**Une image contenant Police, Graphique, logo, graphisme

Description générée automatiquement**

**Architecture Logicielle (A2025)**

**Rapport de Laboratoire**

|  |  |
| --- | --- |
| **Nᵒ du laboratoire** | Laboratoire 1 |
| **Étudiant(s)** | Makesa, Mvuemba Gildor |
| **Cours** | LOG430 |
| **Session** | Aut. 2025 |
| **Groupe** | LOG430-01-02 |
| **Professeur** | Fabio Petrillo |
| **Chargé de Laboratoire** | Gabriel C. Ullmann |
| **Date** | 18 sept. 25 |

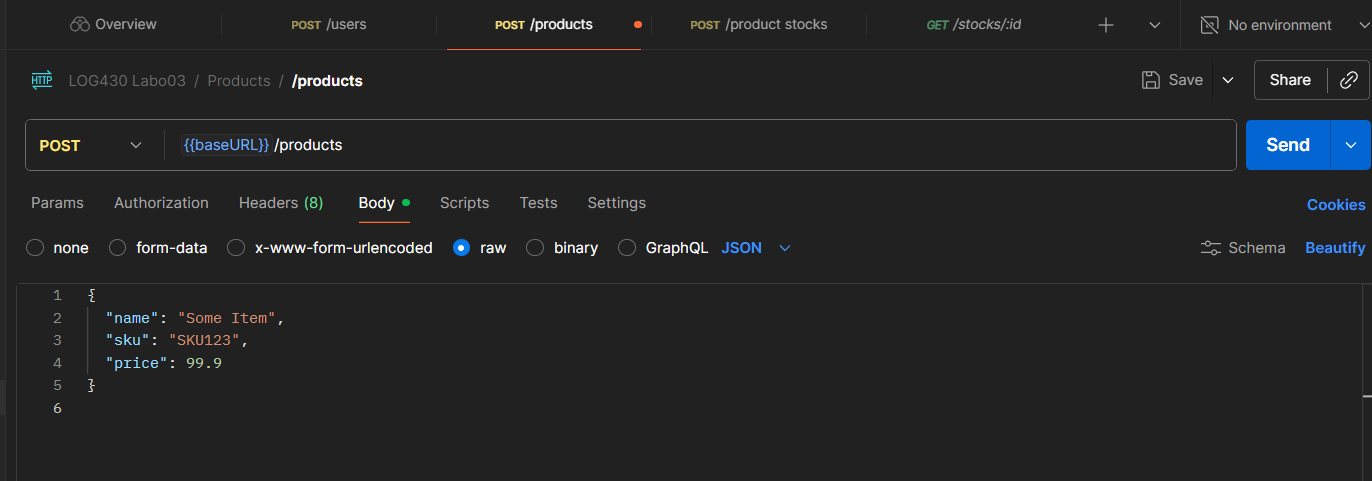
# 💡 Question 1

Quel nombre d'unités de stock pour votre article avez-vous obtenu à la fin du test ? Et pour l'article avec id=2 ? Veuillez inclure la sortie de votre Postman pour illustrer votre réponse

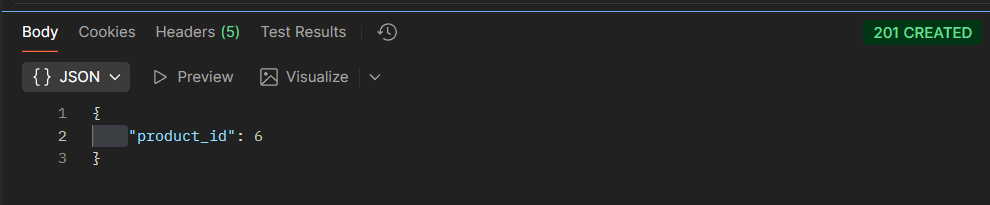
Dans test\_stock\_flow, on :

**A. Créer un produit dans Postman**

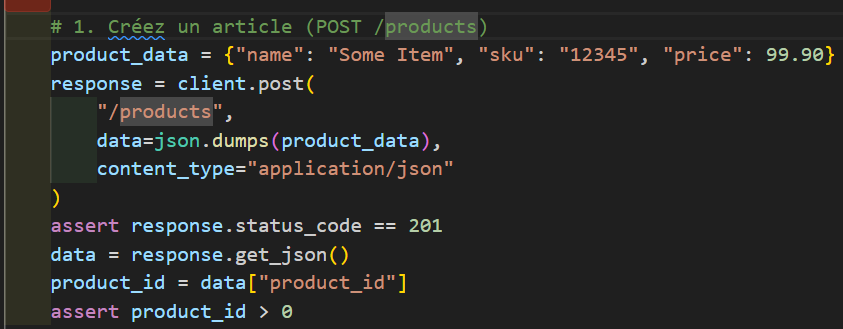
1. Sélectionne la requête **POST {{baseURL}}/products**.
2. Onglet **Headers** → ajoute Content-Type: application/json.
3. Onglet **Body** → **raw** + **JSON** et mets un **produit complet** :



1. Clique **Send**.
2. Réponse attendue (201 Created) :

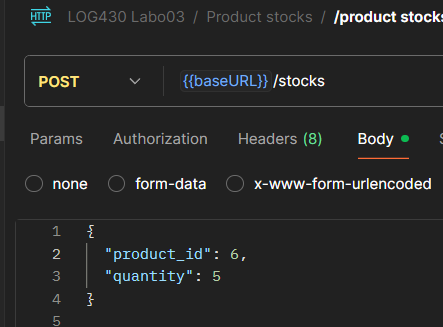


**Tests (onglet “Tests”)** — pour stocker l’ID produit :

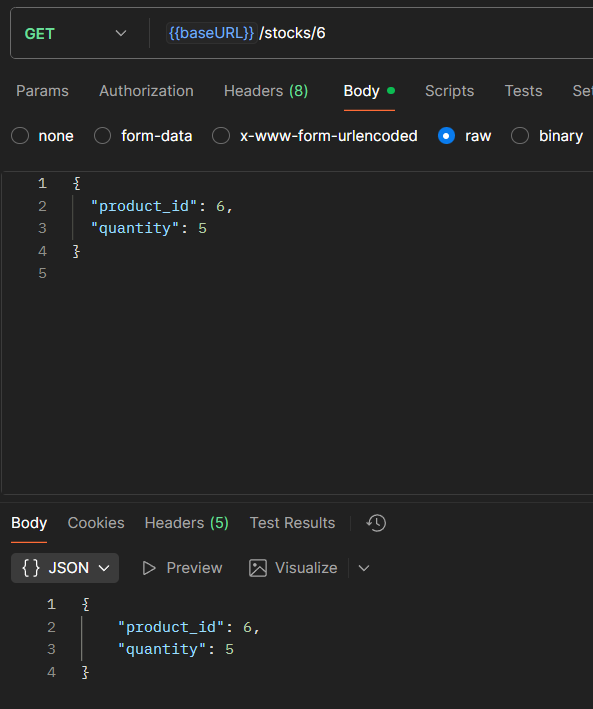


**B. Ajouter du stock pour ce produit**

1. Ensuite, pour augmenter le stock, utilise **POST {{baseURL}}/stocks** avec :



1. Puis vérifie : GET {{baseURL}}/stocks/6

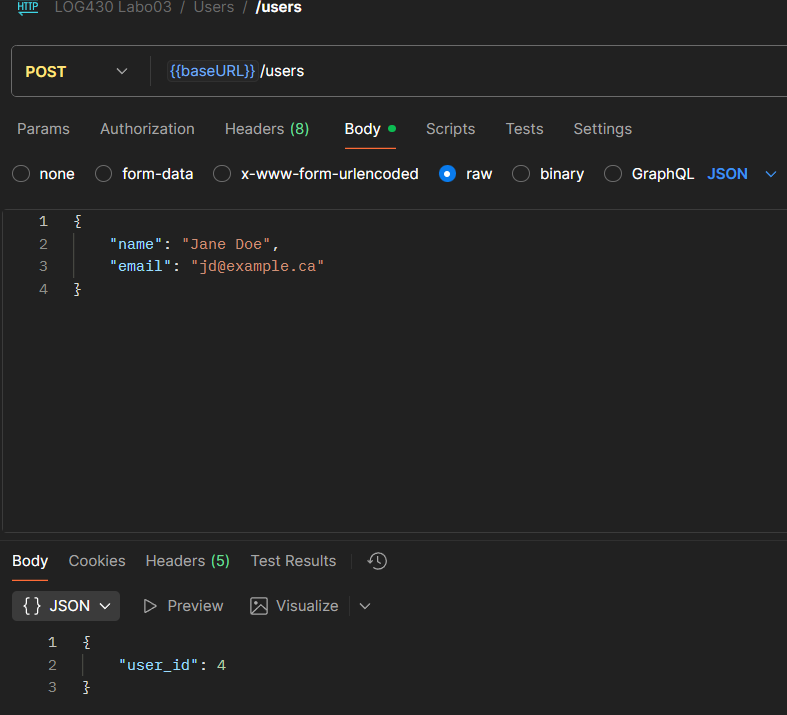


**Tests (onglet “Tests”)** — pour ajouter 5 unité de produit



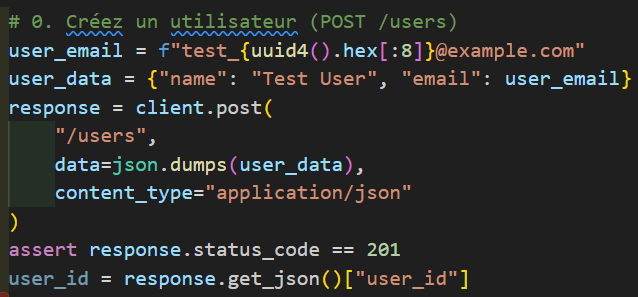
**C.** **Créez une commande de 2 unités (POST /orders)**

1. Tout d’abord, créons un utilisateur en sélectionnant la requête **POST {{baseURL}}/users**.

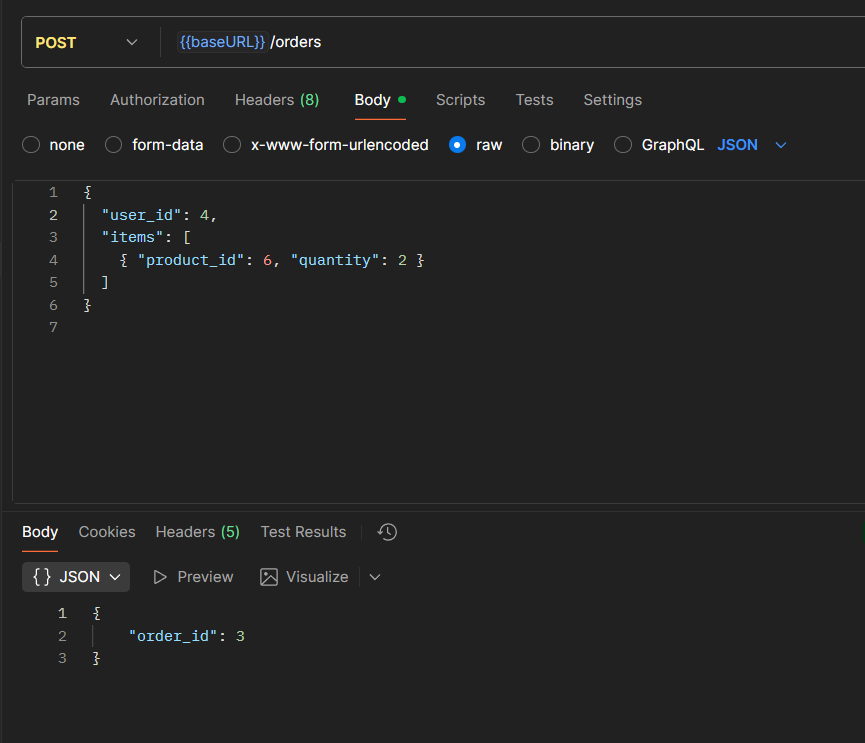


\*Remarque : *Prendre note du user\_id :4*

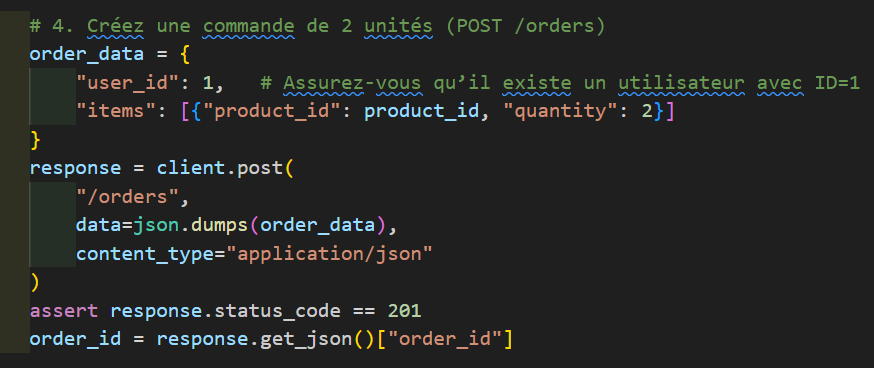
**Tests (onglet “Tests”)** — pour ajouter un user



1. Par la suite, passons une commande de 2 unités dans cet utilisateur :

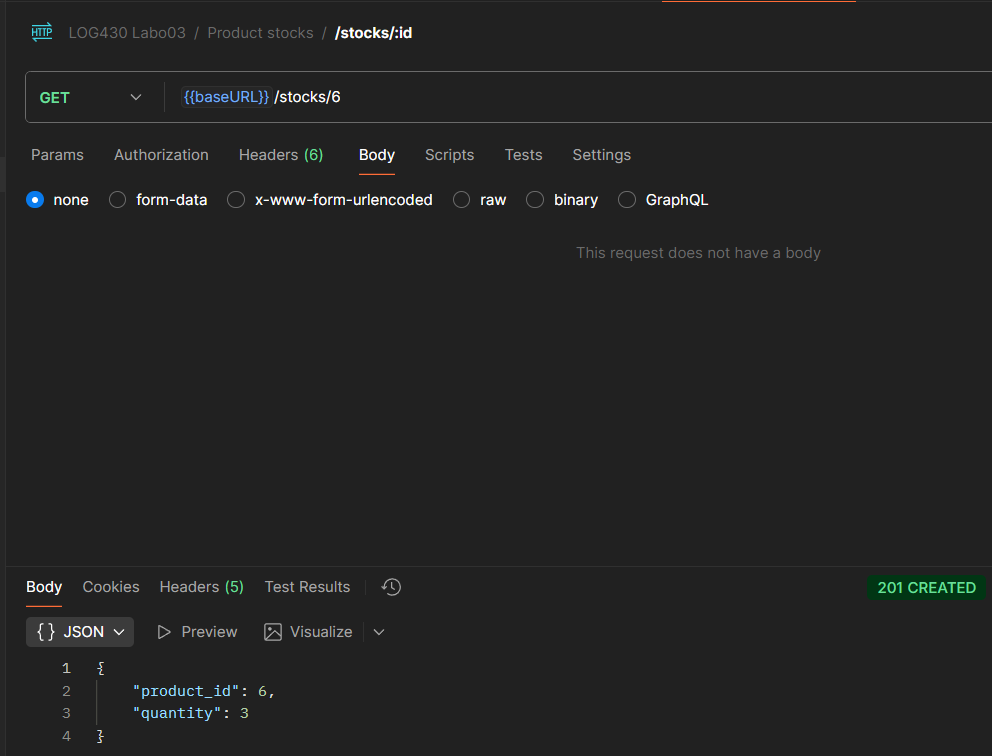


**Tests (onglet “Tests”)** — pour créer une commande de 2 unités (POST /orders)

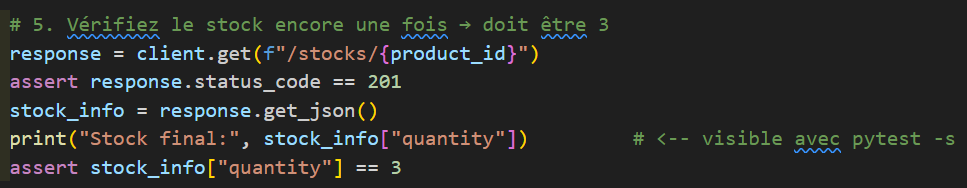


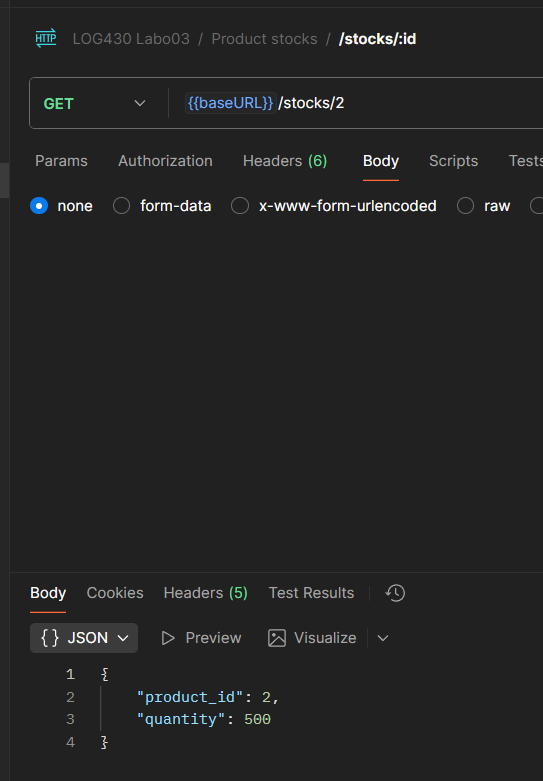
**D.** **Vérifiez le stock encore une fois → doit être 3**

1. Relire le stock **{{baseURL}}/stocks/{{product\_id}}**



**Tests (onglet “Tests”)** — pour Vérifiez le stock encore une fois



Et pour l’article avec le ID = 2 toujours 500 :  


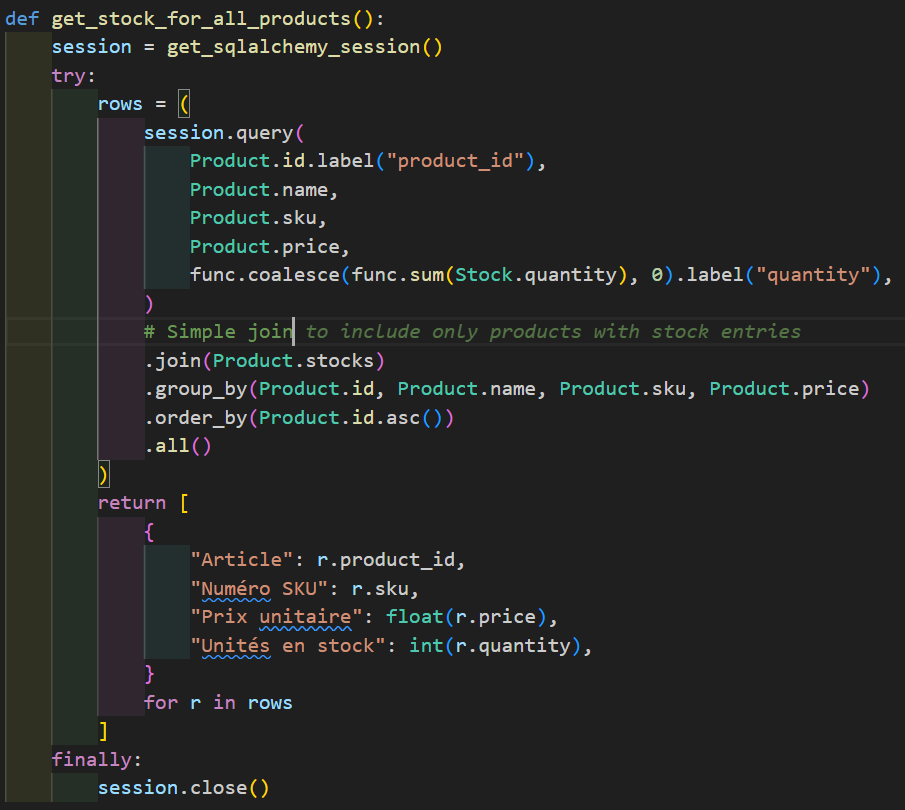
# 💡 Question 2

Décrivez l'utilisation de la méthode join dans ce cas. Utilisez les méthodes telles que décrites à Simple Relationship Joins et Joins to a Target with an ON Clause dans la documentation SQLAlchemy pour ajouter les colonnes demandées dans cette activité. Veuillez inclure le code pour illustrer votre réponse.

L'utilisation de la méthode **join** dans le contexte de la fonction get\_stock\_for\_all\_products() se référe aux mécanismes de jointures de SQLAlchemy. L'objectif est de combiner les données des tables **Product** et **Stock** pour obtenir les informations de base de chaque produit ainsi que la quantité en stock correspondante.

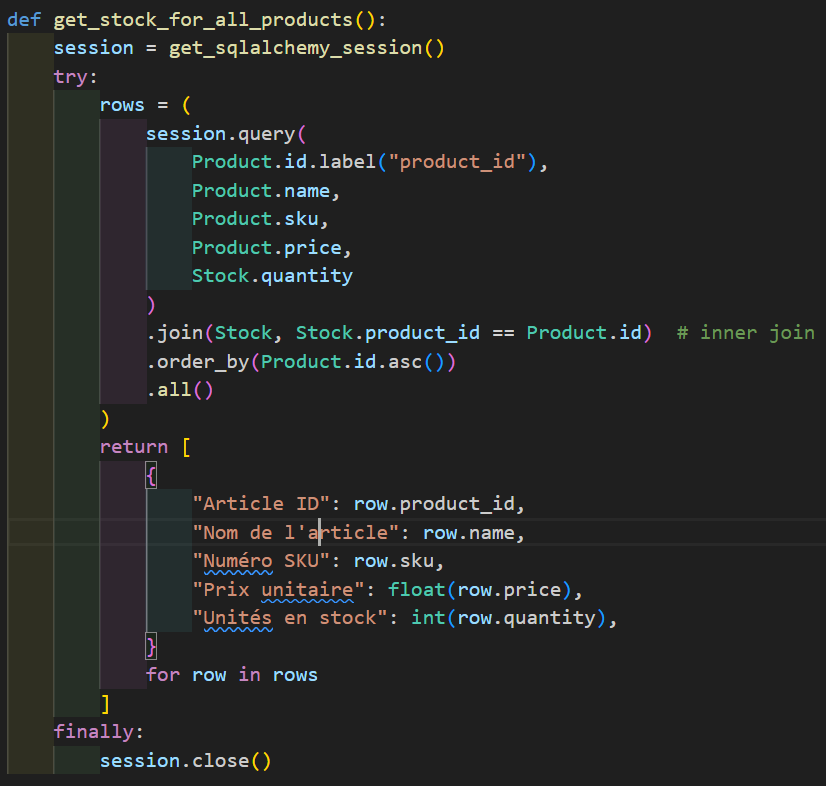
**1. Rôle de la méthode join**

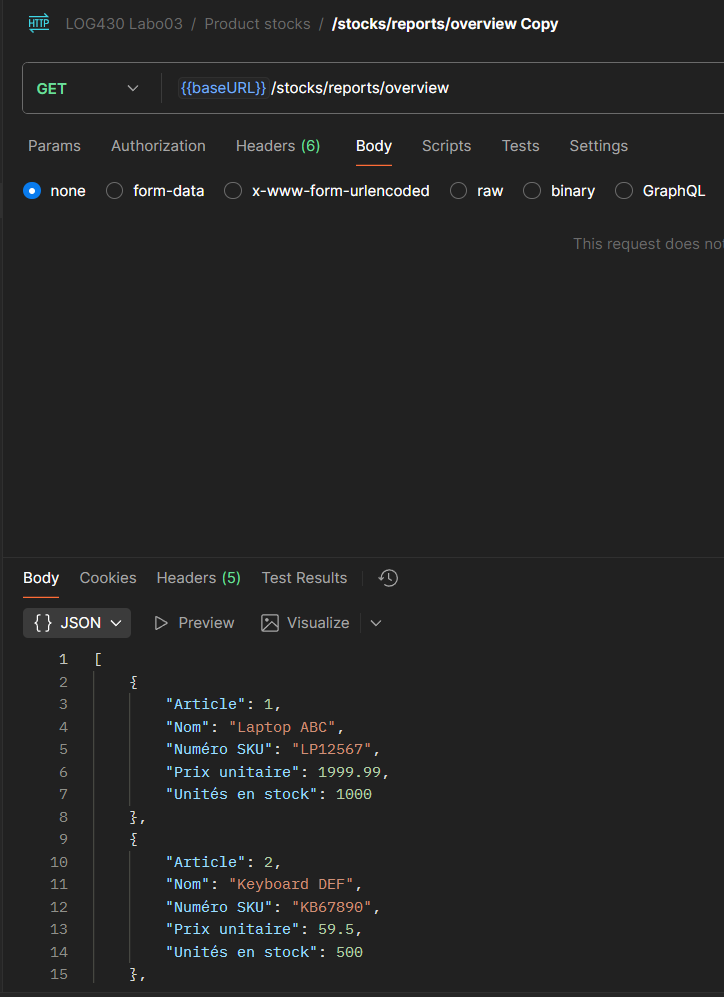
Dans ce cas, la méthode join est utilisée pour effectuer une **jointure interne explicite** entre la table principale **Product** (qui est la source de la requête via session.query(Product...)) et la table **Stock**. Cette opération permet de lier les lignes des deux tables lorsque les valeurs d'une colonne spécifiée correspondent.



**2. Rôle de la méthode join**

L'utilisation de la méthode join pour spécifier la table cible et la condition est une application de la technique **"Joins to a Target with an ON Clause"** de la documentation SQLAlchemy. Elle définit la condition selon laquelle la jointure doit être effectuée. Elle stipule que seules les lignes où la colonne product\_id de la table Stock correspond à la colonne id de la table Product sont combinées et incluses dans le résultat.





# 💡 Question 3

Quels résultats avez-vous obtenus en utilisant l’endpoint POST /stocks/graphql avec la requête suggérée? Veuillez joindre la sortie de votre requête dans Postman afin d’illustrer votre réponse.

En utilisant l'endpoint POST /stocks/graphql avec la requête GraphQL suggérée dans Postman, nous avons obtenu les informations spécifiques du produit ayant l'**ID 1**, en incluant les données de base et la quantité en stock.

La requête, qui ciblait la méthode product(id: 1), a explicitement demandé les champs id, name, sku, et quantity.

**Résultats obtenus**

Les résultats retournés dans le corps de la réponse JSON sont les suivants, confirmant que l'endpoint a réussi à agréger les données du produit et de l'inventaire :

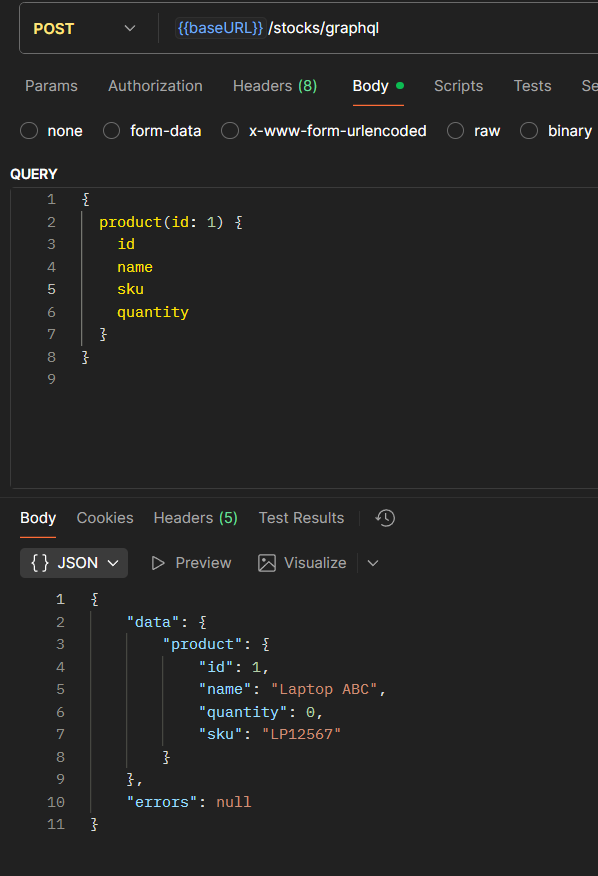
|  |  |  |
| --- | --- | --- |
| **Champs** | **Valeur obtenue** | **Interprétation** |
| ID | 1 | L’identifiant du produit |
| Name | "Laptop ABC" | Le nom du produit. |
| sku | "LP12567" | La référence unique du produit (SKU) |
| Quantity | 0 | La quantité en stock de ce produit (donnée d'inventaire). |

L'objet JSON retourné ne contient **aucune erreur** ("errors": null), ce qui signifie que la requête a été exécutée avec succès et que les champs demandés ont été récupérés.

**llustration de la réponse**

L'image jointe, représentant l'interface Postman, sert d'illustration directe à cette réponse :

1. **Requête** : Le panneau *Query* montre la requête GraphQL envoyée
2. **Réponse** : Le panneau *Body (JSON)* montre le résultat sur mesure



# 💡 Question 4

Quelles lignes avez-vous changé dans update\_stock\_redis? Veuillez joindre du code afin d’illustrer votre réponse.

Les modifications apportées à la fonction update\_stock\_redis visaient à garantir la **cohérence** des données dans **Redis** en gérant deux aspects principaux :

1. La mise à jour de la **quantité** en stock avec un **garde-fou** pour éviter les valeurs négatives.
2. L'enrichissement des **métadonnées** du produit (nom, SKU, prix) dans Redis, en ne les récupérant de MySQL que si elles sont manquantes.

Ainsi,

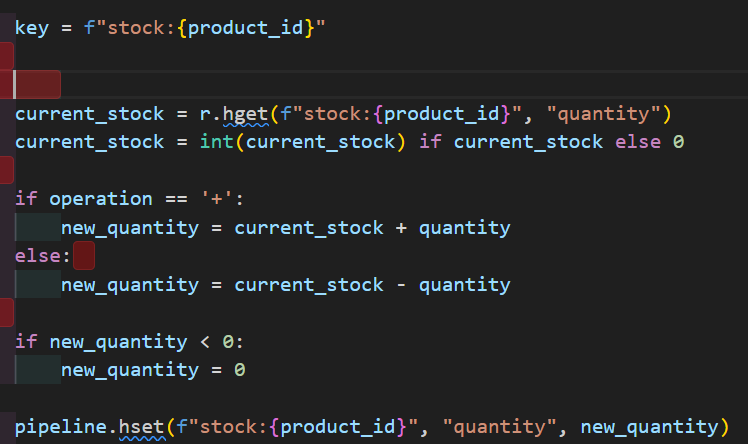
**1. Initialisation de la session SQLAlchemy**

Une session de base de données est ouverte **avant** la boucle de traitement des articles de commande, afin de ne l'ouvrir qu'une seule fois, pour une performance optimale.



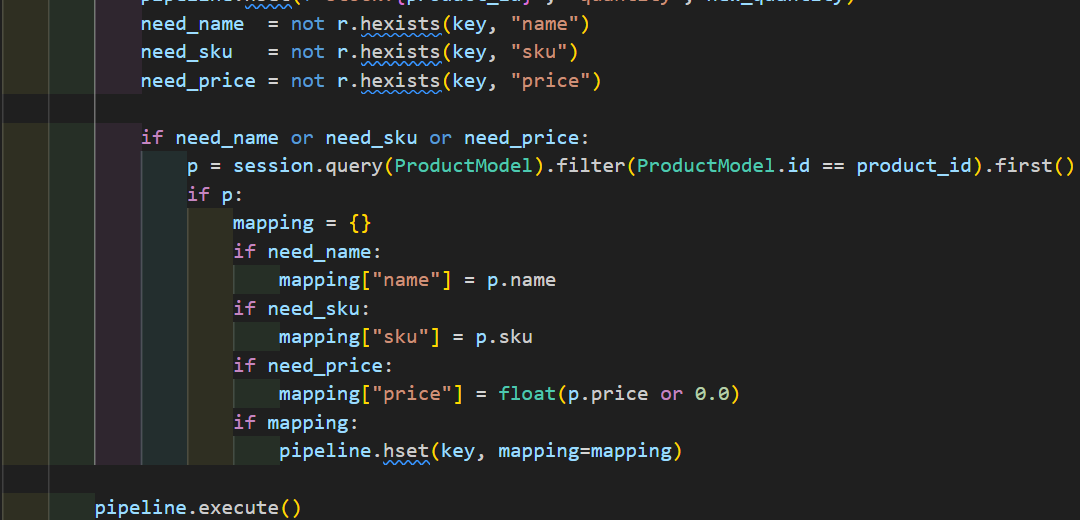
**2. Gestion de la quantité et du garde-fou (Clamp à Zéro)**

Ces lignes définissent la clé Redis et introduisent la logique qui assure que la quantité en stock ne descend jamais en dessous de zéro.



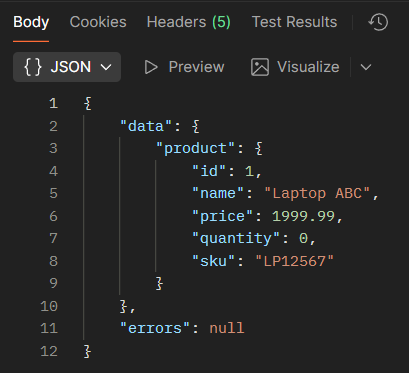
**3. Complétion des métadonnées manquantes (Ajout complet)**

Ce bloc de code est l'ajout le plus significatif. Il vérifie l'existence des champs de métadonnées dans Redis et effectue une requête ciblée vers MySQL seulement si des informations sont manquantes.



# 💡 Question 5

Quels résultats avez-vous obtenus en utilisant l’endpoint POST /stocks/graphql avec les améliorations ? Veuillez joindre la sortie de votre requête dans Postman afin d’illustrer votre réponse.



# 💡 Question 6

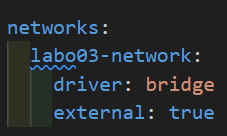
Examinez attentivement le fichier docker-compose.yml du répertoire scripts, ainsi que celui situé à la racine du projet. Qu’ont-ils en commun ? Par quel mécanisme ces conteneurs peuvent-ils communiquer entre eux ? Veuillez joindre du code YML afin d’illustrer votre réponse.

Ils ont en commun qu’ils attachent leurs services au **même réseau Docker** (labo03-network).C’est exactement ce réseau partagé qui permet la **découverte DNS automatique** et la **communication inter-conteneurs**.

C'est ce réseau partagé qui est le fondement de la communication inter-conteneurs, permettant la **découverte de services** (service discovery) automatique.

1. **Le Point Commun : Le Réseau labo03-network**

Les deux fichiers Compose utilisent exactement la même déclaration :



• Le mot-clé **external: true** indique à Docker que ce réseau n'est pas créé par le fichier Compose actuel, mais qu'il existe déjà et doit être partagé.

• Chaque service (store\_manager, mysql, redis et supplier\_app) est ensuite explicitement **attaché** à ce réseau, formant ainsi un **LAN virtuel** commun.

1. **Le Mécanisme de Communication : DNS Interne Docker :**

La communication entre les conteneurs repose sur la fonctionnalité de résolution de noms DNS (Domain Name System) intégrée à Docker.

* Résolution du Hostname : Puisque les services sont sur le même réseau labo03-network, un conteneur peut adresser un autre en utilisant son nom de service (ou nom de conteneur, s'il est spécifié).
* Exemple Concret : Le service supplier\_app doit communiquer avec l'API store\_manager. Il utilise le nom de conteneur store-manager-api comme adresse cible dans sa variable d'environnement :



Docker résout automatiquement store-manager-api à l'adresse IP interne du conteneur *store-manager-api*. La communication se fait alors de conteneur à conteneur, sans jamais passer par le système hôte (votre machine) ni nécessiter les ouvertures de ports externes (ports: "5000:5000")

**En résumé**

Les deux *compose* joignent leurs services au même réseau externe

→ Docker fournit la découverte DNS

→ Je peux appeler http://store-manager-api:5000/... depuis supplier\_app sans passer par localhost

# 💡 Particularités de votre configuration CI/CD

### Environnement de déploiement

Pour ce projet, j'ai utilisé une **machine virtuelle (VM) de l'ÉTS**, à laquelle j'accédais à distance via une connexion VPN. Cet environnement a servi de plateforme unique pour le développement, les tests et le déploiement.

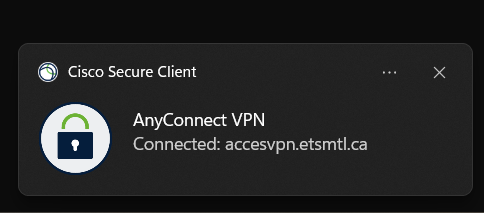
### Méthode de déploiement

Le déploiement a été réalisé en utilisant un **runner auto-hébergé**. J'ai choisi cette approche pour son intégration directe avec GitHub Actions. Contrairement au SSH, qui aurait nécessité des commandes manuelles ou une configuration complexe, le runner auto-hébergé écoute les événements de la pipeline CI/CD et exécute les tâches de déploiement directement sur la VM, ce qui simplifie le processus et évite les problèmes de connectivité réseau.

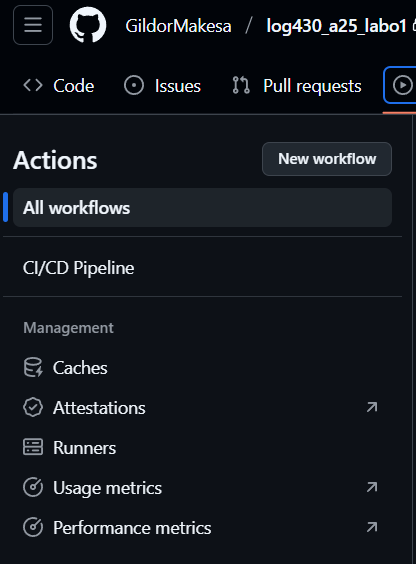
### Étape pour le déploiement

**Étape 1 : Configuration et installation du Runner**

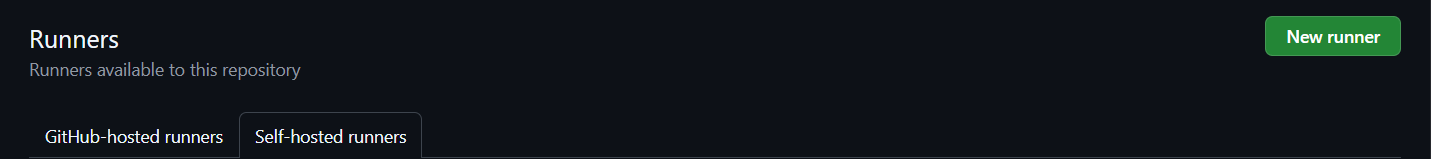
1. Tout d’abord, connectez-vous dans le VPN de l’école :



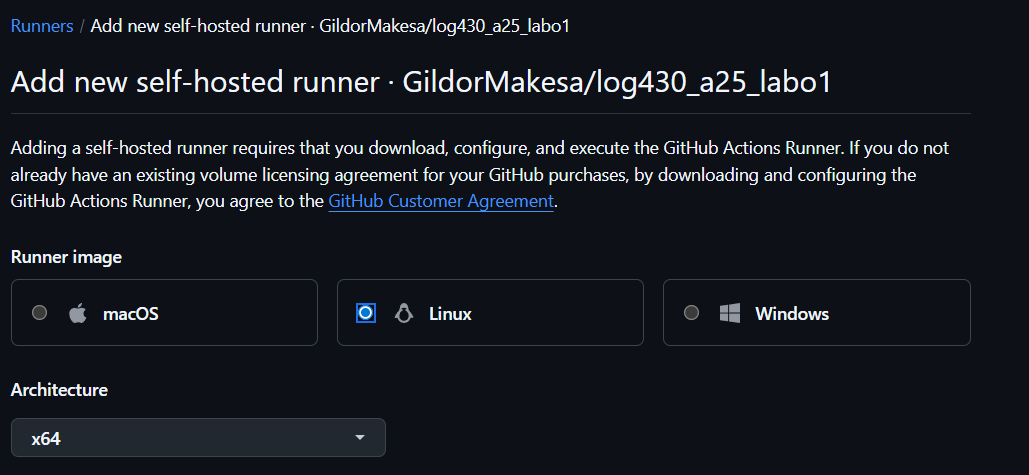
1. **Accédez à votre dépôt GitHub** : Dans le site Github, allez sur [nom d’utilisateur]/log430\_a25\_labo1.
2. **Ouvrez les paramètres des runners** : Cliquez sur l'onglet **Settings** dans le menu supérieur, puis sur **Actions** et enfin sur **Runners**.



1. **Ajoutez un nouveau runner** : Cliquez sur le bouton **New self-hosted runner**.



1. **Choisissez les options** :
   * **OS** : Choisissez Linux.
   * **Architecture** : Choisissez x64 (ou la version correspondant à votre VM).
2. **Suivez les instructions** : GitHub vous affichera une série de commandes. Vous devrez les exécuter dans le terminal de votre VM pour télécharger, configurer et lancer le runner.



**Étape 2 : Exécution des commandes sur votre VM**

Connectez-vous à votre VM en SSH. Exécutez les commandes que GitHub vous a fournies dans la section précédente. Elles ressembleront à ceci :

# 1. Créez un dossier pour le runner

• mkdir actions-runner && cd actions-runner

# 2. Téléchargez le package du runner

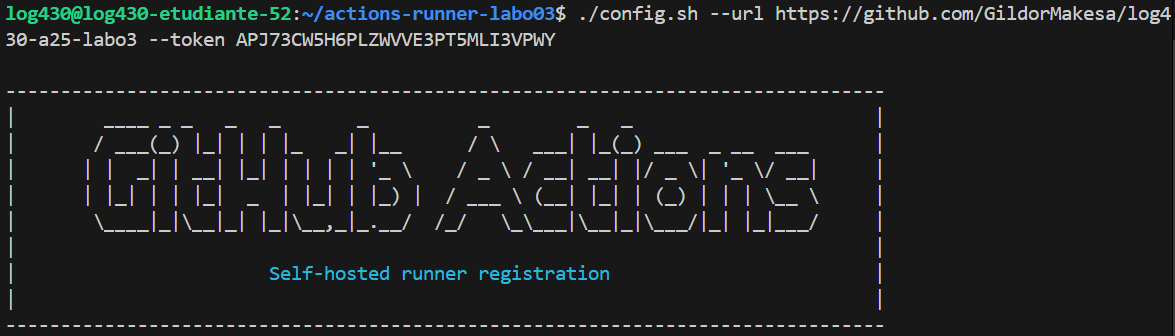
• curl -o actions-runner-linux-x64-2.316.0.tar.gz -L https://github.com/actions/runner/releases/download/v2.316.0/actions-runner-linux-x64-2.316.0.tar.gz

# 3. Extrayez le package

• tar xzf ./actions-runner-linux-x64-2.316.0.tar.gz

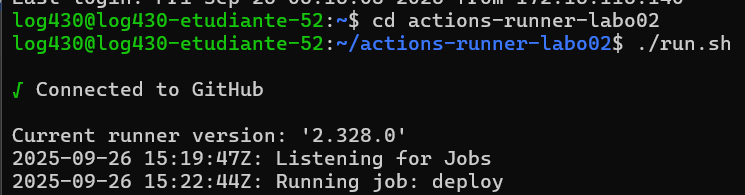
# 4. Exécutez le script de configuration

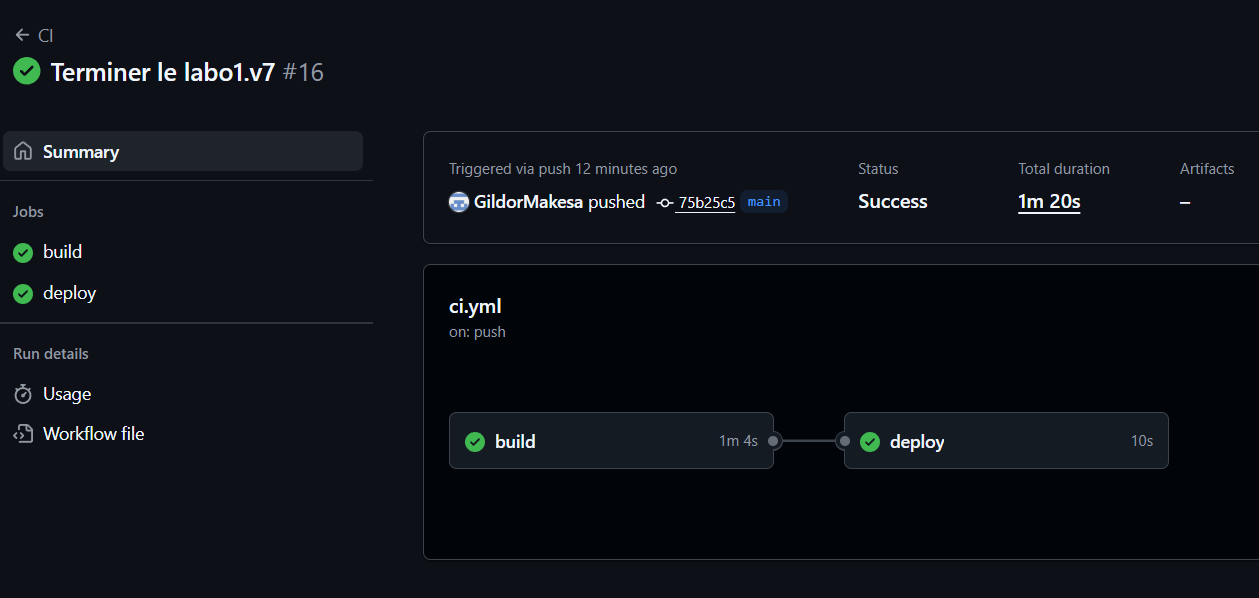
•./config.sh --url https://github.com/GildorMakesa/log430\_a25\_labo1 --token VOTRE\_TOKEN\_UNIQUE



# 5. Lancez le runner pour qu'il écoute les jobs

•./run.sh





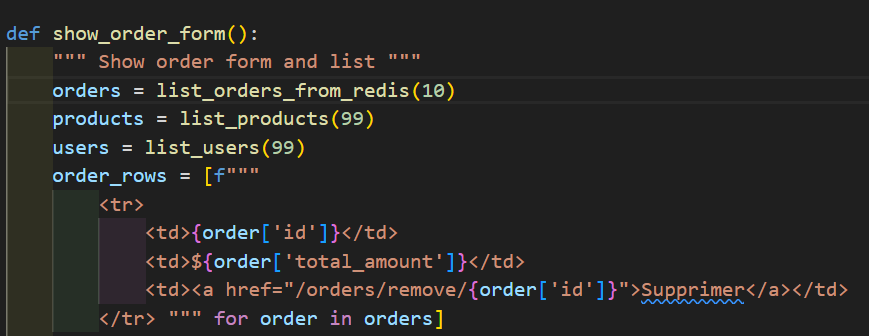
### Défis rencontrés et solutions apportées

Au cours de ce laboratoire, plusieurs difficultés techniques se sont présentées.

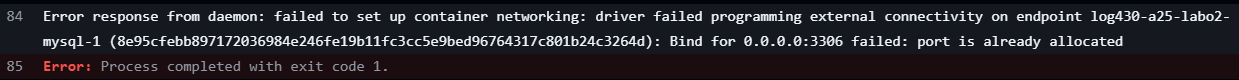
**1. Erreur dans la vue des commandes (order\_view)** : initialement, la vue essayait d’afficher order.id alors que les données venaient sous forme de dictionnaires. Cela entraînait un *AttributeError*.

**Solution**

Par conséquent, je l’ai corrigé en accédant aux valeurs avec order['id'] et order['total\_amount'].



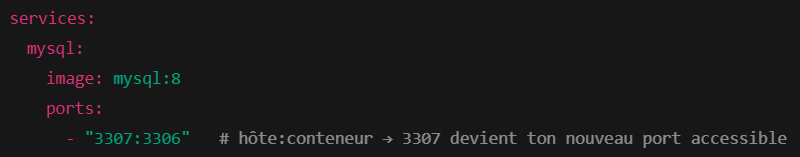
**Port MySQL (3306 déjà utilisé)** : lors de l’exécution avec Docker et GitHub Actions, j’ai rencontré un conflit car le port 3306 était déjà occupé sur la machine hôte.



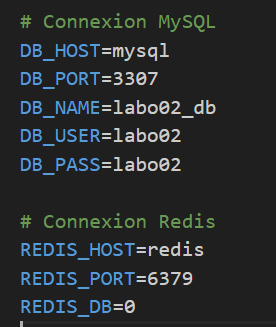
**Solution**

La solution a été de changer le mapping de port vers **3307** dans la configuration Docker/GitHub Action, ce qui a permis d’éviter le conflit et d’exécuter les tests correctement.

• Fichier *Docker-compose.yml (MySQL exposé sur 3307)* :



• Fichier *.env* :



• Fichier ci.yml : Tout les endroit ou il a 3306, on le change par 3307