

**LOG430 - Architecture logicielle**  
**Rapport de laboratoire**  
**Labo 01 – Rapport**

**JOLAN THOMASSIN**

**École de technologie supérieure**

03/10/2025



**ÉCOLE DE  
TECHNOLOGIE  
SUPÉRIEURE**

Université du Québec

# QUESTION 1

## Créez un article (POST /products)

The screenshot shows a REST client interface for a project named 'LOG430 Labo03'. The active endpoint is 'Products /products'. The request method is 'POST' and the URL is '[[baseURL]] /products'. The request body is a JSON object: 

```
{  "name": "Chaise",  "sku": "CHA-001",  "price": 49.99}
```

. The response status is '201 CREATED' with a response time of 21 ms and a size of 189 B. The response body is a JSON object: 

```
{  "product_id": 19}
```

.

LOG430 Labo03 / Products / /products

POST

Params Authorization Headers (8) **Body** Scripts Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON

```
1 {
2   "name": "Chaise",
3   "sku": "CHA-001",
4   "price": 49.99
5 }
```

Body Cookies Headers (5) Test Results ☒ 201 CREATED • 21 ms • 189 B

☒ JSON

```
1 {
2   "product_id": 19
3 }
```

## Ajoutez 5 unités au stock de cet article (POST /stocks)

The screenshot shows the same REST client interface. The active endpoint is 'STOCKS /stocks'. The request method is 'POST' and the URL is '[[baseURL]] /stocks'. The request body is a JSON object: 

```
{  "product_id": 19,  "quantity": 5}
```

. The response status is '201 CREATED' with a response time of 26 ms and a size of 199 B. The response body is a JSON object: 

```
{  "result": "rows added: 19"}
```

.

LOG430 Labo03 / STOCKS / /stocks

POST

Params Authorization Headers (8) **Body** Scripts Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON

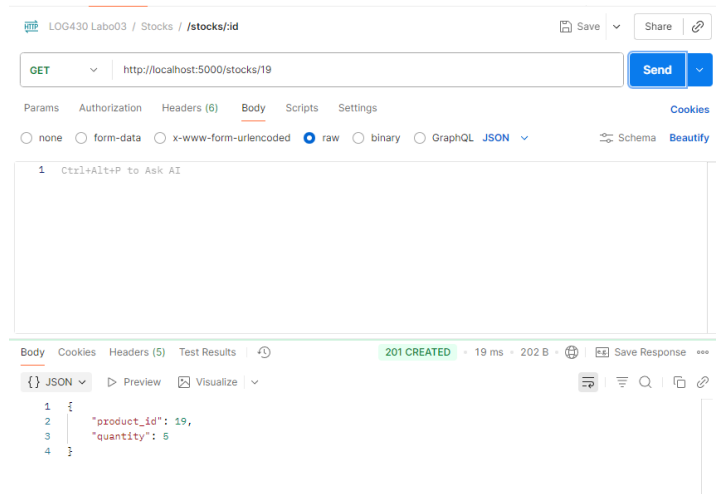
```
1 {
2   "product_id": 19,
3   "quantity": 5
4 }
```

Body Cookies Headers (5) Test Results ☒ 201 CREATED • 26 ms • 199 B

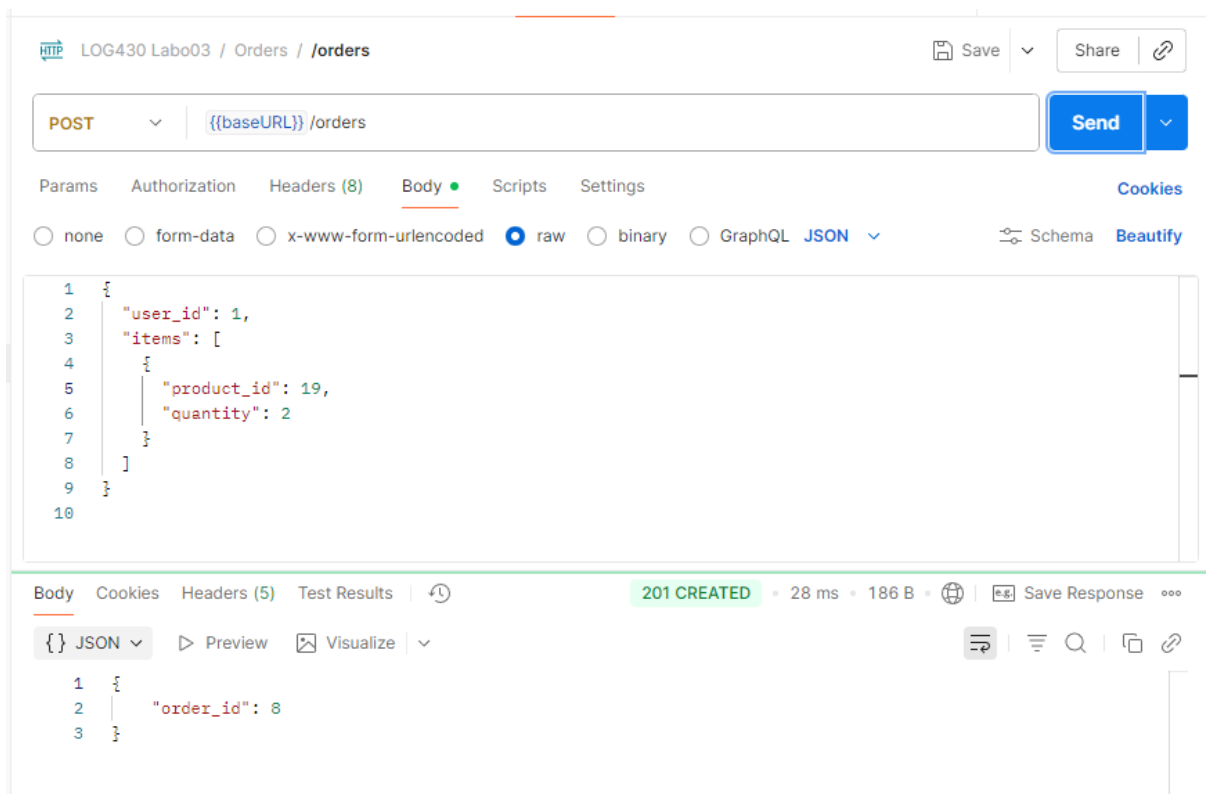
☒ JSON

```
1 {
2   "result": "rows added: 19"
3 }
```

Vérifiez le stock, votre article devra avoir 5 unités dans le stock (GET /stocks/:id)



Faites une commande de 2 unités de l'article que vous avez créé (POST /orders)



## Vérifiez le stock encore une fois (GET /stocks/:id)

LOG430 Labo03 / Stocks / /stocks/:id

GET  Send

Params Authorization Headers (6) **Body** Scripts Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON ☐ Schema Beautify

1 Ctrl+Alt+P to Ask AI

Body Cookies Headers (5) Test Results 201 CREATED • 19 ms • 202 B Save Response

{ } JSON Preview Visualize

```
1 {
2   "product_id": 19,
3   "quantity": 3
4 }
```

**Étape extra:** supprimez la commande et vérifiez le stock de nouveau.  
Le stock devrait augmenter après la suppression de la commande.

The screenshot shows a REST client interface for a project named 'LOG430 Labo03'. The active tab is 'Orders' with the path '/orders'. A 'DELETE' request is configured to the endpoint '({baseUrl})/orders/8'. The response status is '200 OK' with a response time of 33 ms and a size of 183 B. The response body is a JSON object: 

```
{  "deleted": true}
```

The screenshot shows the same REST client interface, now with the 'Stocks' tab selected and the path '/stocks/19'. A 'GET' request is configured to the endpoint 'http://localhost:5000/stocks/19'. The response status is '201 CREATED' with a response time of 17 ms and a size of 202 B. The response body is a JSON object: 

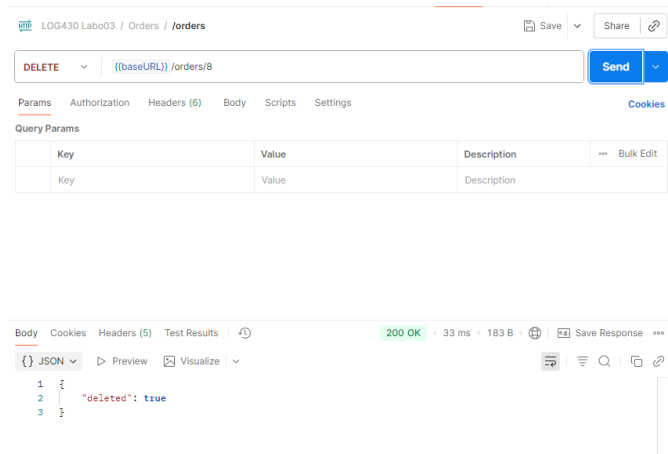
```
{  "product_id": 19,  "quantity": 5}
```

À la fin du test, mon produit que j'ai créé possède 5 unités en stock. En effet, après avoir ajouté 5 unités, passé une commande de 2 unités puis supprimé la commande, le stock est revenu à 5.

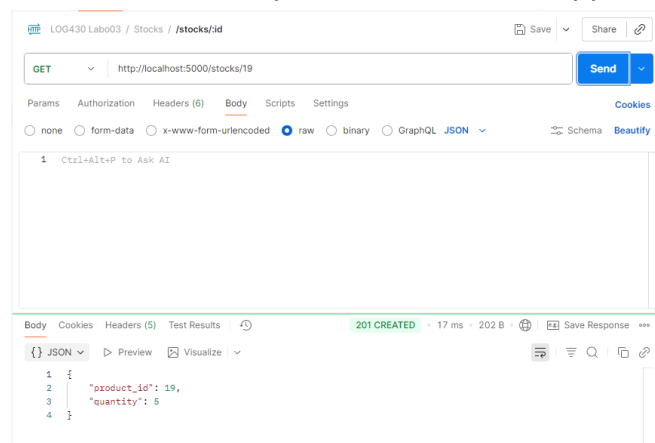
Pour l'article existant avec id = 2, son stock est de 500 unités.

## Captures d'écran Postman :

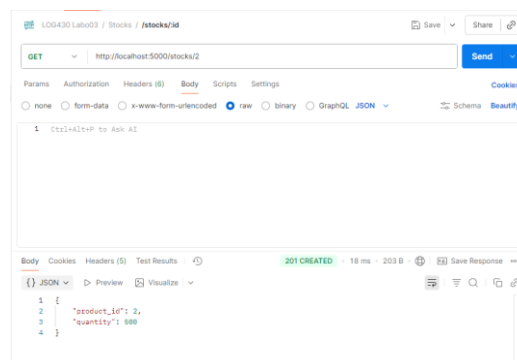
### Création du produit et ajout de stock



### Vérification du stock après commande et suppression



### Vérification du stock de l'article id=2



## QUESTION 2

Dans cette tâche, nous avons modifié la méthode `get_stock_for_all_products` afin d'enrichir le rapport de stock avec des informations supplémentaires provenant de la table `Product` (nom, SKU, prix).

L'utilisation de `join` dans `SQLAlchemy` permet d'associer deux tables sur une clé commune. Ici, la clé est `Stock.product_id`, qui fait référence à `Product.id`.

Cela correspond, en SQL, à une requête du type :

```
SELECT stock.product_id, stock.quantity, product.name, product.sku, product.price
FROM stock
JOIN product ON stock.product_id = product.id
ORDER BY stock.product_id;
```

Avec `SQLAlchemy`, on a écrit l'équivalent Python suivant :

```
def get_stock_for_all_products():
    """Get stock quantity for all products, including product info via JOIN"""
    session = get_sqlalchemy_session()

    rows = (
        session.query(
            Stock.product_id,
            Stock.quantity,
            Product.name,
            Product.sku,
            Product.price,
        )
        .join(Product, Stock.product_id == Product.id) # JOIN Stock / Product
        .order_by(Stock.product_id)
        .all()
```

```
)
```

```
data = []
```

```
for product_id, quantity, name, sku, price in rows:
```

```
    data.append({
```

```
        "Article": int(product_id),
```

```
        "Nom": name,
```

```
        "Numéro SKU": sku,
```

```
        "Prix unitaire": float(price) if price is not None else None,
```

```
        "Unités en stock": int(quantity),
```

```
    })
```

```
return data
```

## Explication

- La méthode `.join(Product, Stock.product_id == Product.id)` établit la **relation explicite** entre les deux tables.
- Cela nous permet de sélectionner des colonnes provenant à la fois de **Stock** et de **Product** dans la même requête.
- Le résultat retourné contient donc toutes les informations nécessaires pour un rapport clair : l'ID de l'article, son nom, son SKU, son prix unitaire et la quantité en stock.



## Capture :

LOG430 Labo03 / Products / /products

Save Share

POST {{baseUrl}}/products Send

Params Authorization Headers (8) Body Scripts Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Schema Beautify

```
1 {
2   "name": "Super Produit",
3   "sku": "CHA-002",
4   "price": 129.99
5 }
```

Body Cookies Headers (5) Test Results 201 CREATED 23 ms 189 B Save Response

JSON Preview Visualize

```
1 {
2   "product_id": 20
3 }
```

LOG430 Labo03 / Stocks / /stocks

Save Share

POST {{baseUrl}}/stocks Send

Params Authorization Headers (8) Body Scripts Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Schema Beautify

```
1 {
2   "product_id": 20,
3   "quantity": 59
4 }
```

Body Cookies Headers (5) Test Results 201 CREATED 25 ms 199 B Save Response

JSON Preview Visualize

```
1 {
2   "result": "rows added: 20"
3 }
```

HTTP LOG430 Labo03 / Stocks / /stocks/reports/overview-stocks

GET {{baseUrl}}/stocks/reports/overview-stocks

Params Authorization Headers (6) Body Scripts Settings

#### Query Params

	Key	Value	Description
	Key	Value	Description

Body Cookies Headers (5) Test Results 200 OK • 26 ms • 1.94 KB

{ } JSON Preview Visualize

```
85     },
86     {
87       "Article": 20,
88       "Nom": "Super Produit",
89       "Numéro SKU": "CHA-002",
90       "Prix unitaire": 129.99,
91       "Unités en stock": 59
92     },
93     {
94       "Article": 21,
95       "Nom": "Some Item",
96       "Numéro SKU": "12345",
```

## QUESTION 3

En utilisant l'endpoint POST /stocks/graphql-query avec la requête suggérée, j'ai pu interroger directement un produit en choisissant les colonnes que je voulais obtenir.

Exemple de requête envoyée dans Postman :

```
{
  product(id: "20") {
    id
    name
    sku
    price
    quantity
  }
}
```

Résultat obtenu :

La réponse JSON a bien retourné les informations du produit avec l'id = 20 :

```
{
  "product": {
    "id": 20,
    "name": "Super Produit",
    "sku": "CHA-002",
    "price": 129.99,
    "quantity": 59
  }
}
```

## Interprétation

- On peut sélectionner uniquement les colonnes nécessaires (id, name, sku, price, quantity), contrairement à l'endpoint REST classique qui renvoie un format prédéfini.
- GraphQL apporte donc plus de **flexibilité** : pas besoin de multiplier les endpoints pour différents besoins.
- L'exemple montre que le produit avec **id=20** est bien récupéré et que ses données sont cohérentes avec ce qui est présent dans la base de données.

# Capture :

LOG430 Lab003 / Stocks / /stocks/graphql-query

POST [baseURL]/stocks/graphql-query

Params Authorization Headers (8) Body GraphQL Auto Fetch

none form-data x-www-form-urlencoded raw binary

QUERY

```
1 {
2   product(id: "20") {
3     id
4     name
5     price
6     quantity
7   }
8 }
9
```

GRAPHQL VARIABLES

```
1 {
2   id: "20"
3 }
4
```

Body Cookies Headers (5) Test Results 200 OK 40 ms 240 B Save Response

JSON Preview Visualize

```
1 {
2   "product": {
3     "id": "20",
4     "name": "Super Product",
5     "price": 129.99,
6     "quantity": 59
7   }
8 }
9
```

LOG430 Lab003 / Stocks / /stocks/graphql-query

POST [baseURL]/stocks/graphql-query

Params Authorization Headers (8) Body GraphQL Auto Fetch

none form-data x-www-form-urlencoded raw binary

QUERY

```
1 {
2   product(id: "20") {
3     id
4     name
5     sku
6     quantity
7   }
8 }
9
```

GRAPHQL VARIABLES

```
1 {
2   id: "20"
3 }
4
```

Body Cookies Headers (5) Test Results 200 OK 32 ms 241 B Save Response

JSON Preview Visualize

```
1 {
2   "product": {
3     "id": "20",
4     "name": "Super Product",
5     "quantity": 59,
6     "sku": "CHA-892"
7   }
8 }
9
```

## QUESTION 4:

Code :

J'avais déjà réalisé ce code dans ma question 3.

```
def update_stock_redis(order_items, operation):
    """ Update stock quantities in Redis """
    if not order_items:
        return
    r = get_redis_conn()
    stock_keys = list(r.scan_iter("stock:*"))
    if stock_keys:
        pipeline = r.pipeline()
        session = get_sqlalchemy_session()
        try:
            for item in order_items:
                if hasattr(item, 'product_id'):
                    product_id = item.product_id
                    quantity = item.quantity
                else:
                    product_id = item['product_id']
                    quantity = item['quantity']
                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
                fields = _get_product_fields(session, product_id)
                pipeline.hset(
                    f"stock:{product_id}",
                    mapping={
                        "quantity": int(new_quantity),
                        "name": fields["name"] or "",
                        "sku": fields["sku"] or "",
                        "price": fields["price"] if fields["price"] is not None else "",
                    },
                )
            pipeline.execute()
        finally:
            session.close()
    else:
        _populate_redis_from_mysql(r)
```

## QUESTION 5 :

Requête GraphQL (Postman) :

```
{  
  product(id: "20") {  
    id  
    name  
    sku  
    price  
    quantity  
  }  
}
```

Réponse obtenue :

```
{  
  "product": {  
    "id": 20,  
    "name": "Super Produit",  
    "price": 129.99,  
    "quantity": 59,  
    "sku": "CHA-002"  
  }  
}
```

# Capture :

LOG430 Labo03 / Stocks / /stocks/graphql-query

Save

Share

POST

{{baseUrl}}/stocks/graphql-query

Send

Params

Authorization

Headers (8)

Body

Scripts

Settings

Cookies

☐ none

☐ form-data

☐ x-www-form-urlencoded

☐ raw

☐ binary

☒ GraphQL

Auto Fetch

QUERY

GRAPHQL VARIABLES

1

{

2

product(id: "20") {

3

id

4

name

5

sku

6

price

7

quantity

8

}

9

}

10

1

Body

Cookies

Headers (5)

Test Results

200 OK

32 ms

256 B

Save Response

{}

JSON

Preview

Visualize

1

{

2

"product": {

3

"id": 20,

4

"name": "Super Produit",

5

"price": 129.99,

6

"quantity": 59,

7

"sku": "CHA-002"

8

}

9

}



## QUESTION 6 :

Pour cette tâche, j'ai simulé un fournisseur externe en lançant un deuxième conteneur basé sur `scripts/supplier_app.py`.

J'ai construit l'image avec le Dockerfile dans `scripts/` et démarré le conteneur via `scripts/docker-compose.yml`.

Ce conteneur exécute périodiquement (toutes les 10 secondes) des appels HTTP POST vers l'endpoint `/stocks/graphql-query` du service `store_manager`.

Comme j'avais modifié le schéma GraphQL à l'étape précédente (colonne `name`, `sku`, `price`, etc.), j'ai pu vérifier dans les logs que la réponse retournait bien un objet produit complet :

Logs	Inspect	Bind mounts	Exec	Files	Stats
2025-10-03 04:54:53,971	-	INFO	-	Waiting 10 seconds until next call...	
2025-10-03 04:55:03,971	-	INFO	-	--- Call #26 ---	
2025-10-03 04:55:03,972	-	INFO	-	Calling <a href="http://store_manager:5000/stocks/graphql-query">http://store_manager:5000/stocks/graphql-query</a> (attempt 1/3)	
2025-10-03 04:55:04,011	-	INFO	-	Response: 200 - OK	
2025-10-03 04:55:04,012	-	INFO	-	Response body: {"product":{"id":20,"name":"Super Produit","price":129.99,"quantity":59,"sku":"CHA-002"}}	
...					
2025-10-03 04:55:04,012	-	INFO	-	Waiting 10 seconds until next call...	
2025-10-03 04:55:14,012	-	INFO	-	--- Call #27 ---	
2025-10-03 04:55:14,013	-	INFO	-	Calling <a href="http://store_manager:5000/stocks/graphql-query">http://store_manager:5000/stocks/graphql-query</a> (attempt 1/3)	
2025-10-03 04:55:14,040	-	INFO	-	Response: 200 - OK	
2025-10-03 04:55:14,040	-	INFO	-	Response body: {"product":{"id":20,"name":"Super Produit","price":129.99,"quantity":59,"sku":"CHA-002"}}	
...					
2025-10-03 04:55:14,040	-	INFO	-	Waiting 10 seconds until next call...	
2025-10-03 04:55:24,039	-	INFO	-	--- Call #28 ---	
2025-10-03 04:55:24,040	-	INFO	-	Calling <a href="http://store_manager:5000/stocks/graphql-query">http://store_manager:5000/stocks/graphql-query</a> (attempt 1/3)	
2025-10-03 04:55:24,068	-	INFO	-	Response: 200 - OK	
2025-10-03 04:55:24,068	-	INFO	-	Response body: {"product":{"id":20,"name":"Super Produit","price":129.99,"quantity":59,"sku":"CHA-002"}}	
...					
2025-10-03 04:55:24,069	-	INFO	-	Waiting 10 seconds until next call...	
2025-10-03 04:55:34,069	-	INFO	-	--- Call #29 ---	
2025-10-03 04:55:34,070	-	INFO	-	Calling <a href="http://store_manager:5000/stocks/graphql-query">http://store_manager:5000/stocks/graphql-query</a> (attempt 1/3)	
2025-10-03 04:55:34,095	-	INFO	-	Response: 200 - OK	
2025-10-03 04:55:34,095	-	INFO	-	Response body: {"product":{"id":20,"name":"Super Produit","price":129.99,"quantity":59,"sku":"CHA-002"}}	
...					
2025-10-03 04:55:34,095	-	INFO	-	Waiting 10 seconds until next call...	
2025-10-03 04:55:44,096	-	INFO	-	--- Call #30 ---	
2025-10-03 04:55:44,096	-	INFO	-	Calling <a href="http://store_manager:5000/stocks/graphql-query">http://store_manager:5000/stocks/graphql-query</a> (attempt 1/3)	
2025-10-03 04:55:44,148	-	INFO	-	Response: 200 - OK	
2025-10-03 04:55:44,148	-	INFO	-	Response body: {"product":{"id":20,"name":"Super Produit","price":129.99,"quantity":59,"sku":"CHA-002"}}	
...					
2025-10-03 04:55:44,148	-	INFO	-	Statistics: 30 calls, 30 successful (100.0%), 0 errors	
2025-10-03 04:55:44,148	-	INFO	-	Waiting 10 seconds until next call...	
2025-10-03 04:55:54,148	-	INFO	-	--- Call #31 ---	
2025-10-03 04:55:54,148	-	INFO	-	Calling <a href="http://store_manager:5000/stocks/graphql-query">http://store_manager:5000/stocks/graphql-query</a> (attempt 1/3)	
2025-10-03 04:55:54,179	-	INFO	-	Response: 200 - OK	
2025-10-03 04:55:54,179	-	INFO	-	Response body: {"product":{"id":20,"name":"Super Produit","price":129.99,"quantity":59,"sku":"CHA-002"}}	

En regardant attentivement les deux fichiers (`docker-compose.yml` à la racine et celui dans `scripts/`), je constate qu'ils ont en commun la définition d'un réseau Docker partagé.

## Mécanisme

- Chaque conteneur attaché à ce réseau peut **résoudre les noms des autres conteneurs** (grâce au DNS interne Docker).
- Concrètement, depuis `supplier_app`, j'accède à `store_manager` via `http://store_manager:5000` au lieu d'utiliser une IP.
- Les **ports exposés** (`5000:5000`, `3306:3306`, etc.) servent uniquement pour accéder depuis ma machine hôte.
- Mais **entre conteneurs**, on utilise le port interne et le nom du service.

## YML : côté racine

```
services:
  store_manager:
    build: .
    environment:
      - PYTHONUNBUFFERED=1
    volumes:
      - ./app
    ports:
      - "5000:5000"
    networks:
      - labo03-network
    depends_on:
      - mysql
      - redis

  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"
    volumes:
      - mysql_data:/var/lib/mysql
      - ./db-init:/docker-entrypoint-initdb.d

  redis:
    image: redis:7
    restart: unless-stopped
    networks:
      - labo03-network
    ports:
      - "6379:6379"

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

volumes:
  mysql_data:
```

## YML : côté scripts

```
1  services:
2  |   supplier_app:
3  |     build: .
4  |     environment:
5  |       - PYTHONUNBUFFERED=1
6  |     volumes:
7  |       - ./app
8  |     networks:
9  |       - labo03-network
10
11 networks:
12 |   labo03-network:
13 |     driver: bridge
14 |     external: true
15 |     name: labo03-network
16
```

**En résumé :** Les deux docker-compose.yml partagent un **même réseau externe**. C'est ce réseau qui permet à `supplier_app` et `store_manager` (ainsi que `mysql` et `redis`) de communiquer entre eux directement par leurs **noms de service**.

## TEST

```
PS C:\Users\Jolan\Desktop\LOG4430\LAB3\sdfsdl\log430-a25-labo3\src> python -m pytest tests/test_store_manager.py -q
>>
..
2 passed in 1.32s
PS C:\Users\Jolan\Desktop\LOG4430\LAB3\sdfsdl\log430-a25-labo3\src>
```

## CI / CD

