Laboratoire 3 - LOG430

Gestion de magasin avec API Flask et GraphQL

Étudiant : [Votre nom]
Date : 2 octobre 2025
Cours : LOG430 - Architecture logicielle

Question 1 : Test du flux de stock

Question : Quel nombre d'unités de stock pour votre article avez-vous obtenu à la fin du test ? Et pour l'article avec id=2 ?

Réponse :

Stock final pour l'article créé (ID 5): 5 unités Stock pour l'article ID=2: 500 unités

Sortie du test pytest :

```
$ cd src
$ python -m pytest tests/test_store_manager.py::test_stock_flow -v -s

============= test session starts ==============
platform win32 -- Python 3.13.7, pytest-8.4.2, pluggy-1.6.0
collected 1 item

tests/test_store_manager.py::test_stock_flow
Produit créé avec ID: 5
5 unités ajoutées au stock
Stock initial vérifié: 5 unités
Utilisateur créé avec ID: 4
Commande créée avec ID: 1 (2 unités commandées)
Stock après commande: 3 unités
Commande 1 supprimée
Stock après suppression de la commande: 5 unités
Stock de l'article ID=2: 500 unités
=== RÉPONSE À LA QUESTION 1 ===
Stock final pour l'article créé (ID 5): 5 unités
Stock pour l'article ID=2: 500 unités
PASSED

========= 1 passed, 1 warning in 1.05s =========
```

Question 2 : Utilisation de la méthode JOIN

Question : Décrivez l'utilisation de la méthode join dans ce cas.

Réponse :

La méthode JOIN a été implémentée dans le fichier `src/stocks/queries/read_stock.py` pour optimiser la récupération des données de stock. Au lieu de faire des requêtes séparées pour obtenir les informations des produits et des stocks, le JOIN permet de combiner les deux tables en une seule requête SQL.Le JOIN effectue une liaison entre la table `stocks` et la table `products` en utilisant la clé étrangère `product_id`, permettant d'enrichir les informations de stock avec les détails complets du produit.

Résultats obtenus via Postman :

- **Méthode :** GET
- **URL :** `http://localhost:5000/stocks/reports/overview-stocks`


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


Question 3 : Endpoint GraphQL de base

Question : Quels résultats avez-vous obtenus en utilisant l'endpoint POST /stocks/graphql-query avec la requête suggérée ?

Réponse :

Requête GraphQL testée :

```
{
  product(id: "1") {
    id
    quantity
  }
}
```

Configuration Postman :

- **Méthode :** POST
- **URL :** `http://localhost:5000/stocks/graphql-query`

- **Headers :** `Content-Type: application/json`
- **Body :**

```
{
  "query": "{ product(id: \"1\") { id quantity } }"
}
```

Résultat obtenu :

```
{
  "data": {
    "product": {
      "id": 1,
      "quantity": 1000
    }
  },
  "errors": null
}
```

Question 4 : Modifications dans update_stock_redis

Question : Quelles lignes avez-vous changé dans update_stock_redis ?

Réponse :

Fichier modifié : `src/stocks/commands/write_stock.py`

Modifications dans la fonction `_populate_redis_from_mysql()` :

Code :

```
def _populate_redis_from_mysql(redis_conn):
    session = get_sqlalchemy_session()
    try:
        # JOIN avec la table products pour récupérer name, sku, price
        stocks = session.execute(
            text("""
                SELECT s.product_id, s.quantity, p.name, p.sku, p.price
                FROM stocks s
                JOIN products p ON s.product_id = p.id
            """)
        ).fetchall()

        for product_id, quantity, name, sku, price in stocks:
            pipeline.hset(
                f"stock:{product_id}",
                mapping={
                    "quantity": quantity,
```

```
                    "name": name,           # ← AJOUTÉ
                    "sku": sku,             # ← AJOUTÉ
                    "price": float(price)   # ← AJOUTÉ
                }
            )
```

Question 5 : Résultats avec améliorations GraphQL

Question : Quels résultats avez-vous obtenus en utilisant l'endpoint POST /stocks/graphql-query avec les améliorations ?

Réponse :

Requête GraphQL améliorée :

```
{
  product(id: "1") {
    id
    name
    sku
    price
    quantity
  }
}
```

Résultat obtenu avec Postman :

```
{
    "data": {
        "product": {
            "id": 1,
            "name": "Laptop ABC",
            "price": 1999.99,
            "quantity": 1000,
            "sku": "LP12567"
        }
    },
    "errors": null
}
```

Question 6 : Communication entre conteneurs

Question : Examinez attentivement les fichiers docker-compose.yml. Qu'ont-ils en commun ? Par quel mécanisme ces conteneurs peuvent-ils communiquer entre eux ?

Réponse :

Les conteneurs Docker peuvent communiquer entre eux grâce à un **réseau Docker partagé**. Dans ce projet, tous les services utilisent le même réseau externe nommé `labo03-network` avec le driver `bridge`. Ce réseau agit comme un pont virtuel permettant la communication inter-conteneurs.

Points communs identifiés :

1. **Réseau partagé** : Tous les services sont connectés au même réseau `labo03-network`
2. **Driver bridge** : Utilisation du même type de réseau pour tous les conteneurs
3. **Configuration externe** : Le réseau est défini comme externe (`external: true`) et partagé
4. **Découverte DNS automatique** : Les conteneurs peuvent se référencer par leur nom de service

Illustration avec les fichiers docker-compose.yml :

Fichier principal (`/docker-compose.yml`) :

```
services:
  store_manager:
    networks:
      - labo03-network
  mysql:
    networks:
      - labo03-network
  redis:
    networks:
      - labo03-network

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

Fichier scripts (`/scripts/docker-compose.yml`) :

```
services:
  supplier_app:
    networks:
      - labo03-network

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