    task_id="sleep",  
    depends_on_past=False,  
    bash_command="sleep 5",  
    retries=3,  
)
```


Fondamentaux Airflow

Création et exécution d'un DAG

Détail du premier DAG – tâches

Notons que dans la deuxième tâche, nous remplaçons le paramètre `retries` par 3.

Les règles de priorité pour une tâche sont les suivantes :

1. ➡ Arguments explicitement passés
2. ➡ Les valeurs qui existent dans le dictionnaire `default_args`
3. ➡ La valeur par défaut de l'opérateur, si elle existe

Fondamentaux Airflow

Création et exécution d'un DAG



Détail du premier DAG – templating Jinja

Airflow est capable d'exploiter la templatisation Jinja pour mettre à disposition un ensemble de paramètres et de macros intégrés. Airflow fournit également des hooks permettant à l'auteur du pipeline de définir ses propres paramètres, macros et modèles.

Dans notre DAG nous effleurons à peine la surface de ce qu'il est possible de faire avec la templatisation dans Airflow. L'objectif de cette section est simplement d'informer de l'existence de cette fonctionnalité, en montrant la variable de template la plus courante : `{{ ds }}` (le "timestamp" d'aujourd'hui). Elle est détaillée dans la formation avancée.

```
templated_command = textwrap.dedent(
    """
    {% for i in range(5) %}
        echo "{{ ds }}"
        echo "{{ macros.ds_add(ds, 7)}}"
    {% endfor %}
    """
)

t3 = BashOperator(
    task_id="templated",
    depends_on_past=False,
    bash_command=templated_command,
)
```

Fondamentaux Airflow

Création et exécution d'un DAG



Détail du premier DAG – définition des dépendances

```
t1 >> [t2, t3]
```

Fondamentaux Airflow

Création et exécution d'un DAG



Détail du premier DAG – documentation

Il est possible d'ajouter de la documentation pour les DAG ou chaque tâche individuelle.

La documentation des DAG prend en charge uniquement le markdown pour l'instant, tandis que la documentation des tâches prend en charge le texte brut, le markdown, reStructuredText, JSON et YAML.

La documentation du DAG peut être écrite comme une chaîne de documentation (docstring) au début du fichier DAG (je vous le conseille), ou n'importe où ailleurs dans le fichier.

```
"""  
  
Premier DAG avec 3 opérateurs.  
  
"""  
  
.../ ...  
  
dag.doc_md = __doc__
```

Fondamentaux Airflow

Création et exécution d'un DAG



Exécution du DAG – premières vérifications

Afin de vérifier si votre code Python ne contient pas d'erreur, nous allons lancer une commande Python à l'intérieur de notre container Docker. Pour cela, exécuter la commande suivante afin d'ouvrir le bash à l'intérieur du conteneur Docker :

```
docker exec -it <nom_du_conteneur_webserver> bash
```

Si vous avez besoin de trouver le nom de votre conteneur, exécuter la commande :

```
docker ps
```

Une fois dans le bash, exécuter la commande :

```
python dags/first_dag.py
```

Si tout va bien, rien ne se passera (en particulier aucune erreur ne sera levée).

Fondamentaux Airflow

Création et exécution d'un DAG



Exécution du DAG – vérifications supplémentaires

Initialiser les tables de BDD :

```
docker exec -it <nom_du_conteneur_webserver> airflow db migrate
```

Afficher la liste des DAGs actifs (et vérifier que celui qui nous intéresse s'y trouve) :

```
docker exec -it <nom_du_conteneur_webserver> airflow dags list
```

Afficher la liste des tâches dans notre DAG « first-dag » :

```
docker exec -it <nom_du_conteneur_webserver> airflow tasks list first-dag
```

Afficher la hiérarchie des tâches dans le DAG « first-dag » :

```
docker exec -it <nom_du_conteneur_webserver> airflow tasks list first-dag --tree
```

Fondamentaux Airflow

Création et exécution d'un DAG



Exécution – tâche par tâche

Il est possible, notamment pour déboguer, de lancer une tâche précise. La syntaxe est la suivante pour nos étapes :

```
docker exec -it <conteneur> airflow tasks test tutorial print_date 2024-03-28
```

```
docker exec -it <conteneur> airflow tasks test tutorial sleep 2024-03-28
```

```
docker exec -it <conteneur> airflow tasks test tutorial templated 2024-03-28
```

Et voilà, il est l'heure de lancer notre premier DAG !!! Vous pouvez le faire à partir de l'interface web, ou bien à partir de la commande suivante :

```
docker exec -it <conteneur> airflow dags trigger first-dag
```

Nous allons parcourir ensemble l'interface pour visualiser les différents éléments relatifs à ce premier DAG (vérification de l'exécution dans l'interface, contrôle de la bonne exécution avec les journaux d'exécution, suivi des exécutions de DAG avec DAG Runs...).

Question 8

DAGs

Que représente un DAG dans Apache Airflow ?

- ☐ Une représentation des workflows composés de tâches et de dépendances entre elles.
- ☐ Un format de fichier pour stocker des données.
- ☐ Une bibliothèque Python pour l'analyse de données.
- ☐ Une interface utilisateur pour la visualisation des tâches.

Question 9

DAGs

Quelle est la caractéristique principale d'un DAG ?

- ☐ Il peut contenir des cycles de dépendances entre les tâches.
- ☐ Il ne peut contenir que des tâches de type Python.
- ☐ Il est limité à un seul type de planification (par exemple, une exécution quotidienne).
- ☐ Il doit être dirigé et acyclique, ce qui signifie qu'il ne peut pas contenir de cycles de dépendances.

Tour d'horizon des fonctionnalités essentielles

TP



Prise en main de l'interface utilisateur

Prise en main de l'interface utilisateur

Liste les DAG dans l'environnement, ainsi qu'un ensemble de raccourcis vers des pages utiles. Il est possible de voir en un coup d'œil combien de tâches ont réussi, échoué ou sont en cours d'exécution.

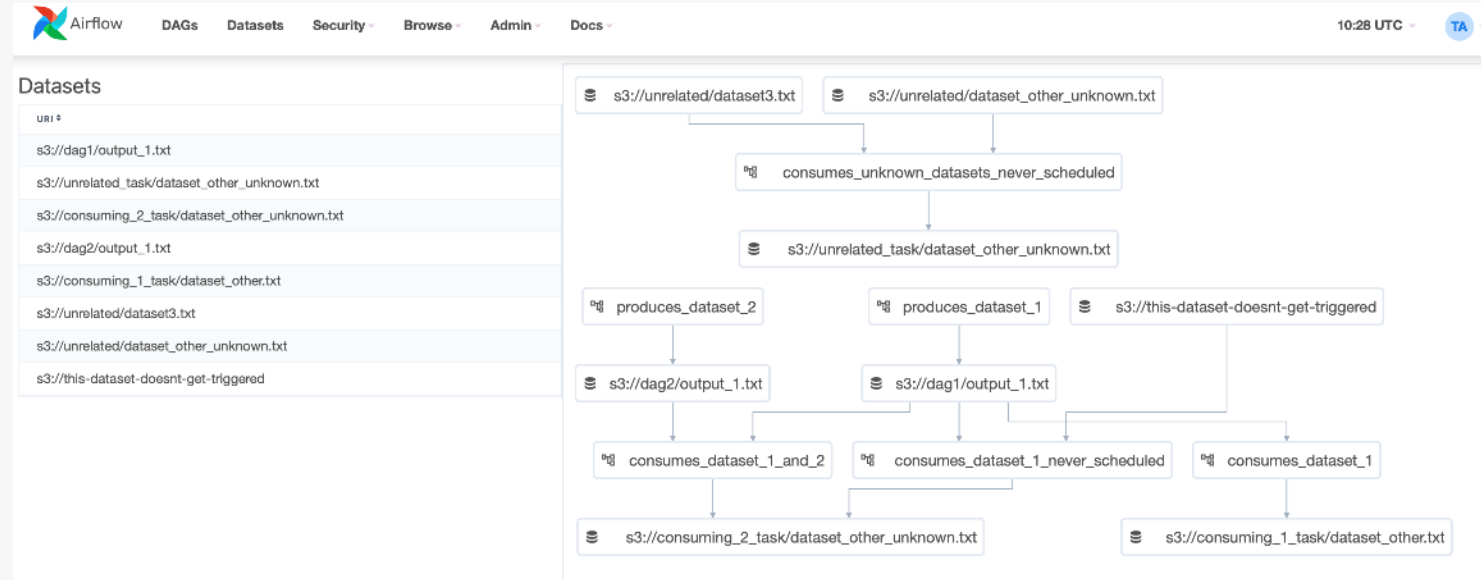
Pour pouvoir filtrer les DAGs (par exemple, par équipe), on peut utiliser des tags. Le filtre est enregistré dans un cookie et peut être réinitialisé par le bouton de réinitialisation.

```
dag = DAG("dag", tags=["équipe1", "sql"])
```

Tour d'horizon des fonctionnalités essentielles

Prise en main de l'interface utilisateur

Dataset view



Une liste combinée des ensembles de données actuels et un graphique illustrant comment ils sont produits et consommés par les DAG.

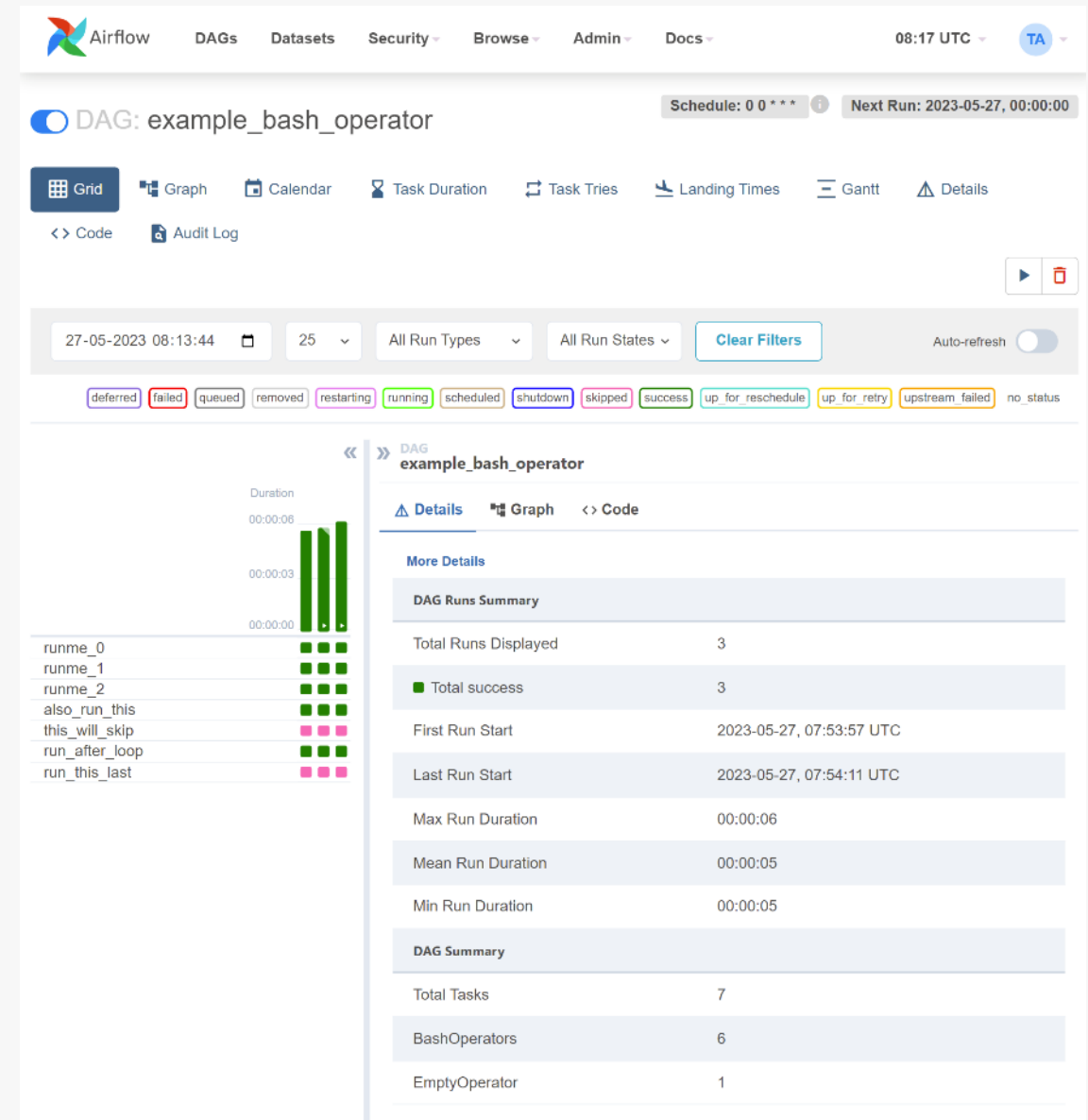
En cliquant sur n'importe quel ensemble de données dans la liste ou le graphique, celui-ci sera mis en évidence ainsi que ses relations, et la liste sera filtrée pour montrer l'historique récent des instances de tâches qui ont mis à jour cet ensemble de données et si cela a déclenché d'autres exécutions de DAG.

Tour d'horizon des fonctionnalités essentielles

Prise en main de l'interface utilisateur

Grid view

Un graphique en barres et une représentation en grille du DAG qui s'étend dans le temps. La rangée supérieure est un graphique des exécutions de DAG par durée, et en dessous, les instances de tâches. Si un pipeline est en retard, vous pouvez rapidement voir où se trouvent les différentes étapes et identifier celles qui bloquent. Entrons ensemble dans les sous-menus.



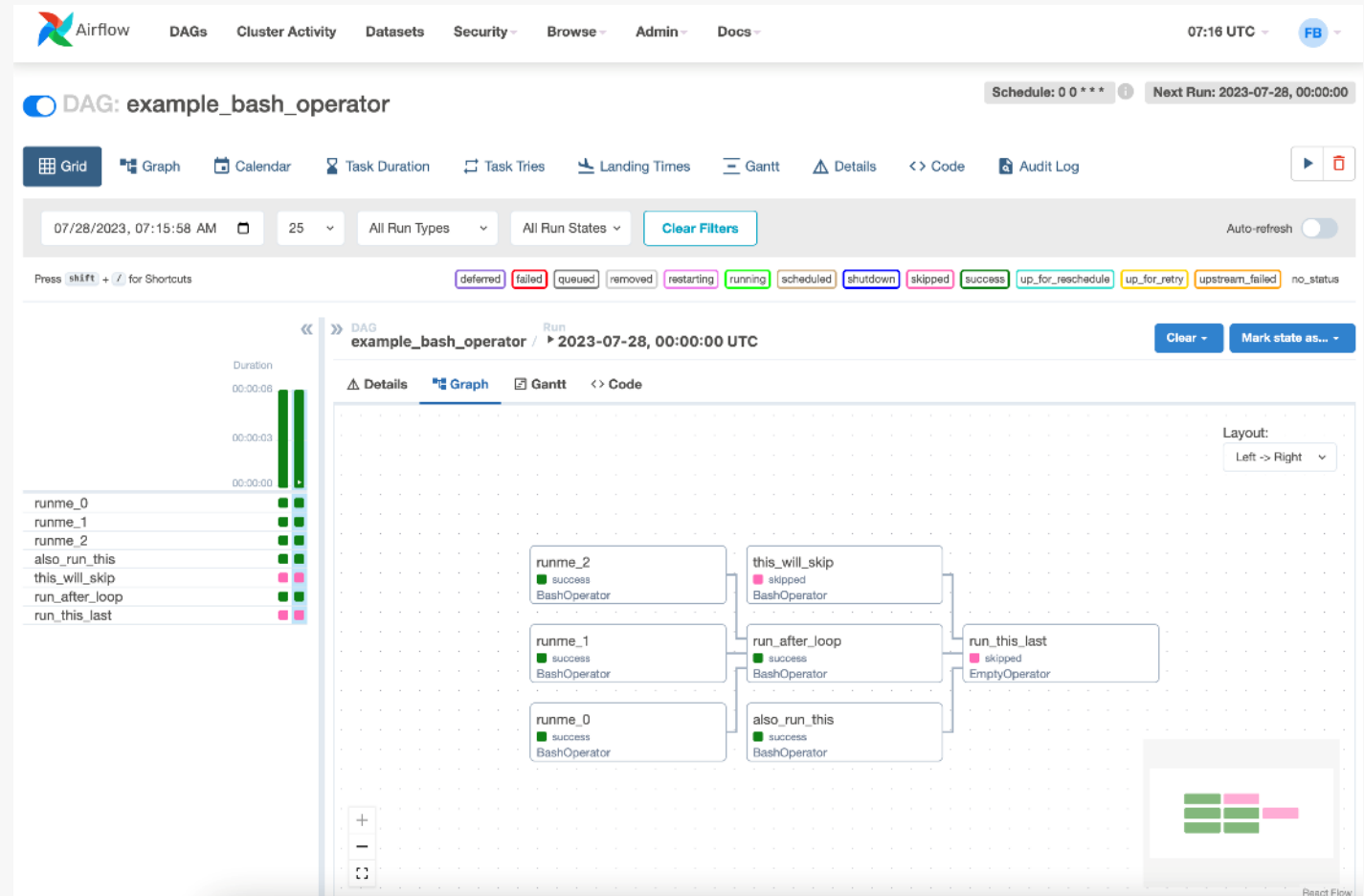
Tour d'horizon des fonctionnalités essentielles

Prise en main de l'interface utilisateur

Graph view

Peut-être la vue la plus complète.













Utile pour visualiser les dépendances du DAG et leur état actuel pour une exécution spécifique.



Tour d'horizon des fonctionnalités essentielles

Prise en main de l'interface utilisateur

Variable view

List Variable			
Search ▾			
+ Actions ▾ ↩			Record Count: 6
<input type="checkbox"/>	Key ↑	Val ↑	Is Encrypted ↑
<input type="checkbox"/>	  airtable_api_key	*****	True
<input type="checkbox"/>	  airtable_base_key	appzasdasdasdas	True
<input type="checkbox"/>	  environment	prod	True
<input type="checkbox"/>	  pipedrive_env	pipedrive	True
<input type="checkbox"/>	  postgres_env	prod	True
<input type="checkbox"/>	  snowflake_password	*****	True

Permet de répertorier, créer, éditer ou supprimer la paire clé-valeur d'une variable utilisée pendant les tâches. La valeur d'une variable sera masquée si la clé contient des mots tels que ('password', 'secret', 'passwd', 'authorization', 'api_key', 'apikey', 'access_token') par défaut, mais peut être configurée pour s'afficher en clair.

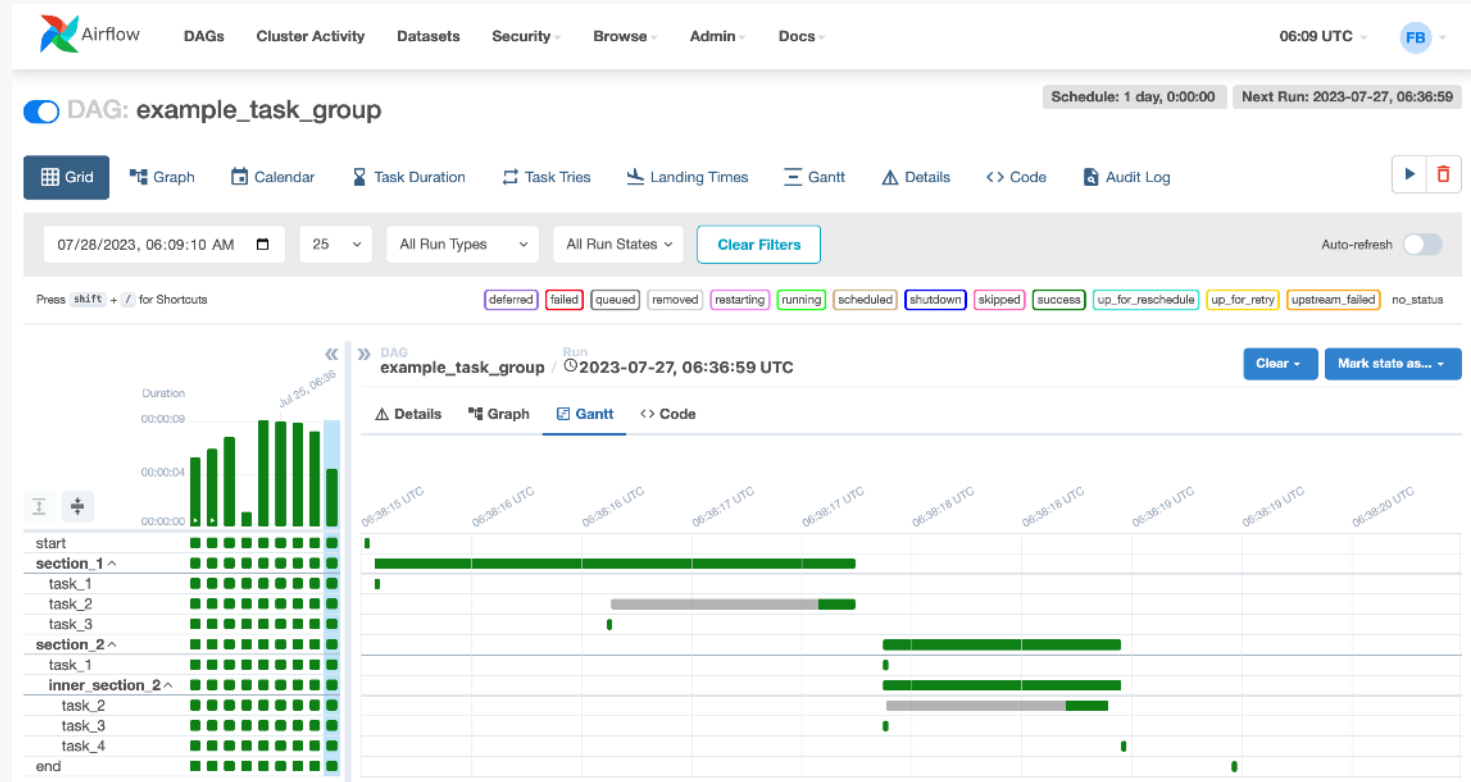
Tour d'horizon des fonctionnalités essentielles

Prise en main de l'interface utilisateur

Gantt view

Permet d'analyser la durée des tâches et les chevauchements.

On peut rapidement identifier les goulots d'étranglement et où la majeure partie du temps est passée pour des exécutions spécifiques de DAG.



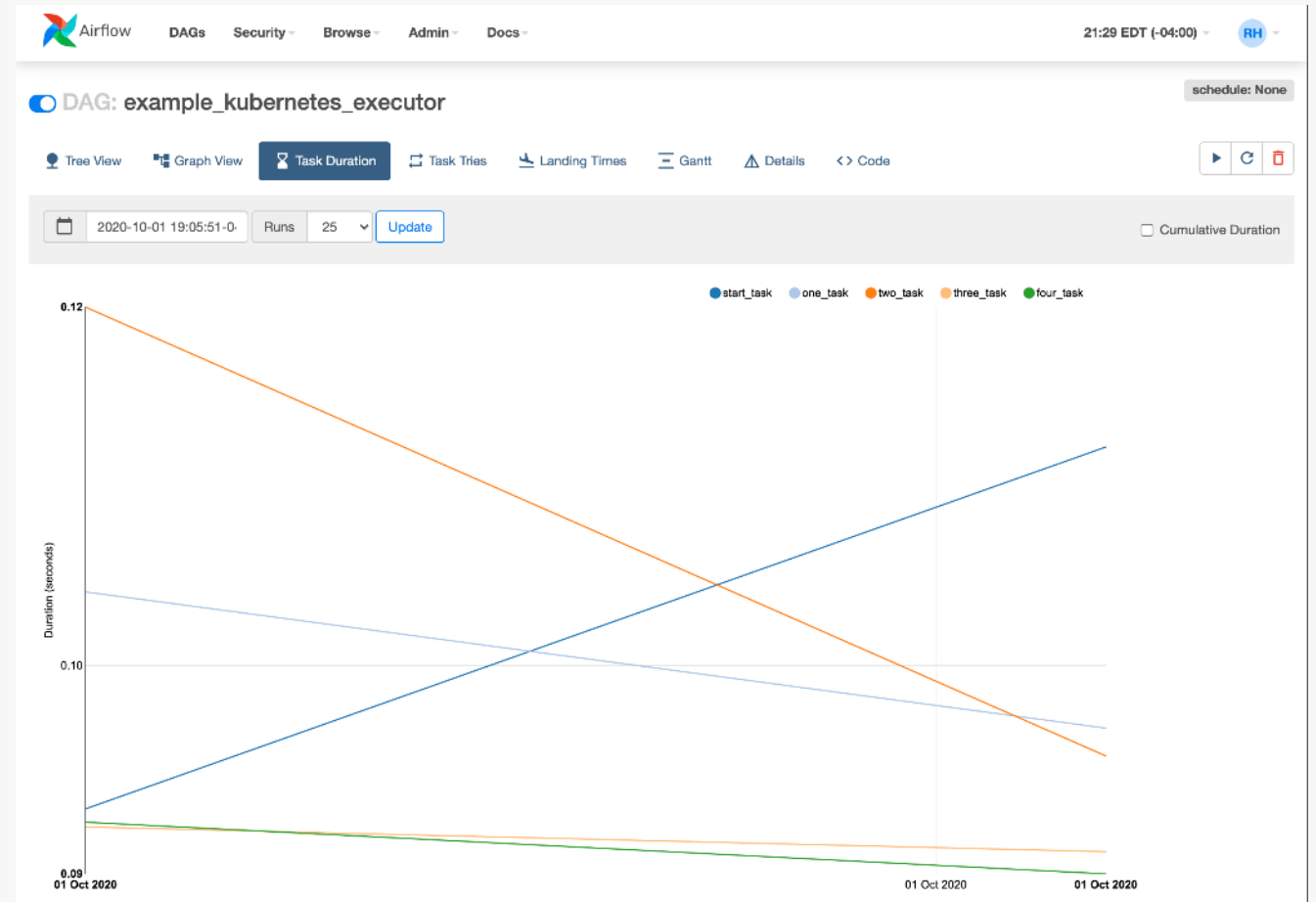
Tour d'horizon des fonctionnalités essentielles

Prise en main de l'interface utilisateur

Task duration

La durée des différentes tâches au cours des N dernières exécutions.

Cette vue permet de trouver les valeurs aberrantes et de comprendre rapidement où le temps est passé dans le DAG sur de nombreuses exécutions.



Tour d'horizon des fonctionnalités essentielles

Prise en main de l'interface utilisateur

Code view

Moyen rapide d'accéder au code qui génère le DAG et de fournir encore plus de contexte (même si on préférera classiquement y accéder à partir de nos outils de développement/contrôle de version).

The screenshot displays the Apache Airflow web interface. At the top, there's a navigation bar with links for DAGs, Cluster Activity, Datasets, Security, Browse, Admin, and Docs. The current view is for a DAG named 'example_bash_operator'. The interface includes a 'Schedule' field showing '0 0 * * *' and a 'Next Run' date of '2023-08-02, 00:00:00'. Below this, there are tabs for Grid, Graph, Calendar, Task Duration, Task Tries, Landing Times, Gantt, Details, and Code (which is currently selected). A 'Code' button is also visible. The main content area shows the DAG's code, which is a Python script defining a DAG with a single task 'run_this_last'. The code is displayed in a syntax-highlighted format. On the left side, there's a 'Task Duration' chart showing the duration of various tasks. Below the chart, there's a table listing tasks and their status: 'runme_0' (green), 'runme_1' (green), 'runme_2' (green), 'also_run_this' (green), 'this_will_skip' (pink), 'run_after_loop' (green), and 'run_this_last' (pink).

```
18 """Example DAG demonstrating the usage of the BashOperator."""
19 from __future__ import annotations
20
21 import datetime
22
23 import pendulum
24
25 from airflow import DAG
26 from airflow.operators.bash import BashOperator
27 from airflow.operators.empty import EmptyOperator
28
29 with DAG(
30     dag_id="example_bash_operator",
31     schedule="0 0 * * *",
32     start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
33     catchup=False,
34     dagrun_timeout=datetime.timedelta(minutes=60),
35     tags=["example", "example2"],
36     params={"example_key": "example_value"},
37 ) as dag:
38     run_this_last = EmptyOperator(
39         task_id="run_this_last",
40     )
```

Question 10

Vues

Que pouvez-vous faire dans la vue DAG ?

- ☐ Visualiser les dépendances entre les tâches
- ☐ Voir l'état actuel des exécutions de tâches
- ☐ Analyser la durée et le chevauchement des tâches
- ☐ Voir l'historique des exécutions de DAG

Question 11

Vues

À quoi sert la vue Gantt ?

- ☐ Analyser la durée et le chevauchement des tâches
- ☐ Voir les dépendances entre les DAG
- ☐ Visualiser les variables définies dans Airflow
- ☐ Visualiser le code des DAGs et des tâches

Tour d'horizon des fonctionnalités essentielles

Gestion des opérateurs

Base Operator

Classe abstraite de base, à partir de laquelle d'autres opérateurs sont dérivés.

Fournit les fonctionnalités de base nécessaires pour exécuter une tâche dans un DAG (Directed Acyclic Graph) d'Airflow :

- ➡ **Exécution de tâches**
- ➡ **Gestion des dépendances**
- ➡ **Gestion des tentatives**
- ➡ **Journalisation**

Tour d'horizon des fonctionnalités essentielles

Gestion des opérateurs

Opérateurs les plus utilisés

BashOperator	Exécute une commande bash.
PythonOperator	Appelle une fonction Python arbitraire.
EmailOperator	Envoie un e-mail.
SqlOperator	Exécute des scripts SQL directement depuis un DAG Airflow
TriggerDagRunOperator	Déclenche l'exécution d'un autre DAG

Nous en avons déjà mis certains en œuvre dans notre premier DAG.

Tour d'horizon des fonctionnalités essentielles

Gestion des opérateurs

Opérateurs de fichiers

Opérateurs spécifiquement conçus pour travailler avec des fichiers, tels que la lecture, l'écriture, la copie, le déplacement...

FileSensor

Surveille un fichier ou un répertoire spécifié et attend qu'il apparaisse ou disparaisse avant de poursuivre l'exécution du DAG.

FileExistsOperator

Vérifie si un fichier spécifié existe dans le système de fichiers avant de continuer l'exécution du DAG.

FileToGoogleCloudStorageOperator

Copie un fichier local vers Google Cloud Storage.

GoogleCloudStorageToBigQueryOperator

Charge des données à partir de Google Cloud Storage dans BigQuery.

HiveToPigOperator

Exécute un script Pig pour traiter des données Hive.

S3FileTransformOperator

Transforme les données stockées dans Amazon S3 en utilisant une transformation spécifiée.

Tour d'horizon des fonctionnalités essentielles

Gestion des opérateurs

Opérateurs de capteurs

Utilisés pour attendre qu'une condition spécifique soit remplie avant de poursuivre l'exécution d'un DAG.

Ces opérateurs sont utiles lorsqu'une tâche dépend de l'achèvement ou de l'occurrence d'un événement externe, tel que la création d'un fichier, la disponibilité d'une ressource, ou tout autre état prédéfini.

FileSensor

Attend qu'un fichier ou un répertoire spécifié soit créé dans le système de fichiers.

HttpSensor

Attend qu'une URL spécifiée soit accessible via HTTP.

SqlSensor

Attend qu'une requête SQL spécifiée retourne un certain résultat.

S3KeySensor

Attend qu'une clé spécifiée soit présente dans un compartiment Amazon S3.

ExternalTaskSensor

Attend qu'une tâche spécifiée dans un autre DAG soit terminée.

TimeDeltaSensor

Attend un certain laps de temps avant de continuer.

Tour d'horizon des fonctionnalités essentielles

Gestion des opérateurs

Listeners

Composants qui écoutent les événements générés par le système Airflow lors de l'exécution des tâches et des DAG.

Ils peuvent être utilisés pour déclencher des actions supplémentaires en réponse à ces événements (par exemple, pour envoyer des notifications par e-mail lorsqu'une tâche échoue ou pour enregistrer des informations sur l'exécution des tâches dans un système externe de suivi des journaux).

Souvent implémentés en tant que classes Python personnalisées qui héritent de la classe de base `BaseDagBag`.

Exemples d'événements que les auditeurs peuvent écouter dans Airflow :

<code>on_success</code>	Déclenché lorsqu'une tâche réussit.
<code>on_failure</code>	Déclenché lorsqu'une tâche échoue.
<code>on_retry</code>	Déclenché lorsqu'une tâche est réessayée après avoir échoué.
<code>on_execute</code>	Déclenché lorsque la tâche commence son exécution.
<code>on_kill</code>	Déclenché lorsqu'une tâche est tuée manuellement.

Tour d'horizon des fonctionnalités essentielles

TP



Gestion des opérateurs

Tour d'horizon des fonctionnalités essentielles

Gestion des opérateurs



A partir de l'ensemble des éléments précédents, créer un DAG dans Airflow qui utilise un BashOperator pour télécharger les données depuis une source externe et le PythonOperator pour les traiter.

La source externe sera la suivante :

<https://raw.githubusercontent.com/CourseMaterial/DataWrangling/main/flowerdataset.csv>

Le traitement sera l'ajout d'une colonne « somme » correspondant à la somme des colonnes `sepal_length` et `sepal_width`.

On pourra passer par le répertoire temporaire `/tmp` pour stocker le fichier téléchargé.

On pourra utiliser une commande `curl`.

Vous pouvez paramétrer vos arguments par défaut ou utiliser les suivants :

```
default_args = {  
    'owner': 'airflow',  
    'start_date': datetime(2024, 3, 28),  
    'retries': 4,  
    'retry_delay': timedelta(minutes=1),  
}
```

Question 12

Opérateurs

Quel opérateur est utilisé pour exécuter des commandes Bash dans Airflow ?

- ☐ BashOperator
- ☐ PythonOperator
- ☐ SQLSensor
- ☐ FileSensor

Question 13

Opérateurs

Quel opérateur est utilisé pour exécuter des commandes Python dans Airflow ?

- ☐ BashOperator
- ☐ PythonOperator
- ☐ SQLSensor
- ☐ FileSensor

Tour d'horizon des fonctionnalités essentielles

TP



DAG avec Sensor

Tour d'horizon des fonctionnalités essentielles

Intégration de bases de données externes

Pour le TP suivant, configurer des bases de données externes avec Docker sur Gitpod fait appel à des fonctionnalités un peu plus avancées. Si vous ne souhaitez plus utiliser Docker/Gitpod, vous pouvez exécuter les commandes suivantes dans un terminal local pour vous ramener au cas précédent :

```
pip install apache-airflow  
airflow standalone
```

Tour d'horizon des fonctionnalités essentielles

TP



**DAG de création, insertion et suppression
de données dans une table postgres**

Tour d'horizon des fonctionnalités essentielles

TP



**Intégration de bases de données
externes dans les DAGs**

Tour d'horizon des fonctionnalités essentielles

Intégration de bases de données externes

Dans le docker-compose, exposer le port 5432 du service postgres.

Configurer la connexion dans l'interface UI d'Airflow.

En reprenant le DAG dag-get-ext-data et sur la base de l'exemple de la démo, ajouter l'écriture de la fleur possédant la sepal_length la plus grande dans une table postgres nommée flower.

Pour information (nous y reviendrons), si process_data renvoie une valeur, il est possible d'y accéder dans notre future requête SQL avec la syntaxe Jinja :

```
{{ task_instance.xcom_pull(task_ids='process_data') }}
```

Question 14

Opérateurs BDD

Quelle est la fonction du PostgresOperator dans Airflow ?

- ☐ Créer une connexion à une base de données PostgreSQL.
- ☐ Exécuter des requêtes SQL sur une base de données PostgreSQL.
- ☐ Surveiller les changements dans une base de données PostgreSQL.
- ☐ Exporter des données depuis une base de données PostgreSQL.

Question 13

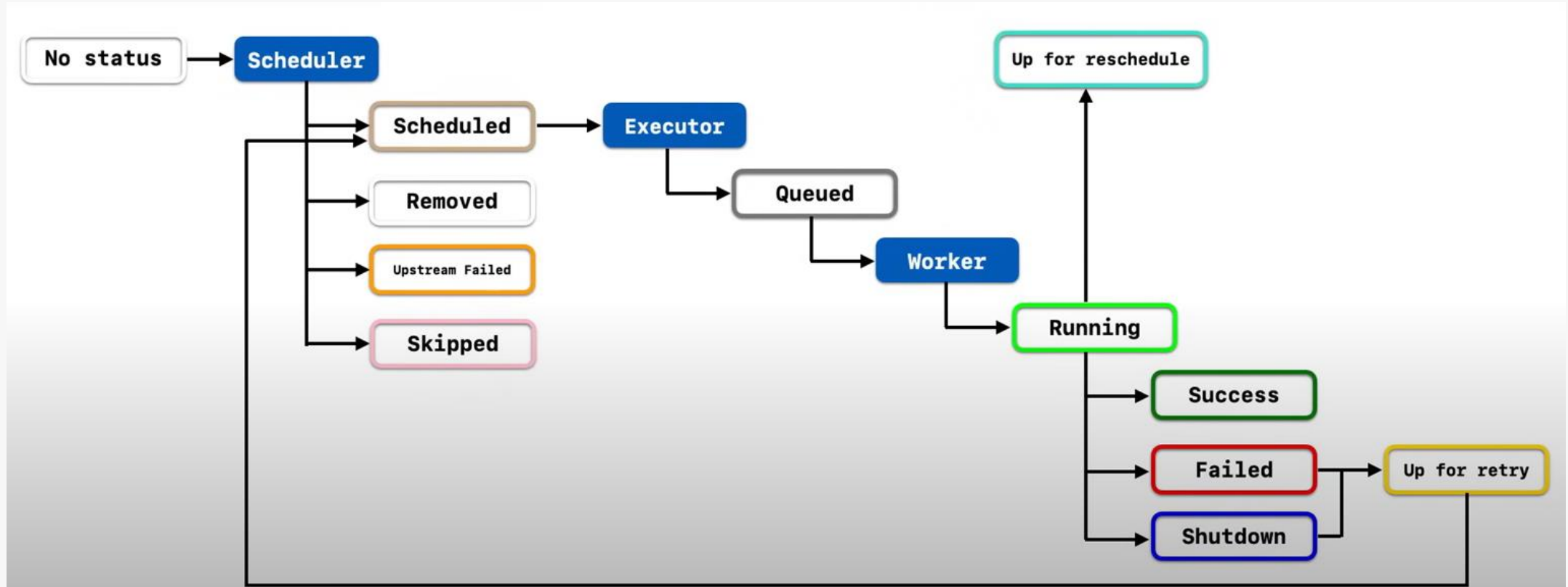
Opérateurs BDD

Quelle est la fonction du PostgresOperator dans Airflow ?

- ☐ Pour créer une connexion à une base de données PostgreSQL.
- ☐ Pour exécuter des requêtes SQL sur une base de données PostgreSQL.
- ☐ Pour surveiller les changements dans une base de données PostgreSQL.
- ☐ Pour exporter des données depuis une base de données PostgreSQL.

Scheduling et workflows

Scheduling et planification



Scheduling et workflows

Scheduling et planification

Expressions cron

Utilisées pour spécifier des horaires de planification précis dans Apache Airflow.

Suivent une syntaxe spécifique basée sur les expressions cron standard.

Permettent de définir des horaires récurrents, tels que "tous les jours à 10h" ou "toutes les heures".

* * * * * Toutes les minutes de toutes les heures de tous les jours de tous les mois.

0 * * * * A la 0ème minute de chaque heure de chaque jour de chaque mois.

0 10 * * * A 10h du matin tous les jours de chaque mois.

0 0 * * MON A minuit chaque lundi.

Dans le contexte d'Airflow, ces expressions sont utilisées dans la configuration `schedule_interval` des DAG pour déterminer quand les tâches doivent être exécutées. Par exemple, pour exécuter un DAG tous les jours à minuit, vous pouvez utiliser l'expression `0 0 * * *`.

Scheduling et workflows

Scheduling et planification

Dates de début et de fin

Start Date Date à partir de laquelle les tâches du DAG commencent à être planifiées pour l'exécution. Toute tâche planifiée avant cette date ne sera pas exécutée, même si son heure de début est antérieure à la date de début. Cela permet de contrôler le moment où le DAG commence à être actif.

End Date Date jusqu'à laquelle les tâches de votre DAG seront planifiées pour l'exécution. Après cette date, aucune nouvelle exécution de tâche ne sera programmée, même si la planification de la tâche dépasse cette date. Cela permet de limiter la durée d'exécution du DAG.

Ces deux paramètres sont souvent utilisés pour contrôler le comportement de planification d'un DAG, en définissant une fenêtre de temps pendant laquelle les tâches peuvent être exécutées. Cela permet de planifier l'exécution de tâches dans des intervalles spécifiques et de limiter la durée de vie du DAG pour des raisons de gestion des ressources ou de conformité aux contraintes de planification.

Scheduling et workflows

Scheduling et planification

Schedule Interval

Spécifie à quelle fréquence les tâches d'un DAG doivent être planifiées pour l'exécution. Cela permet de contrôler la cadence à laquelle les tâches sont exécutées, en définissant des intervalles réguliers entre chaque exécution.

@hourly Une exécution chaque heure.

@daily Une exécution chaque jour à minuit.

@weekly Une exécution chaque semaine le dimanche à minuit.

@monthly Une exécution chaque mois le premier jour du mois à minuit.

`**'/15 * * * '` : Planifie une exécution toutes les 15 minutes.

Bien entendu, on peut définir des intervalles personnalisés en utilisant la notation cron que nous venons de voir.

Utiliser les intervalles de planification pour ajuster la fréquence à laquelle les tâches sont exécutées en fonction des besoins du flux de travail.

Scheduling et workflows

Scheduling et planification

Exécutions manquées avec Backfill

Backfill = exécution rétroactive ou à la récupération des tâches manquées dans un DAG en fonction de la date de début spécifiée et de la configuration de l'option "catchup".

Lorsqu'on active le backfilling pour un DAG, Airflow examine la date de début spécifiée pour ce DAG et vérifie s'il existe des exécutions manquées entre cette date de début et la date actuelle. Si des exécutions sont manquées, Airflow les exécute rétroactivement pour remplir les lacunes dans le calendrier d'exécution.

Il est possible de gérer le backfilling avec l'option catchup ou manuellement (en ligne de commande).

Le backfilling est utile lorsqu'on met en place de nouveaux DAGs ou lorsque Airflow est hors ligne pendant un certain temps et que des exécutions planifiées ont été manquées. Cela garantit que toutes les tâches prévues sont exécutées et que les données sont traitées correctement, même après des interruptions ou des retards dans l'exécution.

Scheduling et workflows

Scheduling et planification

Rattrapage de tâches avec catchup

Catchup = exécuter des tâches pour des périodes passées lorsque le DAG est d'abord créé ou lorsqu'une planification est modifiée.

Par défaut, lorsqu'on active le catchup (rattrapage) de tâches pour un DAG (catchup=True), Airflow exécutera toutes les tâches manquées depuis la date de début du DAG jusqu'à la date actuelle. Cela permet de s'assurer que les tâches sont exécutées pour toutes les périodes passées lorsque le DAG est activé, même si celles-ci se situent avant la création du DAG.

Par exemple, si un DAG démarre le 1er janvier et qu'on l'exécute pour la première fois le 15 janvier avec le catchup activé, Airflow exécutera toutes les tâches manquées pour les jours du 1er janvier au 15 janvier.

En revanche, si le catchup est désactivé (catchup=False), Airflow n'exécutera que les tâches à partir de la date de début spécifiée dans le DAG, sans tenir compte des périodes passées.

Scheduling et workflows

Scheduling et planification

Nombre d'exécutions simultanées avec Max Active Runs

Contrôle le nombre maximal d'exécutions simultanées (ou "runs") d'un DAG. Cela signifie qu'il limite le nombre d'instances du même DAG qui peuvent être en cours d'exécution en même temps.

Par exemple, si vous définissez "Max Active Runs" sur 1 pour un DAG spécifique, cela signifie qu'Airflow n'exécutera qu'une seule instance (run) de ce DAG à la fois. Tant que cette exécution n'est pas terminée ou échouée, Airflow ne démarrera pas une autre exécution de ce même DAG, même si un déclencheur ou une planification a lieu.

Ce paramètre est utile pour contrôler la charge de travail dans le système Airflow et pour éviter une surcharge des ressources lors de l'exécution de nombreux DAGs simultanément. Il faut ajuster ce paramètre en fonction des capacités du système et des exigences du flux de travail.

Scheduling et workflows

Scheduling et planification

Service Level Agreement (SLA, Accord de Niveau de Service)

Permet de définir des contraintes de temps pour l'exécution des tâches dans un DAG. L'utilisation des SLA dans Airflow permet de surveiller les performances du workflow et de garantir que les tâches sont exécutées dans les délais spécifiés.

On peut définir un SLA pour chaque tâche individuelle dans notre DAG en spécifiant un délai d'exécution maximal. Cela signifie que la tâche doit être terminée dans ce délai imparti, suivant son démarrage.

Une fois que les SLA sont définis, Airflow surveille l'exécution des tâches en temps réel. Si une tâche ne parvient pas à se terminer dans le délai spécifié par son SLA, Airflow la marque comme ayant dépassé le délai.

Lorsqu'une tâche dépasse son délai SLA, Airflow peut déclencher des actions définies par l'utilisateur, telles que l'envoi d'alertes par e-mail ou l'exécution de tâches de récupération.

Airflow fournit des outils de surveillance et de visualisation pour suivre les violations de SLA. Vous pouvez afficher des métriques et des rapports pour évaluer les performances du workflow par rapport aux objectifs de niveau de service.

Nous ne les manipulerons pas dans cette formation prise en main, mais il suffit de régler l'argument « sla ».

Scheduling et workflows

Scheduling et planification

Gestion des fuseaux horaires avec Timezone

Dans Airflow, la gestion des fuseaux horaires est effectuée en définissant le paramètre `timezone` au niveau du DAG ou au niveau de la tâche (cela permet de spécifier le fuseau horaire dans lequel les horaires de début et de fin des tâches doivent être interprétés).

```
with DAG(
    dag_id='my_dag',
    default_args=default_args,
    start_date=datetime(2024, 3, 28),
    schedule_interval='@daily',
    timezone='Europe/Paris',
) as dag:
    pass
```

Gestion au niveau DAG

```
task1 = BashOperator(
    task_id='task1',
    bash_command='echo hello',
    timezone='America/New_York',
)
```

Gestion au niveau tâche

Scheduling et workflows

Scheduling et planification

Retry Logic (1/2)

Détermine le comportement des tâches en cas d'échec lors de leur exécution. Lorsqu'une tâche échoue, Airflow peut être configuré pour réessayer automatiquement l'exécution de la tâche après un certain délai. Cette fonctionnalité est utile pour gérer les erreurs temporaires ou les problèmes de connectivité qui peuvent survenir pendant l'exécution d'un workflow.

```
default_args = {
    'owner': 'airflow',
    'retries': 3, # Nombre de réessais
    'retry_delay': timedelta(minutes=5), # Délai entre chaque réessai
    'retry_exponential_backoff': True, # Utiliser un délai de réessai exponentiel
    'retry_exponential_backoff_delay': timedelta(seconds=10), # Délai initial pour le réessai exponentiel
    'retry_on_timeout': False, # Réessayer en cas de dépassement du délai d'attente
    'retry_on_upstream_retry': True, # Réessayer si une tâche parente a échoué
}

dag = DAG(
    dag_id='my_dag',
    default_args=default_args,
    schedule_interval='@daily',
):
    pass
```

Retry Logic au niveau DAG

Scheduling et workflows

Scheduling et planification

Retry Logic (2/2)

```
task1 = BashOperator(  
    task_id='task1',  
    bash_command='echo Hello',  
    retries=2, # Nombre de réessais pour cette tâche  
    retry_delay=timedelta(minutes=5), # Délai entre chaque réessai
```

Retry Logic au niveau tâche

Scheduling et workflows

Scheduling et planification

Délais d'exécution (timeouts)

Les délais d'exécution sont définis au niveau des tâches individuelles à l'aide de l'argument `execution_timeout` lors de la création de la tâche. On lui passe une valeur de type `timedelta`. Cela spécifie la durée maximale pendant laquelle une tâche peut s'exécuter avant d'être automatiquement arrêtée.

Scheduling et workflows

Scheduling et planification

Déclencheurs dans la planification avec les Triggers (1/2)

Triggers = fonctionnalités qui permettent de déclencher l'exécution d'un DAG en fonction d'événements externes ou de conditions spécifiques. Les déclencheurs peuvent être utilisés pour démarrer un DAG immédiatement ou en fonction de certaines conditions prédéfinies.

Schedule Trigger Un DAG peut être déclenché à des intervalles réguliers définis par l'option `schedule_interval` lors de sa création.

External Trigger Un DAG peut être déclenché par un événement externe, tel qu'une API REST appelée ou un fichier déposé dans un répertoire spécifique.

Airflow fournit l'opérateur `ExternalTaskSensor` pour surveiller l'état d'une autre tâche dans un DAG externe et déclencher l'exécution du DAG actuel lorsque la tâche cible est terminée.

Manual Trigger Un DAG peut être déclenché manuellement à partir de l'interface utilisateur Airflow ou en utilisant la ligne de commande `airflow trigger_dag`.

Pour exécuter un DAG à la demande plutôt que selon un horaire fixe.

Scheduling et workflows

Scheduling et planification

Déclencheurs dans la planification avec les Triggers (1/2)

Webhook Trigger

Un DAG peut être déclenché par une requête HTTP entrante (webhook) à une URL spécifique.

On utilise l'opérateur `HttpSensor` pour écouter les requêtes HTTP entrantes et déclencher l'exécution du DAG en fonction des données reçues.

Filesystem Trigger

Un DAG peut être déclenché par des modifications apportées à un système de fichiers local ou distant.

Airflow fournit des opérateurs tels que `S3KeySensor` pour surveiller les changements dans un compartiment Amazon S3 ou `FileSensor` pour surveiller les modifications de fichiers locaux.

Scheduling et workflow

TP



Scheduling et planification

Scheduling et workflows

Scheduling et planification



Petit point d'étape :

A partir des éléments vus dans les deux premières parties, créer un nouveau DAG qui affiche simplement un message dans le terminal (commande echo). Quel opérateur utilisera-t-on pour cela ? Le compléter de sorte à avoir un retry égal à 3, un retry-delai d'une minute. Rajouter un owner (pour le tag associé dans l'UI).

Scheduling et workflows

Scheduling et planification



Attaquons désormais cette partie.

Dans l'instanciation du DAG, préciser une date de départ (`start_time`) remontant à quelques jours dans le passé. Préciser également que l'intervalle de planification (`schedule_interval`) doit être quotidien. Pour cela, utiliser la syntaxe que nous avons détaillée. Enfin, préciser que l'argument `catchup` vaut `False` (pour le moment).

Lancer le DAG et observer le résultat dans l'UI.

Supprimer le DAG.

Modifier l'argument `catchup` à `True`, relancer le DAG et observer le résultat dans l'UI.

Supprimer à nouveau le DAG.

Scheduling et workflows

Scheduling et planification



Repasser une dernière fois le paramètre catchup à False.

Lancer le DAG.

Lancer la commande suivante pour rentrer dans le bash du scheduler :

```
docker exec -it <conteneur_scheduler> bash
```

Exécuter un backfill grâce à la commande suivante :

```
airflow dags backfill -s <date_depart aaaa-mm-jj> -e <date_fin aaaa-mm-jj> <dag_id>
```

Observer les résultats dans l'UI.

Scheduling et workflows

Scheduling et planification



Nous souhaitons désormais ne pas lancer ce script le week-end.

Créer un nouveau DAG à partir du précédent, mais utiliser une syntaxe cron pour rajouter cette contrainte, ainsi que pour préciser l'horaire de lancement à 10h du matin.

Scheduling et workflows

Scheduling et planification



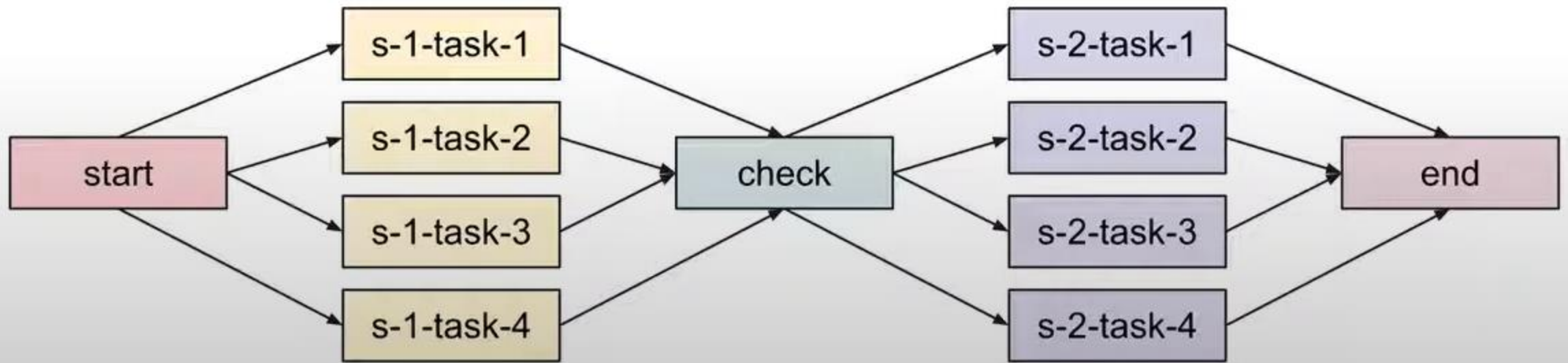
Créer un dag avec une commande artificiellement longue (par exemple à l'aide de la commande sleep) et un timeout choisi en conséquence de sorte à créer un scénario de timeout.

Lancer le DAG et visualiser le résultat dans l'UI.

Scheduling et workflows

Création de workflows

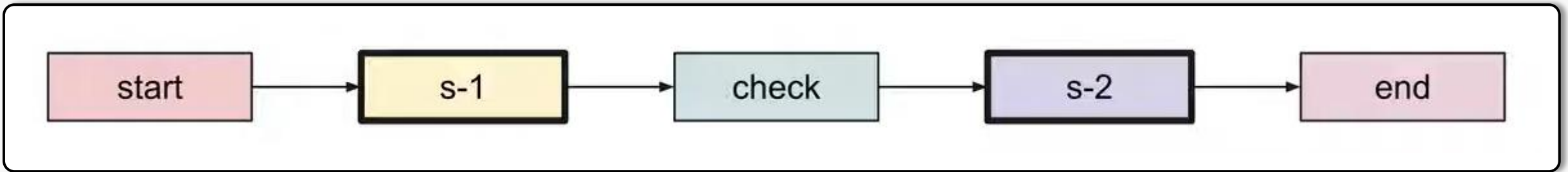
SubDAGs / Task Grouping



Scheduling et workflows

Création de workflows

SubDAGs / Task Grouping



Scheduling et workflows

Création de workflows

Dynamic Task Generation

Utile pour générer un **nombre variable de tâches en fonction de certains paramètres ou de données**.

La méthode est la suivante :

- ➡ Définir une fonction pour la génération dynamique des tâches (cette fonction peut prendre des paramètres pour personnaliser la génération des tâches en fonction de la logique métier).
- ➡ À l'intérieur de cette fonction, utiliser une boucle (par exemple, une boucle for) pour générer les tâches en fonction des données d'entrée ou des paramètres fournis.
- ➡ Pour chaque itération de la boucle, créer une nouvelle instance de tâche en utilisant les opérateurs Airflow appropriés (PythonOperator, BashOperator...).
- ➡ Après avoir généré les tâches, configurer les dépendances entre elles.
- ➡ Intégrer la fonction de génération de tâches dans le DAG, en utilisant l'opérateur PythonOperator pour appeler cette fonction et générer les tâches dynamiquement au moment de l'exécution du DAG.

Scheduling et workflows

Création de workflows

Dynamic Task Generation

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta

# Définition de la fonction pour la génération dynamique des tâches
def generate_tasks(task_id_prefix, num_tasks):
    tasks = []
    for i in range(num_tasks):
        task_id = f"{task_id_prefix}_{i}"
        bash_command = f"echo 'Executing task {task_id}'"
        task = BashOperator(
            task_id=task_id,
            bash_command=bash_command,
            dag=dag,
        )
        tasks.append(task)
    return tasks
```

Scheduling et workflows

Création de workflows

Dynamic Task Generation

```
default_args = {  
    'start_date': datetime(2024, 3, 28),  
    'retries': 1,  
    'retry_delay': timedelta(minutes=5),  
}  
  
dag = DAG(  
    'dynamic_task_generation_example',  
    default_args=default_args,  
    description='Example DAG demonstrating dynamic task generation',  
    schedule_interval=timedelta(days=1),  
)
```

Scheduling et workflows

Création de workflows

Dynamic Task Generation

```
# Appel de la fonction pour générer dynamiquement les tâches
generated_tasks = generate_tasks("dynamic_task", 5)

# Tâche finale
final_task = BashOperator(
    task_id='final_task',
    bash_command='echo "All tasks completed"',
    dag=dag,
)

# Définir les dépendances entre les tâches générées
for task in generated_tasks:
    task >> final_task
```

Scheduling et workflows

Création de workflows

Branching (1/3)

Etudions l'exemple suivant :

```
from airflow.operators.python_operator import BranchPythonOperator

# Définir la fonction pour le branchement
def decide_branch(**kwargs):
    # Récupérer le jour de la semaine (0 pour lundi, 6 pour dimanche)
    weekday = kwargs['execution_date'].weekday()
    # Si le jour est un jour de semaine (lundi à vendredi)
    if weekday < 5:
        return 'weekday_task'
    # Sinon, c'est un week-end
    else:
        return 'weekend_task'
```

Scheduling et workflows

Création de workflows

Branching (2/3)

```
# Instancier le DAG
dag = DAG(
    'dag_with_branching',
    default_args=default_args,
    schedule_interval='@daily',
)

# Créer l'opérateur de branchement
branching_task = BranchPythonOperator(
    task_id='branching',
    python_callable=decide_branch,
    provide_context=True, # Cette option permet à la fonction d'accéder aux informations de
    contexte comme la date d'exécution
    dag=dag,
)
```


Scheduling et workflows

Création de workflows

Branching (3/3)

```
# Créer les tâches correspondant aux branches
weekday_task = BashOperator(
    task_id='weekday_task',
    bash_command='echo "It\'s a weekday"',
    dag=dag,
)

weekend_task = BashOperator(
    task_id='weekend_task',
    bash_command='echo "It\'s a weekend"',
    dag=dag,
)

# Définir les dépendances entre les tâches
branching_task >> [weekday_task, weekend_task]
```

Scheduling et workflows

Création de workflows

Parameter passing

Parameter passing : utiliser un résultat de type paramètre/variable produit par une tâche dans une autre tâche.

Pour cela, on utilise la fonctionnalité XCom (Cross-Communication), pour peu que les données soient « petites » (on ne va pas transmettre des tables entières de cette manière, mais plutôt écrire quelque part les résultats).

Au niveau de la tâche amont, on peut utiliser l'objet `task_instance` (ou `ti`) pour accéder aux données de la tâche en cours d'exécution et pour pousser des données vers Xcom :

```
ti.xcom_push(key, value)
```

Dans la (fonction de la) tâche aval, on accède aux données passées en utilisant cette même `ti` :

```
ti.xcom_pull(task_ids='task_id', key='key')
```

On trouve notamment une utilité au parameter passing pour le branching. La fonction de branching peut utiliser des paramètres et décider en fonction d'eux de la branche à suivre (et renvoyer le nom de la tâche correspondante).

Scheduling et workflow

TP



Création de workflows

Scheduling et workflows

Scheduling et planification



Reprendre le DAG dag-get-ext-data. Rajouter après le traitement de données un branching en fonction d'une condition aléatoire (pour simplifier), ainsi que des tâches pour afficher le message « Belle journée pour ramasser des fleurs » dans un cas, « Malheureusement il faut travailler » dans l'autre. On pourra utiliser des PythonOperators ou des BashOperators au choix.

Scheduling et workflows

Scheduling et planification



Bien entendu, les conditions réelles ne correspondent pas au tirage aléatoire d'un nombre.

Nous allons créer une condition au branching qui dépend du résultat d'une tâche précédente.

Adapter le DAG précédent pour que les messages après le branching soient « Gros dataset » si le nombre de lignes dans le jeu de données dépasse 1000, « Petit dataset » sinon. Cela peut représenter en pratique la situation métier où les gros et petits datasets ne suivraient pas le même traitement en aval.

Fondamentaux Airflow

TP



Construction d'un pipeline sur des données

Utilisation et scénarios

Construction d'un pipeline sur des données

Mettons en œuvre les principes que nous avons vus jusque-là.

Pour ce TP, utiliser néanmoins l'API TaskFlow pour avoir une bonne représentation du panel des syntaxes possibles.

ETAPE 1

A partir de la documentation de l'API CoinGecko (<https://www.coingecko.com/api/documentation>), réaliser un DAG avec les tâches suivantes :

extract_bitcoin_price	Récupère le champ bitcoin de la réponse à une requête get envoyée à l'adresse donnant les prix des cryptomonnaies.
process_data	Renvoie un dictionnaire formé des champs usd et change du résultat précédent.
store_data	Simule l'enregistrement des données en loggant les données traitées.

Utilisation et scénarios

Construction d'un pipeline sur des données

ETAPE 2

A partir de la documentation de l'API CoinGecko (<https://www.coingecko.com/api/documentation>), adapter le DAG avec les évolutions suivantes :

extract_bitcoin_price	Récupère les données de marché historiques du bitcoin le jour de la date d'exécution de la tâche, grâce à une requête GET envoyée à un nouvel endpoint. Il faudra accéder à la date d'exécution et la formater.
process_data	Renvoie un dictionnaire formé des champs prix en usd et volume en usd en parsant le résultat précédent.
store_data	<i>Inchangée.</i>

Utilisation et scénarios

Construction d'un pipeline sur des données

ETAPE 3

Continuer à adapter le DAG :

extract_bitcoin_price	Récupère les données de marché historiques du bitcoin le jour de la date d'exécution de la tâche ainsi que les 9 jours précédents.
process_data	Calcule le RSI associé à ces points de données. Renvoie le dictionnaire précédent au jour d'exécution ainsi que le RSI.

Le RSI est donné par la formule :

$$RSI = 100 - [100 \div (1 + (\text{Gain moyen en période de hausse} \div \text{Perte moyenne en période baissière}))]$$

En pratique, nous ne pourrions pas exécuter ce traitement sans payer, à cause du rate limit de CoinGecko. Remplacer la réponse par un jeu factice.

Utilisation et scénarios

Construction d'un pipeline sur des données

ETAPE 4

Finalement :

Rajouter une tâche qui crée un dataframe à partir des dates, prix et du RSI.

Rajouter une tâche calculant les moyennes de prix de chaque semaine à partir de ce dataframe et renvoyant le tout.

Utilisation et scénarios

Bonnes pratiques Airflow

Elaborer un DAG est un processus en trois étapes :

1. ➡ Écrire du code Python pour créer un objet DAG,
2. ➡ Tester si le code répond à vos attentes,
3. ➡ Configurer les dépendances de l'environnement pour exécuter votre DAG.

Il y a des bonnes pratiques pour chacune de ces étapes.

Utilisation et scénarios

Bonnes pratiques Airflow

Minimiser les dépendances inter-DAGs (1/2)

Diviser les fonctionnalités

Divisez vos workflows en DAGs distincts en fonction de leur domaine fonctionnel. Chaque DAG devrait se concentrer sur une tâche ou un ensemble de tâches spécifiques et ne pas avoir de dépendances directes sur d'autres DAGs.

Utiliser des déclencheurs externes

Plutôt que de dépendre directement des DAGs les uns des autres, utilisez des déclencheurs externes pour déclencher l'exécution d'un DAG en fonction de l'état ou de l'exécution d'un autre DAG. Cela permet une meilleure isolation et une moindre interdépendance entre les DAGs.

Partager des données via des sources externes

Plutôt que de partager des données directement entre les DAGs, utilisez des sources de données externes telles que des bases de données ou des systèmes de stockage pour partager des informations entre les DAGs. Chaque DAG peut alors accéder aux données nécessaires sans dépendre directement des autres DAGs.

Utilisation et scénarios

Bonnes pratiques Airflow

Minimiser les dépendances inter-DAGs (2/2)

Utiliser des variables Airflow

Utilisez les variables Airflow pour partager des paramètres ou des configurations entre les DAGs si nécessaire. Cela permet de réduire les dépendances directes entre les DAGs en évitant le besoin de communiquer directement entre eux.

Eviter les dépendances cycliques

Assurez-vous qu'il n'y a pas de dépendances cycliques entre les DAGs, où un DAG dépend de lui-même ou d'un autre DAG qui dépend indirectement de lui. Cela peut entraîner des problèmes d'exécution et de planification.

Utilisation et scénarios

Bonnes pratiques Airflow

Réduire la complexité de ses DAGs (1/5)

Bien qu'Airflow soit capable de gérer de nombreux DAG avec de nombreuses tâches et dépendances entre elles, lorsque vous avez de nombreux DAG complexes, leur complexité peut avoir un impact sur les performances de l'ordonnancement. Pour maintenir votre instance Airflow performante et bien utilisée, vous devez vous efforcer de simplifier et d'optimiser vos DAGs chaque fois que possible - il faut se rappeler que le processus d'analyse et de création des DAGs n'est que l'exécution de code Python et c'est à vous de le rendre aussi performant que possible. Il n'existe pas de recettes magiques pour rendre votre DAG "moins complexe" - puisqu'il s'agit de code Python, c'est l'auteur du DAG qui contrôle la complexité de son code.

Il n'y a pas de "métriques" pour la complexité des DAGs, en particulier, il n'y a pas de métriques qui peuvent vous dire si votre DAG est "assez simple". Cependant, comme pour tout code Python, vous pouvez certainement dire que votre code de DAG est "plus simple" ou "plus rapide" lorsqu'il est optimisé.

Malgré tout, voici quelques pistes :

Utilisation et scénarios

Bonnes pratiques Airflow

Réduire la complexité de ses DAGs (2/5)

Rendez le chargement de votre DAG plus rapide.

Il s'agit d'un seul conseil d'amélioration qui peut être mis en œuvre de différentes manières, mais c'est celui qui a le plus grand impact sur les performances de l'ordonnanceur. Chaque fois que vous avez l'occasion de rendre le chargement de votre DAG plus rapide, allez-y, si votre objectif est d'améliorer les performances.

<https://airflow.apache.org/docs/apache-airflow/stable/best-practices.html>

Utilisation et scénarios

Bonnes pratiques Airflow

Réduire la complexité de ses DAGs (3/5)

Rendez la structure de votre DAG plus simple.

Chaque dépendance de tâche ajoute des frais généraux de traitement supplémentaires pour l'ordonnancement et l'exécution. Le DAG qui a une structure linéaire simple $A \rightarrow B \rightarrow C$ connaîtra moins de retards dans l'ordonnancement des tâches que le DAG qui a une structure arborescente profondément imbriquée avec un nombre de tâches dépendantes en croissance exponentielle, par exemple. Si vous pouvez rendre vos DAGs plus linéaires - où à un seul point d'exécution il y a le moins de tâches potentielles à exécuter parmi les tâches, cela améliorera probablement les performances d'ordonnancement globales.

Utilisation et scénarios

Bonnes pratiques Airflow

Réduire la complexité de ses DAGs (4/5)

Réduisez le nombre de DAGs par fichier.

Bien qu'Airflow 2 soit optimisé pour le cas où plusieurs DAGs se trouvent dans un seul fichier, il existe certaines parties du système qui le rendent parfois moins performant, ou introduisent plus de retards que d'avoir ces DAGs répartis entre plusieurs fichiers. Le simple fait qu'un fichier ne puisse être analysé que par un `FileProcessor` le rend moins évolutif par exemple. Si vous avez de nombreux DAGs générés à partir d'un fichier, envisagez de les diviser si vous constatez qu'il faut beaucoup de temps pour refléter les modifications apportées à vos fichiers DAG dans l'interface utilisateur d'Airflow.

Utilisation et scénarios

Bonnes pratiques Airflow

Réduire la complexité de ses DAGs (5/5)

Ecrire un code Python efficace.

Un équilibre doit être trouvé entre moins de DAGs par fichier, comme mentionné ci-dessus, et l'écriture de moins de code dans l'ensemble. La création des fichiers Python qui décrivent les DAGs doit suivre les meilleures pratiques de programmation et ne pas être traitée comme des configurations. Si vos DAGs partagent un code similaire, vous ne devez pas les copier encore et encore dans un grand nombre de fichiers sources presque identiques, car cela entraînera un certain nombre d'imports répétés inutiles des mêmes ressources. Au contraire, vous devriez viser à minimiser le code répété à travers tous vos DAGs afin que l'application puisse s'exécuter efficacement et puisse être facilement déboguée.

Utilisation et scénarios

Bonnes pratiques Airflow

Utiliser le templating (1/2)

Utilisation de Jinja Templating

Airflow utilise Jinja Templating pour le rendu des templates. Vous pouvez utiliser des balises Jinja (`{{ }}`) pour insérer des variables, des expressions, des filtres et des boucles dans vos tâches, dans les paramètres des opérateurs et dans d'autres parties de votre code Airflow.

Variables Airflow

Les variables Airflow peuvent être utilisées comme des templates pour stocker et récupérer des valeurs dynamiques telles que des chemins de fichiers, des configurations ou des identifiants de connexion. Vous pouvez accéder aux variables dans votre code Airflow en utilisant `{{var.variable_name }}`.

Paramètres dynamiques des opérateurs

Vous pouvez utiliser des templates pour définir des paramètres dynamiques dans les opérateurs Airflow, tels que les chemins de fichiers, les requêtes SQL, les adresses e-mail, etc. Ces paramètres peuvent être rendus dynamiquement en fonction de variables ou d'autres valeurs dans votre environnement Airflow.

Utilisation et scénarios

Bonnes pratiques Airflow

Utiliser le templating (2/2)

Utilisation de Macros

Airflow fournit un ensemble de macros intégrées qui peuvent être utilisées dans les templates Jinja pour obtenir des informations telles que la date actuelle, les dates de début et de fin d'exécution, les décalages temporels, etc. Vous pouvez utiliser ces macros dans vos tâches et paramètres d'opérateur pour une planification dynamique et une exécution conditionnelle.

Personnalisation avancée

En combinant Jinja Templating avec des variables Airflow, des macros et des expressions conditionnelles, vous pouvez créer des workflows très personnalisés et dynamiques. Cela permet une configuration flexible des tâches, des dépendances conditionnelles, des paramètres d'exécution et d'autres aspects de votre pipeline de données.

Utilisation et scénarios

Bonnes pratiques Airflow

Créer une tâche comme il faut (1/2)

Une bonne analogie pour les tâches dans Airflow : les transactions dans une base de données.

Cela implique que vous ne devez jamais produire de résultats incomplets à partir de vos tâches. Un exemple est de ne pas produire de données incomplètes dans HDFS ou S3 à la fin d'une tâche.

Il arrive (souvent) qu'Airflow réessaye une tâche en cas d'échec. Ainsi, les tâches doivent produire le même résultat à chaque nouvelle exécution (idempotence). En particulier :

- ➡ N'utilisez pas INSERT lors d'une nouvelle exécution de tâche, une instruction INSERT pourrait entraîner des lignes en double dans votre base de données. **Remplacez-la par UPSERT.**
- ➡ Lisez et écrivez dans une partition spécifique. Ne lisez jamais les données les plus récentes disponibles dans une tâche. Quelqu'un peut mettre à jour les données d'entrée entre les nouvelles exécutions, ce qui entraîne des sorties différentes. Une meilleure façon est de lire les données d'entrée à partir d'une partition spécifique. Vous pouvez utiliser `data_interval_start` comme partition. Vous devez suivre cette méthode de partitionnement également lors de l'écriture de données dans S3/HDFS.
- ➡ La fonction Python `datetime.now()` renvoie l'objet `datetime` actuel. Cette fonction ne doit jamais être utilisée à l'intérieur d'une tâche, en particulier pour effectuer le calcul critique, car elle conduit à des résultats différents à chaque exécution. (Il est acceptable de l'utiliser, par exemple, pour générer un journal temporaire.)

Utilisation et scénarios

Bonnes pratiques Airflow

Créer une tâche comme il faut (2/2)

Vous devez définir des paramètres répétitifs tels que `connection_id` ou les chemins S3 dans `default_args` plutôt que de les déclarer pour chaque tâche. Les `default_args` aident à éviter les erreurs telles que les fautes de frappe. De plus, la plupart des types de connexion ont des noms de paramètres uniques dans les tâches, vous pouvez donc déclarer une connexion une seule fois dans `default_args` (par exemple `gcp_conn_id`) et elle est automatiquement utilisée par tous les opérateurs qui utilisent ce type de connexion.

Utilisation et scénarios

Bonnes pratiques Airflow

Supprimer une tâche comme il faut

De manière générale, **soyez prudent lorsque vous supprimez une tâche d'un DAG**. Vous ne pourrez pas voir la tâche dans Graph View, Grid View... ce qui rendra difficile la vérification des journaux de cette tâche à partir du serveur Web. Si ce n'est pas souhaité, veuillez créer un nouveau DAG.

Il ne faut pas oublier la possibilité de vouloir réaliser des **analyses *a posteriori***.

Utilisation et scénarios

Bonnes pratiques Airflow

Communication (un peu hors programme de cette formation prise en main, pour référence)

Airflow exécute les tâches d'un DAG sur différents serveurs si vous utilisez l'executor Kubernetes ou l'executor Celery. Par conséquent, vous ne devez pas stocker de fichier ou de configuration dans le système de fichiers local car la tâche suivante est susceptible de s'exécuter sur un serveur différent sans y avoir accès, par exemple une tâche qui télécharge le fichier de données que la tâche suivante traite. Dans le cas de l'executor Local, le stockage d'un fichier sur le disque peut rendre les réessais plus difficiles, par exemple si votre tâche nécessite un fichier de configuration supprimé par une autre tâche dans le DAG.

Si possible, utilisez XCom pour communiquer de petits messages entre les tâches et une bonne façon de passer des données plus importantes entre les tâches est d'utiliser un stockage distant tel que S3/HDFS. Par exemple, si nous avons une tâche qui stocke des données traitées dans S3, cette tâche peut pousser le chemin S3 pour les données de sortie dans XCom, et les tâches en aval peuvent récupérer le chemin depuis XCom et l'utiliser pour lire les données.

Les tâches ne doivent pas non plus stocker de paramètres d'authentification tels que des mots de passe ou des jetons à l'intérieur d'elles. Dans la mesure du possible, utilisez des Connexions pour stocker les données de manière sécurisée dans la base de données d'Airflow et récupérez-les en utilisant un identifiant de connexion unique.

Utilisation et scénarios

Bonnes pratiques Airflow

Niveau du code Python

Vous devez **éviter d'écrire du code au niveau supérieur** qui n'est pas nécessaire pour créer des opérateurs et établir des relations DAG entre eux. Cela est dû à la décision de conception pour le planificateur d'Airflow et à l'impact de la vitesse de traitement du code au niveau supérieur sur les performances et la scalabilité d'Airflow.

Le planificateur d'Airflow exécute le code en dehors des méthodes execute des opérateurs avec un intervalle minimal de `min_file_process_interval` secondes. Cela est fait pour permettre la planification dynamique des DAG - où la planification et les dépendances peuvent changer avec le temps et affecter la prochaine planification du DAG. Le planificateur d'Airflow essaie de s'assurer en permanence que ce que vous avez dans les DAG est correctement reflété dans les tâches planifiées.

Plus précisément, vous ne devez pas exécuter d'accès à la base de données, de calculs lourds et d'opérations réseau.

Un des facteurs importants impactant le temps de chargement des DAG, qui peut être négligé par les développeurs Python, est que les imports au niveau supérieur peuvent prendre beaucoup de temps et générer beaucoup d'overhead, ce qui peut être facilement évité en les convertissant en imports locaux à l'intérieur des fonctions Python, par exemple.

Utilisation et scénarios

Bonnes pratiques Airflow

Variables Airflow (1/2)

L'utilisation des variables Airflow entraîne des appels réseau et un accès à la base de données, donc d'après ce que nous venons de dire, leur utilisation dans le code Python au niveau supérieur pour les DAG devrait être évitée autant que possible. Si les variables Airflow doivent être utilisées dans le code DAG au niveau supérieur, leur impact sur l'analyse du DAG peut être atténué en activant le cache expérimental, configuré avec un ttl raisonnable.

Il est possible d'utiliser librement les variables Airflow à l'intérieur des méthodes `execute()` des opérateurs, mais vous pouvez également passer les variables Airflow aux opérateurs existants via un modèle Jinja, ce qui retardera la lecture de la valeur jusqu'à l'exécution de la tâche.

Syntaxe :

```
{{ var.value.<variable_name> }}
```

Ou, si besoin de désérialiser un objet json depuis la variable :

```
{{ var.json.<variable_name> }}
```

Utilisation et scénarios

Bonnes pratiques Airflow

Variables Airflow (2/2)

Dans le code au niveau supérieur, les variables utilisant des modèles Jinja ne génèrent pas de requête tant qu'une tâche n'est pas en cours d'exécution, tandis que `Variable.get()` génère une requête à chaque fois que le fichier de DAG est analysé par le planificateur si le cache n'est pas activé. Utiliser `Variable.get()` sans activer le cache entraînera des performances sous-optimales dans le traitement du fichier de DAG. Dans certains cas, cela peut entraîner une expiration du délai du fichier de DAG avant qu'il ne soit entièrement analysé.

Utilisation et scénarios

Bonnes pratiques Airflow

Timetables

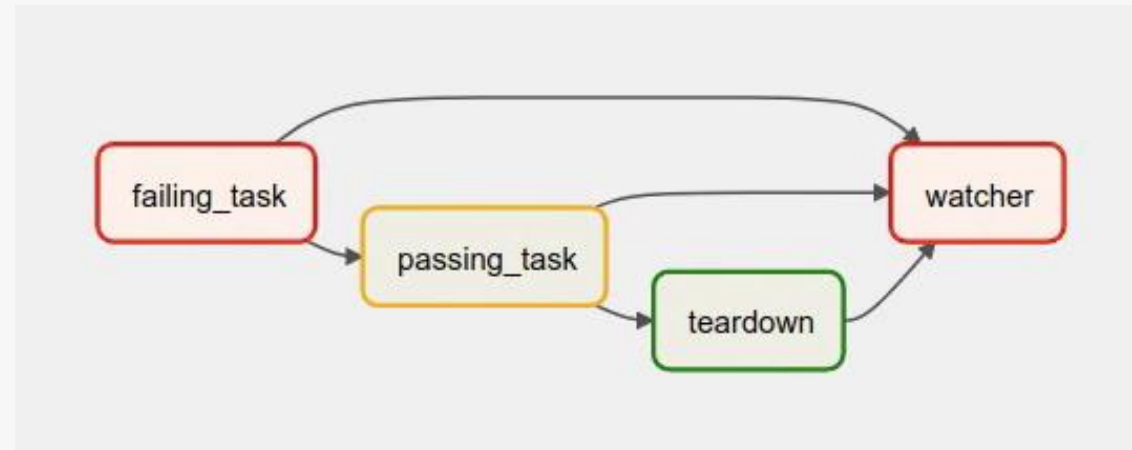
Toujours dans la même idée, évitez d'utiliser les Variables/Connexions Airflow ou d'accéder à la base de données Airflow au niveau supérieur de votre code d'horaire. L'accès à la base de données doit être retardé jusqu'au moment de l'exécution du DAG. Cela signifie que vous ne devez pas avoir de récupération de variables/connexions en argument à l'initialisation de votre classe d'horaire ou avoir des Variables/Connexions au niveau supérieur de votre module d'horaire personnalisé.

Utilisation et scénarios

Bonnes pratiques Airflow



Watchers



Fondamentaux Airflow

TP



**Bonnes pratiques : optimisation de
« mauvais » DAGs**

02

Intégration de bases de données externes

Intégration de bases de données externes

Configuration

Il est possible de configurer les connections Airflow de 2 manières : via l'UI ou avec des variables de configuration.

Nous allons utiliser l'UI.

En fonction du type de base de données, les champs à remplir ne seront évidemment pas exactement les mêmes. Par exemple, on retrouvera le champ `host` pour les BDD hébergées ou locales mais pas pour les services de type SaaS ou API.

Dans notre exemple, nous manipulerons une base Postgresql.

Nous pouvons donner à `Host` le nom du service postgres du fichier `docker-compose` ou (si vous n'utilisez pas Gitpod) `localhost` avec Linux, `host.docker.internal` avec Docker Desktop/Windows/Mac OS.

Les autres champs sont transparents.

En particulier en utilisant Docker, le port doit avoir été ouvert.

Intégration de bases de données externes

TP



**Configuration de la connexion à
Postgresql**

Intégration de bases de données externes

Ecriture

PostgresOperator

```
create_table = PostgresOperator(  
    task_id='create_table',  
    postgres_conn_id='postgre_dwh',  
    sql= """  
        CREATE TABLE IF NOT EXISTS airflow_test (  
            id SERIAL PRIMARY KEY,  
            name VARCHAR NOT NULL,  
            type VARCHAR NOT NULL,  
            date DATE NOT NULL,  
            OWNER VARCHAR NOT NULL);  
        """  
)
```

Intégration de bases de données externes

Ecriture



Ecrire un DAG

Ecrire un DAG qui :

1. crée une table `dag_runs` si elle n'existe pas, contenant les colonnes `dt` (de type `date`) et `dag_id` (de type `character varying`), avec une clé primaire sur (`dt`, `dag_id`),
2. supprime tout enregistrement existant pour la date d'exécution actuelle et l'ID du DAG (utiliser un `PostgresOperator` pour supprimer l'enregistrement de la table `dag_runs` qui correspond à la date d'exécution actuelle et à l'ID du DAG),
3. insère un nouvel enregistrement dans la table `dag_runs` contenant la date d'exécution actuelle et l'ID du DAG (grâce au templating `jinja`)

L'idée est de garantir que la table est à jour avec un seul enregistrement par exécution de DAG et par jour.

La configuration du DAG est laissée libre à ce stade.

Intégration de bases de données externes

Lecture et manipulation

Afin de lire les données d'une table PostgreSQL et les manipuler (par exemple avec pandas), on utilise un Hook.

On peut tout à fait combiner un PostgresOperator pour les opérations SQL pures, et un PythonOperator comportant un Hook pour le traitement des données.

Intégration de bases de données externes

Lecture et manipulation

Le PostgresHook est une interface pour se connecter à une base de données PostgreSQL. Il facilite l'exécution de requêtes SQL et la récupération des résultats.

Import :

```
from airflow.providers.postgres.hooks.postgres import PostgresHook
```

Instanciation :

```
pg_hook = PostgresHook(postgres_conn_id='postgres_default')
```

Intégration de bases de données externes

Lecture et manipulation

Le PostgresHook est une interface pour se connecter à une base de données PostgreSQL. Il facilite l'exécution de requêtes SQL et la récupération des résultats.

Principales méthodes et attributs (1/2) :

get_conn() Etablit la connexion à la base de données PostgreSQL et retourne l'objet connexion.

```
conn = pg_hook.get_conn()
```

get_records() Exécute une requête SQL et retourne tous les résultats sous forme de liste de tuples.

```
sql_query = "SELECT * FROM my_table"  
records = pg_hook.get_records(sql=sql_query)
```

get_first() Exécute une requête SQL et retourne le premier résultat.

```
first_record = pg_hook.get_first(sql=sql_query)
```

Intégration de bases de données externes

Lecture et manipulation

Principales méthodes et attributs (2/2) :

run() Exécute une requête SQL sans retourner de résultats.

```
insert_query = "INSERT INTO my_table (col1, col2) VALUES (%s, %s)"  
pg_hook.run(sql=insert_query, parameters=('value1', 'value2'))
```

insert_rows() Insère plusieurs lignes dans une table en une seule commande.

```
rows = [(1, 'value1'), (2, 'value2')]  
pg_hook.insert_rows(table="my_table", rows=rows)
```

Intégration de bases de données externes

Lecture et manipulation

Curseur

Le PostgresHook utilise des curseurs pour interagir avec la base de données PostgreSQL.

Un curseur est un **objet** utilisé pour exécuter des requêtes SQL et récupérer des résultats.

Création et utilisation d'un curseur

```
pg_hook = PostgresHook(postgres_conn_id='postgres_default')  
conn = pg_hook.get_conn()  
cursor = conn.cursor()
```

Exécution de requêtes SQL avec le curseur et récupération des résultats

```
cursor.execute("SELECT * FROM my_table")  
records = cursor.fetchall()  
print(records)
```


Intégration de bases de données externes

Lecture et manipulation

Curseur

Insertion de données

```
insert_query = "INSERT INTO my_table (col1, col2) VALUES (%s, %s)"  
cursor.execute(insert_query, ('value1', 'value2'))  
conn.commit()
```

Fermeture du curseur et de la connexion

```
cursor.close()  
conn.close()
```

Intégration de bases de données externes

Etude d'un exemple



Hooks

Intégration de bases de données externes

Manipulation des données



A la suite du DAG précédent ou dans un nouveau DAG, créer une tâche consistant à :

1. sélectionner l'ensemble des données de `dags_runs` à l'aide d'une requête SQL,
2. charger ces données dans un dataframe,
3. filtrer grâce à pandas ce dataframe sur un `dag_id` bien précis,
4. afficher dans les logs le dataframe filtré.

03

Scheduling et planification avancée

Scheduling et planification avancée

Implémentation dans notre DAG



Reprendre le DAG précédent et l'adapter de sorte à :

1. restreindre la période d'exécution sur la plage du 1^{er} mai 2024 au 1^{er} juillet 2024,
2. planifier une exécution quotidienne à midi (dans notre timezone), à l'aide d'une expression cron,
3. rattraper les exécutions manquées,
4. implémenter la logique de retry suivante : en cas d'échec, réessayer 1 minute plus tard, jusqu'à 3 fois,
5. définir un SLA de 5 minutes par tâche (nous reviendrons ensuite sur l'effet de ce paramètre),
6. limiter la durée d'exécution totale du DAG à 1 heure.

Scheduling et planification avancée

Visualisation de l'effet du SLA



Dans un DAG à part, définir une première tâche réalisant un sleep de 20 secondes avec un SLA de 10 secondes, ainsi qu'une seconde tâche avec un simple sleep de 10 secondes.

Pour cette question, utiliser l'API TaskFlow.

Visualiser le résultat.

Scheduling et planification avancée

SLA

SLA (accord de niveau de service) : attente concernant le temps maximum dans lequel une tâche doit être terminée par rapport à l'heure de début de l'exécution du DAG.

Si une tâche prend plus de temps que cela pour s'exécuter, elle sera alors visible dans la section "SLA Misses" de l'interface utilisateur, ainsi qu'envoyée par email avec toutes les tâches qui ont manqué leur SLA pour peu d'avoir configuré le serveur SMTP et défini les paramètres d'email dans le DAG (nous ne le ferons pas).

Les tâches dépassant leur SLA ne sont pas annulées - elles sont autorisées à s'exécuter jusqu'à leur achèvement.

➡ Pour annuler une tâche après un certain temps d'exécution, il faut utiliser des timeouts.

Pour définir un SLA pour une tâche, on passe un objet `datetime.timedelta` au paramètre `sla` de la tâche ou de l'opérateur. On peut également fournir un `sla_miss_callback` qui sera appelé lorsque le SLA est manqué, pour exécuter une logique spécifique.

Il est possible de désactiver complètement la vérification des SLA après-coup, en définissant `check_slas = False` dans la configuration `[core]` d'Airflow.

Scheduling et planification avancée

Pour aller plus loin : configurer l'envoi d'emails

<https://airflow.apache.org/docs/apache-airflow/stable/howto/email-config.html>

Scheduling et planification avancée

Tâches zombies

Aucun système ne fonctionne parfaitement, et il va forcément arriver que des instances de tâches échouent, de temps en temps.

Airflow détecte deux types de mismatch tâches/processus :

Les tâches **zombies** sont des instances de tâches bloquées dans un état d'exécution malgré l'inactivité de leurs jobs associés (leur processus n'a pas envoyé de signal de vie récent car il a été tué, ou la machine est tombée en panne).

➡ Airflow les trouve périodiquement, les nettoie, et échoue ou réessaie la tâche en fonction de ses paramètres.

Les tâches **mortes-vivantes** sont des tâches qui ne sont pas censées être en cours d'exécution mais qui le sont (elles sont souvent causées par une modification manuelle des instances de tâches via l'interface utilisateur).

➡ Airflow les trouve périodiquement et les termine.

Scheduling et planification avancée

Etude de code



**Snippet du scheduler Airflow pour la
détection des tâches zombies/mort-
vivantes**

Scheduling et planification avancée

Variables d'environnement liées à une tâche zombie



```
export AIRFLOW__SCHEDULER__LOCAL_TASK_JOB_HEARTBEAT_SEC=600  
export AIRFLOW__SCHEDULER__SCHEDULER_ZOMBIE_TASK_THRESHOLD=2  
export AIRFLOW__SCHEDULER__ZOMBIE_DETECTION_INTERVAL=5
```

04

Création et gestion de workflows complexes

Création et gestion de workflows complexes

Versionnage

La gestion de versions des DAGs dans Apache Airflow est cruciale pour s'assurer que les modifications apportées aux DAGs sont suivies, testées et déployées de manière contrôlée.

Que peut-on faire en pratique ?

Utiliser un système de contrôle de version (Git), de sorte à stocker les DAGs dans un dépôt et suivre les modifications à travers des commits.

Versionner les DAGs dans le code

```
from airflow import DAG
from datetime import datetime, timedelta

VERSION = '1.0.0'

with DAG(
    dag_id=f'dag_with_version_{VERSION}',
) as dag:
    pass
```

Création et gestion de workflows complexes

CI/CD

Intégrer Airflow dans un pipeline de déploiement CI/CD (e.g., GitHub Actions, Jenkins, GitLab CI) permet bien évidemment d'automatiser les tests et les déploiements des DAGs.

```
name: Deploy DAGs

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.8'

      - name: Install dependencies
        run: pip install apache-airflow

      - name: Deploy DAGs
        run: |
          scp -r dags/ user@your-airflow-server:/path/to/airflow/dags/
```

Création et gestion de workflows complexes

Etude d'un DAG



Lien entre versionning et branching

Création et gestion de workflows complexes

Priority weights

Les poids de priorité (`priority_weight`) des tâches dans Airflow sont utilisés pour gérer l'ordre d'exécution des tâches lorsque les ressources sont limitées.

Lorsqu'un DAG contient plusieurs tâches qui peuvent être exécutées en parallèle, Airflow utilise les poids de priorité pour décider quelles tâches exécuter en premier.

Cela est particulièrement utile dans les environnements où les ressources de calcul (comme les slots de travail, la capacité du cluster, etc.) sont limitées.

Bien entendu, ce sont les tâches avec des poids de priorité les plus élevés qui seront exécutées avant celles avec des poids plus faibles, pour peu qu'elles soient en concurrence pour les mêmes ressources.

Création et gestion de workflows complexes

Deferrals

Les opérateurs différés ou différables (deferred operators) dans Airflow sont conçus pour améliorer l'efficacité des ressources en ne consommant pas de slots de travail pendant qu'ils attendent qu'une condition soit remplie. Les tâches différées utilisent des triggers pour se réveiller quand la condition est remplie, ce qui permet à Airflow de gérer plus efficacement les pauses et les attentes.

Un opérateur différable est écrit avec la capacité de se suspendre et de libérer le travailleur lorsqu'il sait qu'il doit attendre, et de confier la reprise à quelque chose appelé un déclencheur. Par conséquent, lorsqu'il est suspendu (différé), il ne prend pas d'emplacement du worker et le cluster dépense beaucoup moins de ressources sur des opérateurs ou des capteurs inactifs.

En pratique, les déclencheurs sont de petits morceaux de code Python asynchrones conçus pour être exécutés tous ensemble dans un seul processus Python ; étant donné qu'ils sont asynchrones, ils peuvent tous coexister efficacement.

Utiliser des opérateurs différables en tant qu'auteur de DAG est presque transparent ; les écrire, cependant, demande un peu plus de travail.

#Efficacité_des_ressources

#Scalabilité

Création et gestion de workflows complexes

Aller plus loin avec les deferrals

Pour apprendre à écrire ses propres deferrals (peu classique) :

<https://airflow.apache.org/docs/apache-airflow/2.2.5/concepts/deferring.html>

Création et gestion de workflows complexes

Priority weights et deferrals



Créer un DAG commençant par une tâche comportant un `TimeDeltaSensor` de 5 minutes, puis deux tâches en parallèle, une de haute priorité, une de basse priorité, simulées par des `PythonOperator` faisant appel à une `dummy_task_function` loggant simplement la tâche effectuée au niveau INFO.

Remarquer le comportement du `TimeDeltaSensor` dans le passé.

Remplacer le `TimeDeltaSensor` par un `DateTimeSensorAsync` avec un `target_time` égal au temps d'exécution + 5 minutes.

05

Pipeline complet

Pipeline complet

TP



**Retour sur l'application « API
financière »**

Création et gestion de workflows complexes

API financière



Pour ceux qui avaient fait la formation prise en main, nous avons vu que nous étions limités dans nos requêtes à l'API.

Recoder le DAG final de l'API financière en suivant les consignes de la formation Prise en main (pour ceux qui ont fait cette formation, c'est l'occasion de mesurer vos progrès !)

Adapter le DAG avec une logique de retry permettant de gérer les échecs dus à cette cause.

Faire évoluer le DAG afin d'insérer les résultats dans une table Postgres (attention à la question des clefs primaires dupliquées...).

Une fois les résultats insérés, recharger l'ensemble de la table Postgres.

Si le nombre de lignes total est strictement supérieur à 100, lancer le traitement de calcul des moyennes hebdomadaires.

Sinon, afficher dans le terminal qu'il manque des données pour que le résultat soit exploitable.

06

Bonnes pratiques

Bonnes pratiques

Minimiser les dépendances inter-DAGS

Pourquoi minimiser les dépendances inter-DAGs ?

Complexité réduite

Les dépendances entre DAGs peuvent rendre le flux de travail difficile à comprendre et à maintenir.

Chaque changement dans un DAG peut avoir un impact sur d'autres DAGs, augmentant ainsi la charge de gestion.

Isolation des échecs

Si un DAG échoue, les dépendances inter-DAGs peuvent propager cet échec à d'autres DAGs. En gardant les DAGs indépendants, on limite l'impact des échecs.

Facilité de débogage

Les DAGs indépendants sont plus faciles à déboguer, car il est plus simple de comprendre le flux de données et de diagnostiquer les problèmes sans avoir à considérer les interactions complexes avec d'autres DAGs.

Scalabilité

Les pipelines de données peuvent être facilement mis à l'échelle lorsque les DAGs sont indépendants. Les dépendances inter-DAGs peuvent créer des goulots d'étranglement et des points de contention.

Bonnes pratiques

Minimiser les dépendances inter-DAGS

Stratégies pour minimiser les dépendances inter-DAGs (1/2)

Utiliser les capteurs (sensors)

Les capteurs permettent de créer des points de synchronisation légers sans établir de dépendances directes. Par exemple, on peut utiliser un `ExternalTaskSensor` pour attendre l'achèvement d'une tâche dans un autre DAG sans créer de dépendance directe.

Centraliser les données partagées

On peut utiliser des systèmes de stockage de données centralisés comme des bases de données, des data lakes ou des files d'attente de messages pour partager des données entre DAGs. Chaque DAG peut lire ou écrire dans ces systèmes indépendamment. Les systèmes de messagerie peuvent être utilisés pour orchestrer des événements entre DAGs sans établir de dépendances directes.

Passer des métadonnées via une base de données ou un service de métadonnées

Pour partager des informations entre DAGs, penser à utiliser une base de données centrale ou un service de métadonnées- pour stocker et récupérer ces informations.

Bonnes pratiques

Minimiser les dépendances inter-DAGS

Stratégies pour minimiser les dépendances inter-DAGs (2/2)

Décomposer les tâches complexes en sous-DAGs

Si un flux de travail est trop complexe, il faut envisager de le décomposer en plusieurs sous-DAGs ou DAGs plus petits et indépendants qui peuvent être exécutés de manière autonome.

VERSUS

Utiliser des tâches locales autant que possible

Plutôt que de déclencher des tâches dans d'autres DAG, essayez de gérer autant de logique que possible à l'intérieur d'un seul DAG en utilisant des dépendances entre les tâches.

Partager le code commun

Si plusieurs DAG ont besoin d'une logique similaire, extrayez cette logique dans des fonctions ou des modules Python réutilisables et partagez-les entre les DAG.

Utilisez des déclencheurs de DAG judicieusement

Les déclencheurs de DAG peuvent être utiles dans certains cas, mais il faut éviter de les utiliser de manière excessive, car cela peut compliquer la gestion des dépendances.

Utiliser des variables Airflow

Les variables Airflow peuvent être utilisées pour partager des données entre les DAG de manière sécurisée et efficace.

Bonnes pratiques

Utiliser des templates

Templates = rendre les DAGs plus dynamiques, réutilisables et faciles à maintenir.

Paramétrage dynamique des tâches Dates, chemins de fichiers, noms de tables...

De manière plus générale

Réutilisation de code

L'extraire dans des fonctions, des classes, utiliser des templates pour les paramètres variables.

Centralisation de la configuration

Centraliser la configuration répétitive, telle que les connexions à la base de données, les adresses e-mail, les chemins de fichiers... grâce à des templates. Cela facilite la maintenance en évitant la duplication de code.

Utilisation des opérateurs de templates intégrés

Validation des templates

Et bien sûr bien documenter tout cela !

<https://airflow.apache.org/docs/apache-airflow/stable/templates-ref.html>

Bonnes pratiques

Template personnalisé

```
def custom_function():  
    return "custom_value"  
  
templated_command = """  
echo "Custom value is {{ custom_function() }}"  
"""
```

Bonnes pratiques

Eviter les dépendances circulaires

Une dépendance circulaire se produit lorsque deux tâches ou plus se dépendent mutuellement, créant ainsi une boucle infinie qui empêche l'exécution correcte des tâches.

C'est du bon sens : éviter les dépendances circulaires dans Airflow est crucial pour garantir la stabilité et la fiabilité des workflows.

Et pour cela :

- Bien planifier le workflow

- Décomposer les tâches complexes

- Vérifier que le DAG est bien un DAG, en particulier en notamment les visualisations d'Airflow

- Créer des sous-DAGs

- Utiliser des opérateurs sensoriels et de déclenchement (plutôt que de créer des dépendances directes entre les tâches dans différents DAGs)

- Eviter les dépendances inter-DAG

- Analyser les logs et plus généralement faire du monitoring

Bonnes pratiques

Paralléliser les tâches

Bien paralléliser ce qui peut l'être !

Ne pas coder que des tâches sont séquentielles si elles ne le sont pas : on se prive d'une optimisation.

Bonnes pratiques

Paralléliser les tâches et gérer la concurrence

Configurer les paramètres en lien avec la parallélisation, en particulier le nombre maximum de tâches et de DAGs concurremment exécutées (`max_active_tasks_per_dag` et `max_active_runs`).

```
[core]
max_active_tasks_per_dag = 16
```

```
from airflow import DAG
from datetime import datetime

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
}

dag = DAG(
    'example_dag',
    default_args=default_args,
    description='A simple tutorial DAG',
    schedule_interval='@daily',
    start_date=datetime(2024, 6, 1),
    catchup=False,
    max_active_runs=3 # Limite à 3 DAGs actifs
)
```

Bonnes pratiques

Paralléliser les tâches et gérer la concurrence

Utiliser des pools de tâches

Les pools permettent de limiter le nombre de tâches en cours d'exécution pour certains types de ressources. On peut les définir dans l'interface utilisateur Airflow ou dans le fichier `airflow.cfg`

```
airflow pool -s my_pool 10 "Description du pool"
```

Puis, attribuer des pools aux tâches pour gérer les ressources.

Bonnes pratiques

Paralléliser les tâches et gérer la concurrence

Utiliser des pools de tâches

```
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime

with DAG() as dag:

    task1 = BashOperator(
        task_id='task1',
        bash_command='echo "Task 1"',
    )

    task2 = BashOperator(
        task_id='task2',
        bash_command='echo "Task 2"',
        pool='my_pool'
    )

    task3 = BashOperator(
        task_id='task3',
        bash_command='echo "Task 3"',
        pool='my_pool'
    )

    task4 = BashOperator(
        task_id='task4',
        bash_command='echo "Task 4"',
    )

    task1 >> [task2, task3] >> task4
```

Bonnes pratiques

Paralléliser les tâches et gérer la concurrence

Optimiser le cluster Airflow

Le scheduler d'Airflow peut (doit) être configuré pour gérer efficacement un grand nombre de tâches. Pour cela, on ajuste principalement les paramètres `scheduler_heartbeat_sec`, `min_file_process_interval`, et `dag_dir_list_interval` dans `airflow.cfg`.

```
[scheduler]
scheduler_heartbeat_sec = 5
min_file_process_interval = 30
dag_dir_list_interval = 60
```

Bonnes pratiques

Gérer la périodicité pour éviter les surcharges

On pourrait l'oublier mais, au final, la gestion de la périodicité est essentielle pour éviter les surcharges du système et garantir que les workflows s'exécutent de manière fluide et efficace.

Utiliser des intervalles de planification appropriés (éviter les planifications trop fréquentes)

Utiliser le catchup à bon escient

Utiliser les fenêtres de temps

Bonnes pratiques

Eviter les longs temps d'attente et les blocages

Déjà, **optimiser les tâches individuelles** (design du code, utilisation de caches, élimination des opérations redondantes, optimisation des requêtes SQL...) !

Configurer correctement les paramètres de performance dont on a parlé (parallelism, dag_concurrency, max_active_runs, concurrency...).

Une fois cela réalisé, l'optimisation viendra d'éléments spécifiques classiques plus larges que le monde d'Airflow. Par exemple, utiliser des indexes sur les bases de données pour aider à accélérer les opérations de lecture et d'écriture, en particulier sur les grandes bases de données.

Bonnes pratiques

Last but not least, planifier la reprise après échec

Tout système est sujet à panne.

Déjà, d'un point de vue métier, bien penser au scénario à suivre en cas d'échec.

Côté Airflow, implémenter une logique de retry, gérer le catchup, utiliser des capteurs de temps mais aussi de status.

En cas de criticité élevée (à utiliser avec modération), éventuellement penser à planifier des exécutions redondantes. Dans ce cas, prêter une attention particulière à :

- ➡ l'éventuelle non-idempotence des traitements,
- ➡ la surcharge des ressources,
- ➡ les coûts supplémentaires,
- ➡ la complexité accrue du projet Airflow.

Fin de la formation

Merci !