Documentation Technique du Pipeline Dagster

Introduction :

Ce pipeline a été conçu pour automatiser le processus de scraping, de nettoyage et de traitement des données de produits à partir du site web de Carter Cash en utilisant l'orchestrateur Dagster. Les données sont stockées dans une base de données Azure.

Configuration :

Le fichier `.env` contient les informations de connexion à la base de données Azure. Pour effectuer les injections dans la base de données (BDD), nous avons utilisé un fichier `.env` contenant les informations de connexion à la BDD. Ce fichier contient les variables d'environnement suivantes :

- **DB_SERVER** : le nom du serveur de la BDD
- **DB_DATABASE** : le nom de la base de données
- **DB_USERNAME** : le nom d'utilisateur pour se connecter à la BDD
- **DB_PASSWORD** : le mot de passe pour se connecter à la BDD

Ces variables d'environnement sont utilisées dans le code pour se connecter à la BDD et effectuer les opérations d'injection de données.

Les tâches du pipeline :

Tâche 1 : "1_Azure_Scrapy"

Cette tâche exécute le script de scraping qui récupère les données de produits à partir du site web de Carter Cash en utilisant Scrapy.

Tâche 2 : "2_Azure_Count"

Cette tâche compte le nombre de lignes insérées dans la base de données Azure après l'exécution du script de scraping.

Tâche 3 : "3_Azure_Nettoyage"

Cette tâche exécute le script de nettoyage qui met à jour les URL des produits dans la base de données Azure en suivant les redirections.

Tâche 4 : "4_Azure_delete_doublon"

Cette tâche supprime les doublons de produits dans la base de données Azure en se basant sur l'URL et le prix.

Tâche 5 : "5_Azure_changement_prix"

Cette tâche met à jour les prix des produits dans la base de données Azure en se basant sur les informations récupérées à partir des URL des produits.

Tâche 6 : "6_Azure_ajouts_marque"

Cette tâche ajoute la marque des produits dans la base de données Azure en utilisant le premier mot de la description du produit.

Tâche 7 : "7_Azure_delete_666"

Cette tâche supprime les produits avec un prix de 666 de la base de données Azure.

Dépendances :

Chaque tâche dépend de la tâche précédente. Si une tâche échoue, le pipeline s'arrête et les tâches suivantes ne sont pas exécutées.

Groupes de tâches :

Les tâches sont regroupées en un seul groupe appelé "azure_tasks".

Planification :

Le pipeline est planifié pour s'exécuter tous les 7 jours à 8h00 du matin, heure de Paris.

Conclusion :

Ce pipeline permet d'automatiser le processus de collecte, de nettoyage et de traitement des données de produits à partir du site web de Carter Cash en utilisant Dagster et en stockant les données dans une base de données Azure.

Exemple de code du fonctionnement du pipeline tache N°1 puis N°2 :

```
@asset(key="1_Azure_Scrapy", group_name="azure_tasks")
def execute_Scrapy(context: AssetExecutionContext):
    """
    Lancement du scraping de carter cash
    """
    context.log.info("Début de l'exécution de la tâche de scraping.")
    try:
        # cherche le chemin du script de scraping
        scrapy_project_path = "Script_projet/leboncoin/leboncoin/spiders"

        os.chdir(scrapy_project_path)

        # Run Scrapy spider
        subprocess.run(["scrapy", "crawl", "carter"])

        context.log.info("Scraping terminé avec succès.")
        return True
    except Exception as e:
        raise Failure(f"Erreur lors de l'exécution du scraping : {str(e)}")

#############################################################################################

@asset(key="2_Azure_Count", group_name="azure_tasks", ins={"upstream":
AssetIn(key="1_Azure_Scrapy")})
def execute_Count(context: AssetExecutionContext, upstream: bool):
    """
   nombre de lignes injecter lors du dernier scraping
    """
    if not upstream:
        context.log.info("La tâche précédente a échoué, donc cette tâche ne sera pas exécutée.")
        return

    context.log.info("Début de l'exécution de 1_bis_count_inject.py")
    try:
        result = subprocess.run(["python", "Script_projet/1_bis_count_inject.py"],
capture_output=True, text=True)
        if result.returncode != 0:
            raise Failure(f"Erreur lors de l'exécution de 1_bis_count_inject.py :
{result.stderr}")
        else:
            lines_inserted = [int(s) for s in result.stdout.split() if s.isdigit()]
            if lines_inserted:
                context.log.info(f"{lines_inserted[0]} lignes ont été insérées.")
            context.log.info("1_bis_count_inject.py exécuté avec succès")
            return True
    except FileNotFoundError as e:
        raise Failure(f"Erreur lors de l'exécution de 1_bis_count_inject.py : {str(e)}")
```