



Le génie pour l'industrie

Architecture Logicielle (A2025)

Rapport de Laboratoire

Nº du laboratoire	Laboratoire 3
Étudiant(s)	Makesa, Mvuemba Gildor
Cours	LOG430
Session	Aut. 2025
Groupe	LOG430-01-02
Professeur	Fabio Petrillo
Chargé de Laboratoire	Gabriel C. Ullmann
Date	03 oct. 25

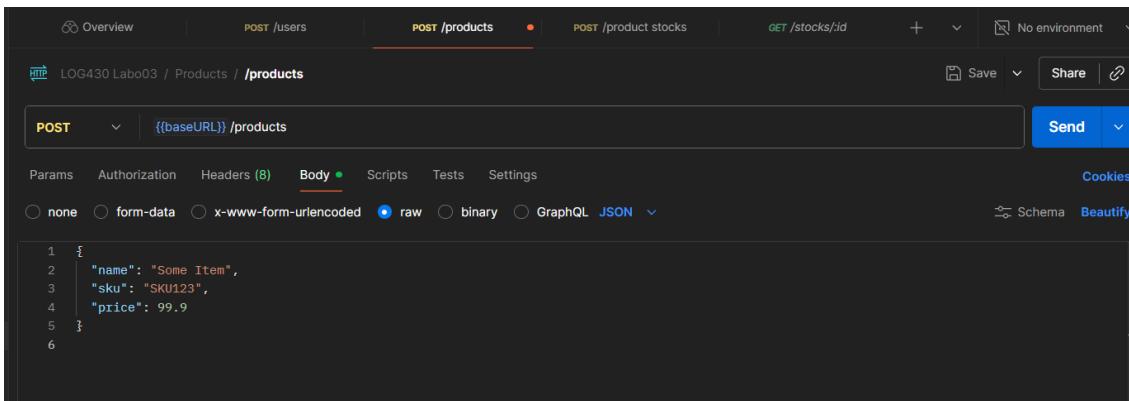
Question 1

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

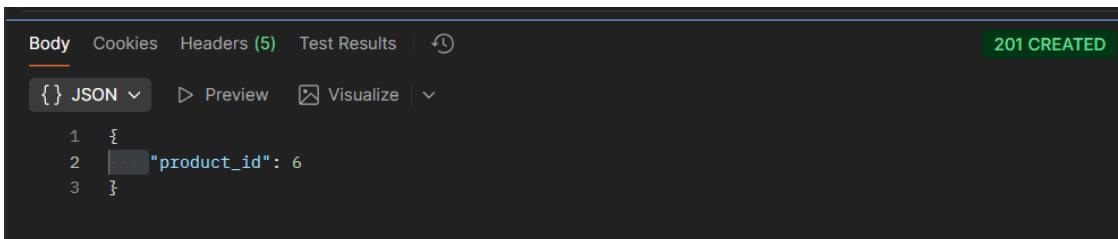
1. Sélectionne la requête **POST {{baseUrl}}/products**.
2. Onglet **Headers** → ajoute Content-Type: application/json.
3. Onglet **Body** → raw + JSON et mets un **produit complet** :



The screenshot shows the Postman interface with a POST request to `/products`. The request URL is `http://LOG430.Labo03/Products/{{baseUrl}}/products`. The Body tab is selected, showing a raw JSON payload:

```
1 {
2   "name": "Some Item",
3   "sku": "SKU123",
4   "price": 99.9
5 }
```

4. Clique **Send**.
5. Réponse attendue (201 Created) :



The screenshot shows the Postman interface after sending the request. The status bar indicates **201 CREATED**. The response body is displayed in JSON format:

```
{ } JSON ▾
```

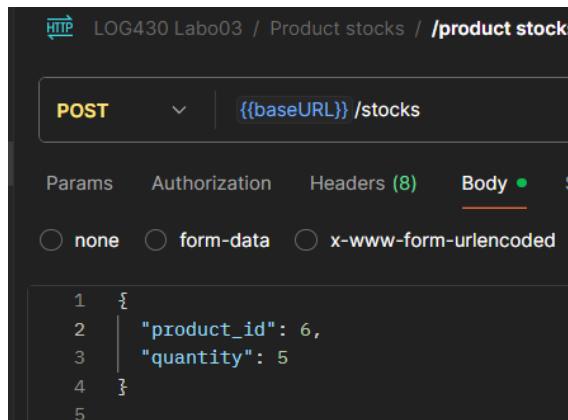
```
1 {
2   "product_id": 6
3 }
```

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

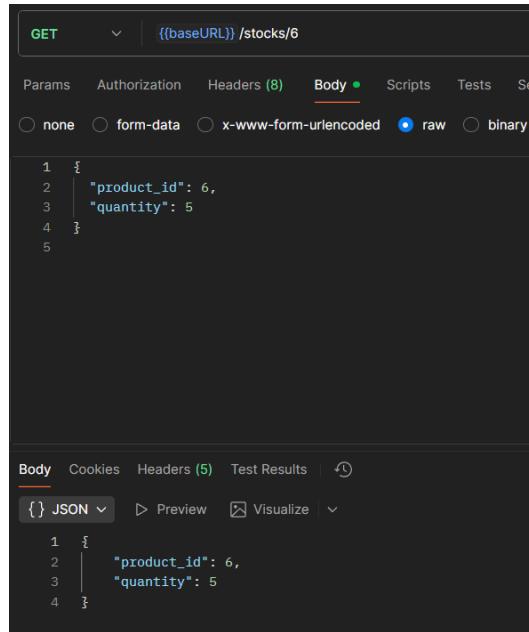
```
# 1. Créez un article (POST /products)
product_data = {"name": "Some Item", "sku": "12345", "price": 99.90}
response = client.post(
    "/products",
    data=json.dumps(product_data),
    content_type="application/json"
)
assert response.status_code == 201
data = response.get_json()
product_id = data["product_id"]
assert product_id > 0
```

B. Ajouter du stock pour ce produit

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



2. Puis vérifie : GET {{baseUrl}}/stocks/6



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

```
# 2. Ajoutez 5 unités au stock de cet article (POST /products_stocks)
stock_data = {"product_id": product_id, "quantity": 5}
response = client.post(
    "/stocks",
    data=json.dumps(stock_data),
    content_type="application/json"
)
```

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

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

The screenshot shows the Postman application interface. At the top, it says "HTTP LOG430 Labo03 / Users / /users". Below that, a "POST" method is selected. The URL is {{baseURL}} /users. The "Body" tab is active, showing the raw JSON payload:

```
1 {  
2   "name": "Jane Doe",  
3   "email": "jd@example.ca"  
4 }
```

Below the body, there are tabs for "Cookies", "Headers (5)", "Test Results", and a preview section. The preview shows the response body:

```
{} JSON ▾ Preview Visualize ▾  
1 {  
2   "user_id": 4  
3 }
```

*Remarque : Prendre note du user_id :4

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

```
# 0. Créez un utilisateur (POST /users)  
user_email = f"test_{uuid4().hex[:8]}@example.com"  
user_data = {"name": "Test User", "email": user_email}  
response = client.post(  
    "/users",  
    data=json.dumps(user_data),  
    content_type="application/json"  
)  
assert response.status_code == 201  
user_id = response.get_json()["user_id"]
```

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

The screenshot shows the Postman application interface. A POST request is being made to the endpoint {{baseUrl}} /orders. The Body tab is selected, showing a raw JSON payload:

```
1 {  
2   "user_id": 4,  
3   "items": [  
4     { "product_id": 6, "quantity": 2 }  
5   ]  
6 }  
7
```

The response tab shows the returned JSON data:

```
1 {  
2   "order_id": 3  
3 }
```

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

```
# 4. Créez une commande de 2 unités (POST /orders)  
order_data = {  
    "user_id": 1, # Assurez-vous qu'il existe un utilisateur avec ID=1  
    "items": [{"product_id": product_id, "quantity": 2}]  
}  
response = client.post(  
    "/orders",  
    data=json.dumps(order_data),  
    content_type="application/json"  
)  
assert response.status_code == 201  
order_id = response.get_json()["order_id"]
```

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

1. Relire le stock {{baseUrl}}/stocks/{{product_id}}

The screenshot shows the Postman interface with the following details:

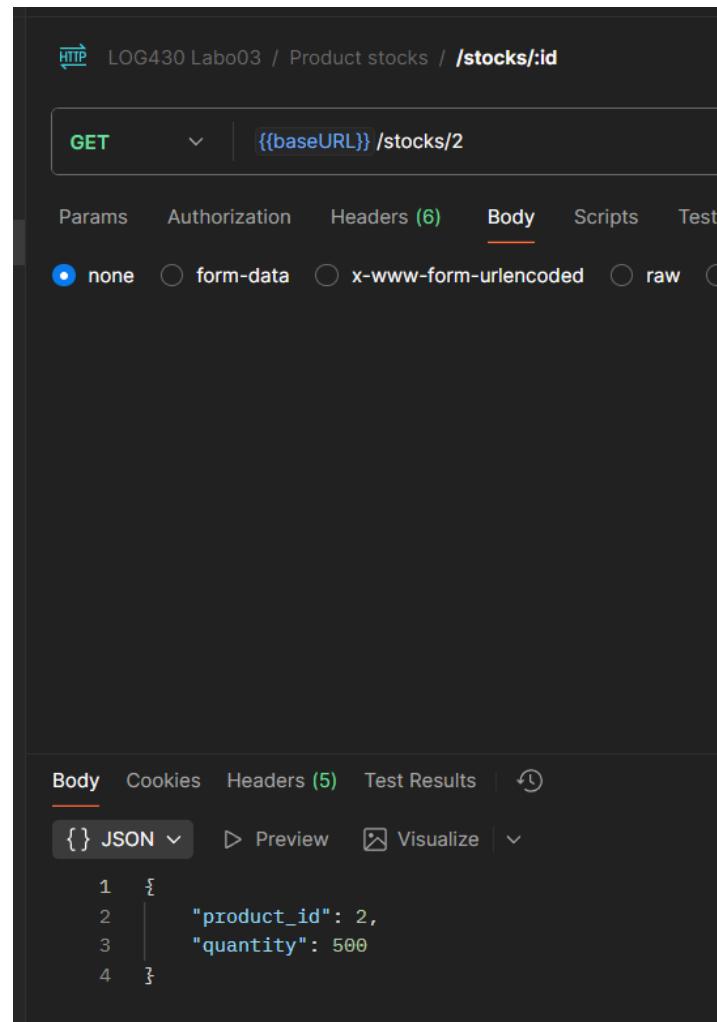
- Method: GET
- URL: {{baseUrl}} /stocks/6
- Body tab selected, showing "none" selected.
- Headers tab (6) selected.
- Test Results tab (5) selected, showing a green "201 CREATED" status.
- Body panel showing JSON response:

```
1  {
2   "product_id": 6,
3   "quantity": 3
4 }
```

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

```
# 5. Vérifiez le stock encore une fois → doit être 3
response = client.get(f"/stocks/{product_id}")
assert response.status_code == 201
stock_info = response.get_json()
print("Stock final:", stock_info["quantity"])           # <-- visible avec pytest -s
assert stock_info["quantity"] == 3
```

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



The screenshot shows a Postman interface with the following details:

- Method: GET
- URL: {{baseURL}}/stocks/2
- Body tab selected, showing "none" selected.
- Body content:

```
{ } JSON
```

```
1 {  
2     "product_id": 2,  
3     "quantity": 500  
4 }
```

💡 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 réfère 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.

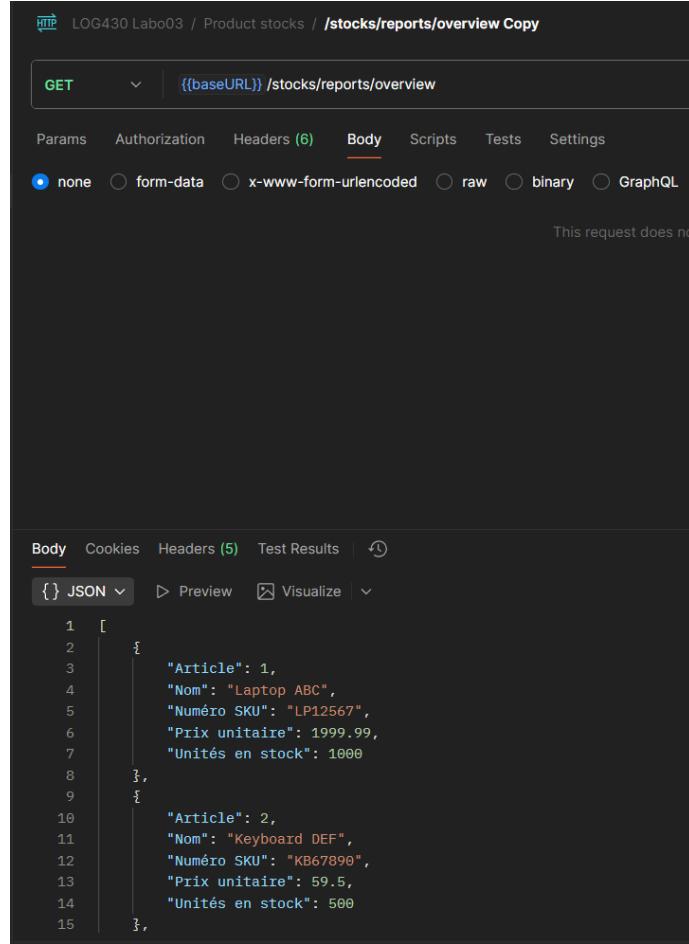
```
def get_stock_for_all_products():
    session = get_sqlalchemy_session()
    try:
        rows = [
            session.query(
                Product.id.label("product_id"),
                Product.name,
                Product.sku,
                Product.price,
                func.coalesce(func.sum(Stock.quantity), 0).label("quantity"),
            )
            # Simple join to include only products with stock entries
            .join(Product.stocks)
            .group_by(Product.id, Product.name, Product.sku, Product.price)
            .order_by(Product.id.asc())
            .all()
        ]
        return [
            {
                "Article": r.product_id,
                "Numéro SKU": r.sku,
                "Prix unitaire": float(r.price),
                "Unités en stock": int(r.quantity),
            }
            for r in rows
        ]
    finally:
        session.close()
```

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.

```
def get_stock_for_all_products():
    session = get_sqlalchemy_session()
    try:
        rows = (
            session.query(
                Product.id.label("product_id"),
                Product.name,
                Product.sku,
                Product.price,
                Stock.quantity
            )
            .join(Stock, Stock.product_id == Product.id)  # inner join
            .order_by(Product.id.asc())
            .all()
        )
        return [
            {
                "Article ID": row.product_id,
                "Nom de l'article": row.name,
                "Numéro SKU": row.sku,
                "Prix unitaire": float(row.price),
                "Unités en stock": int(row.quantity),
            }
            for row in rows
        ]
    finally:
        session.close()
```



The screenshot shows a Postman interface with a dark theme. At the top, it says "HTTP LOG430 Labo03 / Product stocks / /stocks/reports/overview Copy". Below that, there's a "GET" method selected. The "Body" tab is active, showing the following JSON response:

```
[{"Article": 1, "Nom": "Laptop ABC", "Numéro SKU": "LP12567", "Prix unitaire": 1999.99, "Unités en stock": 1000}, {"Article": 2, "Nom": "Keyboard DEF", "Numéro SKU": "KB67890", "Prix unitaire": 59.5, "Unités en stock": 500}]
```

💡 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.

Illustration de la réponse

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

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

The screenshot shows a GraphQL playground interface. At the top, it says "POST" and the URL is "{{baseUrl}} /stocks/graphql". Below that, there are tabs for "Params", "Authorization", "Headers (8)", "Body" (which is selected), "Scripts", "Tests", and "Se". Under "Body", there are options for "none", "form-data", "x-www-form-urlencoded", "raw", and "binary". The "QUERY" section contains the following GraphQL code:

```

1 {
2   product(id: 1) {
3     id
4     name
5     sku
6     quantity
7   }
8 }
9

```

The "Body" section shows the JSON response:

```

1 {
2   "data": {
3     "product": {
4       "id": 1,
5       "name": "Laptop ABC",
6       "quantity": 0,
7       "sku": "LP12567"
8     }
9   },
10   "errors": null
11 }

```

💡 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.

```
session = Session()
```

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.

```
key = f"stock:{product_id}"

current_stock = r.hget(f"stock:{product_id}", "quantity")
current_stock = int(current_stock) if current_stock else 0

if operation == '+':
    new_quantity = current_stock + quantity
else:
    new_quantity = current_stock - quantity

if new_quantity < 0:
    new_quantity = 0

pipeline.hset(f"stock:{product_id}", "quantity", new_quantity)
```

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.

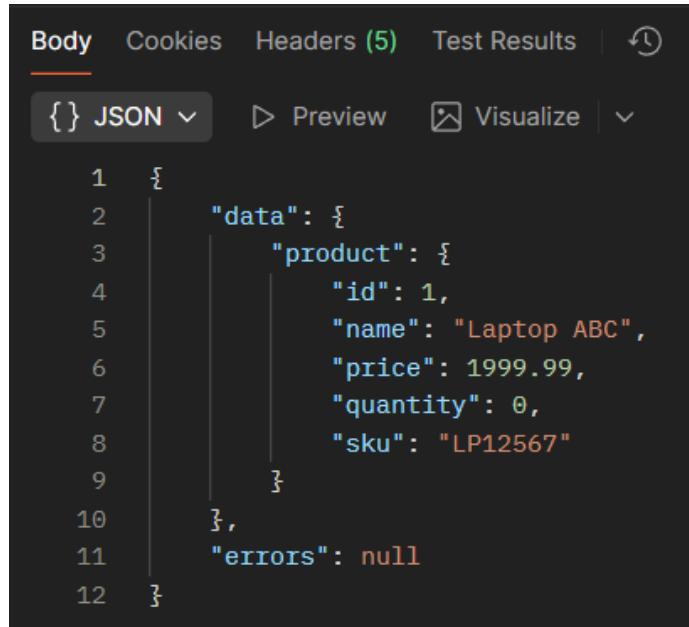
```
    need_name = not r.hexists(key, "name")
    need_sku = not r.hexists(key, "sku")
    need_price = not r.hexists(key, "price")

    if need_name or need_sku or need_price:
        p = session.query(ProductModel).filter(ProductModel.id == product_id).first()
        if p:
            mapping = {}
            if need_name:
                mapping["name"] = p.name
            if need_sku:
                mapping["sku"] = p.sku
            if need_price:
                mapping["price"] = float(p.price or 0.0)
            if mapping:
                pipeline.hset(key, mapping=mapping)

    pipeline.execute()
```

💡 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.



The screenshot shows the Postman interface with the 'Body' tab selected. The response body is a JSON object with the following structure:

```
{ } JSON ▾
```

```
1  {
2   "data": {
3     "product": {
4       "id": 1,
5       "name": "Laptop ABC",
6       "price": 1999.99,
7       "quantity": 0,
8       "sku": "LP12567"
9     }
10   },
11   "errors": null
12 }
```

The 'Headers' tab shows five items: 'Content-Type: application/json', 'Accept: application/json', 'User-Agent: PostmanRuntime/7.29.0', 'Host: localhost:4000', and 'Connection: keep-alive'. The 'Test Results' tab is also visible.

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 :

```
networks:  
  labo03-network:  
    driver: bridge  
    external: true
```

- 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.

2. 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 :

```
supplier_app:
  build: .
  environment:
    - PYTHONUNBUFFERED=1
    - ENDPOINT_URL=http://store-manager-api:5000/stocks/graphql
  volumes:
    - .:/app
  networks:
    - labo03-network
```

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/...](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.

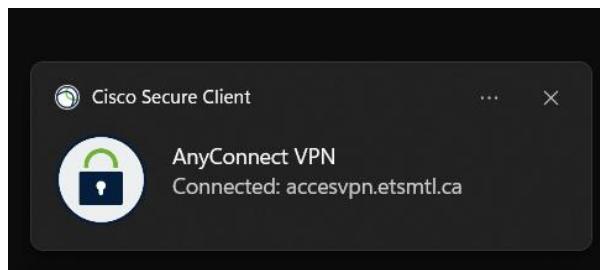
Méthode de déploiement

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.

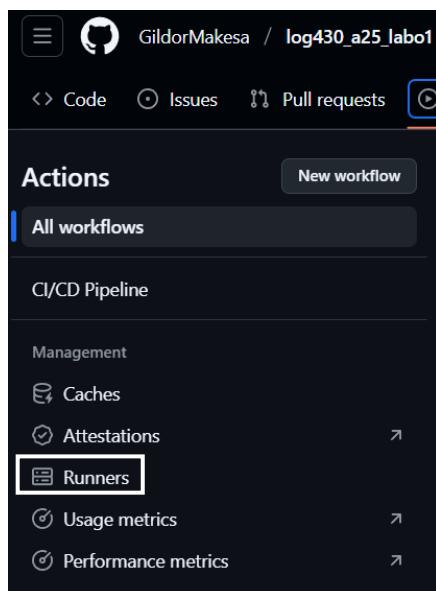
Étape pour le déploiement

Étape 1 : Configuration et installation du Runner

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



2. **Accédez à votre dépôt GitHub** : Dans le site Github, allez sur [nom d'utilisateur]/log430_a25_lab01.
3. **Ouvrez les paramètres des runners** : Cliquez sur l'onglet **Settings** dans le menu supérieur, puis sur **Actions** et enfin sur **Runners**.



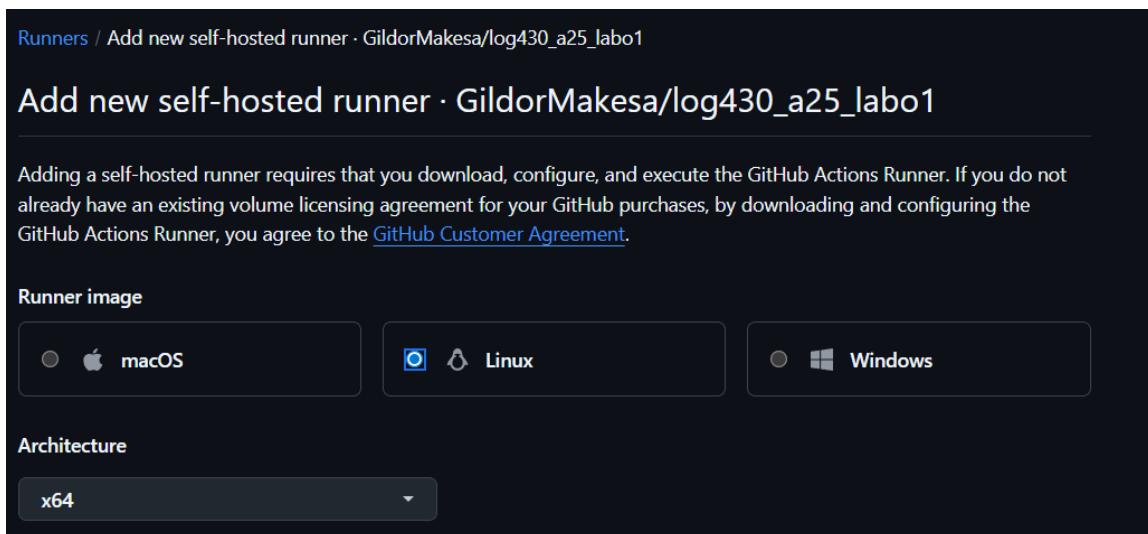
4. Ajoutez un nouveau runner : Cliquez sur le bouton **New self-hosted runner**.



5. Choisissez les options :

- **OS** : Choisissez Linux.
- **Architecture** : Choisissez x64 (ou la version correspondant à votre VM).

6. 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_lab01 --token
VOTRE_TOKEN_UNIQUE
```

```
log430@log430-estudante-52:~/actions-runner-labo03$./config.sh --url https://github.com/GildorMakesa/log430-a25-labo3 --token APJ73CW5H6PLZWVVE3PT5MLI3WPVY
```

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

- ./run.sh

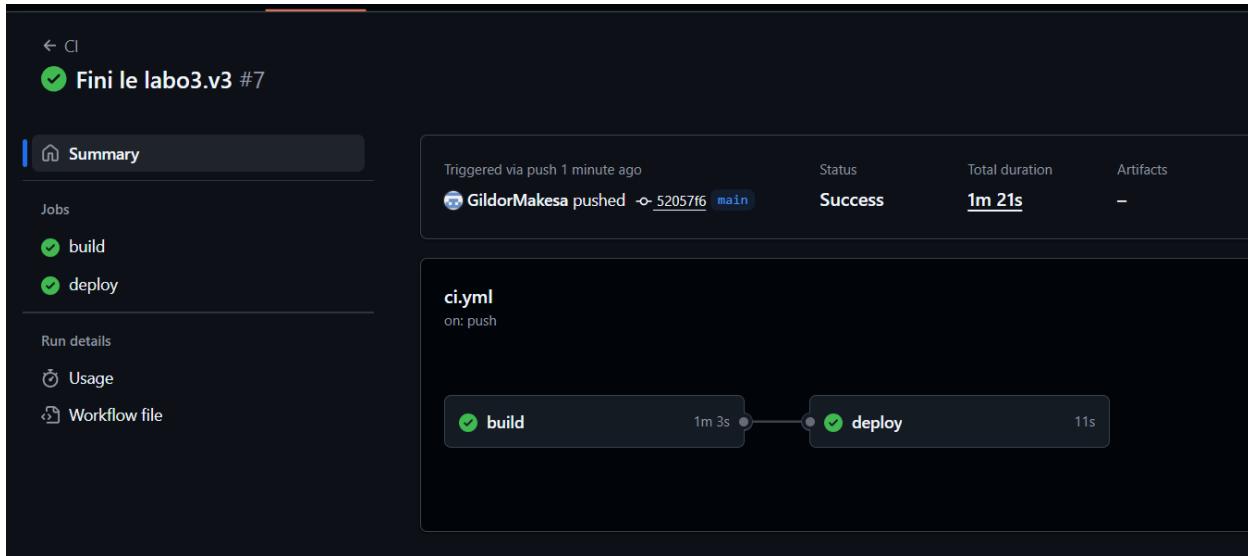
```
log430@log430-etudiante-52:~/actions-runner-labo03$./run.sh
```

✓ Connected to GitHub

Current runner version: '2.328.0'

2025-10-04 02:35:18Z: Listening for Jobs

2025-10-04 02:35:20Z: Running job: deploy



## Défis rencontrés et solutions apportées

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

### 1. Erreur d'Unicité lors de l'Insertion d'Utilisateur

La validation (L'assertion `response.status_code == 201`) dans mes tests échoue, retournant un statut **500 Internal Server Error**. Les journaux MySQL révèlent une erreur :

**Duplicate entry 'test\_user@example.com' for key 'users.email'**

#### Cause:

Une contrainte d'unicité est appliquée à la colonne `email` de la table `users`. L'exécution répétée du même test tentait d'insérer un utilisateur avec une adresse e-mail déjà présente en base de données, violant cette contrainte.

#### Solution

Pour garantir l'**isolation des tests** et l'unicité des données, l'e-mail de l'utilisateur est désormais généré dynamiquement à chaque exécution du test. L'approche choisie est d'utiliser un identifiant unique (UUID) pour composer l'e-mail.

```

0. Créez un utilisateur (POST /users)
user_email = f"test_{uuid4().hex[:8]}@example.com"
user_data = {"name": "Test User", "email": user_email}
response = client.post(
 "/users",
 data=json.dumps(user_data),
 content_type="application/json"
)
assert response.status_code == 201
user_id = response.get_json()["user_id"]

```

## **2. Conflit d'Allocation des Ports Hôtes**

Lorsque je faisais mon **CI/CD** dans Github, j'avais un problème de port d'allocation. En effet, le démarrage d'un second environnement Docker Compose échoue avec le message : Bind for 0.0.0.0:3306 failed: port is already allocated (et un message similaire pour le port 6379).

```

81 Error response from daemon: failed to set up container networking: driver failed programming external connectivity on endpoint log4j0-a25-labo3-
mysql-1 (23aa928d04f1dac23b3f70c090809e1b6414f5a67f3364905972268327486a7d): Bind for :::3306 failed: port is already allocated
82 Error: Process completed with exit code 1.

```

### **Solution**

Les services se parlent via le réseau Docker, donc pas besoin d'un bind sur l'hôte. Dans ton docker-compose.yml (côté serveur), **supprime** la section ports de Redis /MySQL

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

```

▷ Run Service
redis:
 image: redis:7
 restart: unless-stopped
 networks:
 - labo03-network
 # ports:
 # - "6379:6379"

```

```

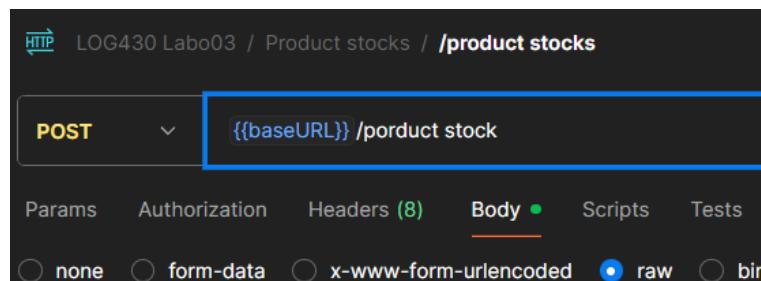
▷ Run Service
mysql:
 image: mysql:8
 restart: unless-stopped
 environment:
 MYSQL_ROOT_PASSWORD: root
 MYSQL_DATABASE: labo03_db
 MYSQL_USER: labo03
 MYSQL_PASSWORD: labo03
 networks:
 - labo03-network
ports:
- "3306:3306"

```

Par conséquent, supprimer l'exposition de ports était une bonne solution en production CI/CD, car les conteneurs communiqueront via le **réseau Docker** (labo03-network) sans conflit de port.

### 3. Erreur de Chemin (Endpoint) pour la Gestion des Stocks

Tentative de requête sur un chemin incorrect (/product stocks ou variations) qui résulte en une réponse **405 Method Not Allowed** ou **404 Not Found**. En effet, j'étais mélangé, car dans le fichier *store\_manager.py*, on voit que la route est /product stocks, mais cela ne fonctionnait pas lorsque je le mettais dans Postman.



#### Cause

L'API utilise une ressource unique (/stocks) pour faire une requête POST, et non une URL plus descriptive ou paramétrée pour la création.

#### Solution

La route correcte pour créer ou ajuster le stock (en utilisant la méthode POST avec le corps de la requête) est : POST {{baseUrl}}/stocks

The screenshot shows a POST request in Postman. The URL is {{baseUrl}}/stocks/2. The Headers tab shows 8 items. The Body tab is selected and contains the following JSON:

```
1 {
2 "product_id": 6,
3 "quantity": 5
4 }
```

```
name: CI

on:
 push:
 branches: [main, master]
 pull_request:

jobs:
 build:
 runs-on: ubuntu-latest
```

```

services:
 mysql:
 image: mysql:8
 env:
 MYSQL_ROOT_PASSWORD: root
 MYSQL_DATABASE: labo03_db
 MYSQL_USER: labo03
 MYSQL_PASSWORD: labo03
 ports:
 - 3306:3306
 options: >-
 --health-cmd "mysqladmin ping -h 127.0.0.1 -ulabo03 -plabo03"
 --health-interval 10s
 --health-timeout 5s
 --health-retries 10

 redis:
 image: redis:7
 ports:
 - 6379:6379
 options: >-
 --health-cmd "redis-cli ping || exit 1"
 --health-interval 10s
 --health-timeout 5s
 --health-retries 10

steps:
 - uses: actions/checkout@v4

 - name: Set up Python
 uses: actions/setup-python@v4
 with:
 python-version: '3.11'

 - name: Install deps
 run: |
 python -m pip install --upgrade pip
 pip install -r requirements.txt
 pip install pytest

 - name: Create .env for tests
 run: |
 cat > .env << 'EOF'
 DB_HOST=127.0.0.1

```

```

DB_PORT=3306
DB_NAME=labo03_db
DB_USER=labo03
DB_PASS=labo03
REDIS_HOST=127.0.0.1
REDIS_PORT=6379
REDIS_DB=0
EOF

- name: Install mysql-client
 run: sudo apt-get update && sudo apt-get install -y mysql-client

- name: Wait for MySQL
 run: |
 for i in {1..40}; do
 if mysql -h127.0.0.1 -P3306 -ulabo03 -plabo03 -e "SELECT 1"
labo03_db; then
 echo "MySQL is up!"
 break
 fi
 echo "MySQL not ready yet... ($i)"
 sleep 3
 done

- name: Init DB schema
 run: |
 mysql -h127.0.0.1 -P3306 -ulabo03 -plabo03 labo03_db < db-init/init.sql

- name: Run tests
 working-directory: src
 run: python -m pytest -q

deploy:
 runs-on: self-hosted
 needs: build
 steps:
 # (Optionnel mais conseillé) reprendre la main sur le workspace côté runner
 - name: Fix workspace permissions
 run: |
 if [-n "$GITHUB_WORKSPACE"]; then
 sudo chown -R "$USER":"$USER" "$GITHUB_WORKSPACE" || true
 find "$GITHUB_WORKSPACE" -type d -name "__pycache__" -prune -exec rm -rf {} + || true
 fi

```

```
- uses: actions/checkout@v4

Si ton .env de prod est géré par secrets, recrée-le ici
- name: Create .env (prod)
run: |
cat > .env << 'EOF'
DB_HOST=mysql
DB_PORT=3306
DB_NAME=labo03_db
DB_USER=labo03
DB_PASS=${{ secrets.DB_PASS }}
REDIS_HOST=redis
REDIS_PORT=6379
REDIS_DB=0
EOF

- name: Deploy with Docker Compose
 run: |
 echo "🚀 Déploiement..."
 docker network create labo03-network || true
 docker compose pull || true
 docker compose down || true
 docker compose up -d --build
 docker compose ps
 docker compose down -v
```