

Labo 02 — Rapport



ÉCOLE DE
TECHNOLOGIE
SUPÉRIEURE

Université du Québec

Felix-Antoine Legault

Rapport de laboratoire

LOG430 — Architecture logicielle

Montreal, 3 octobre 2025

École de technologie supérieure

Questions

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.

À la fin de la suite du Smoke Test, j'ai obtenu 5 unités de stock pour l'article créé durant le test. En effet, on crée un article avec une quantité initiale de 5, puis on crée une commande de 2 unités, ce qui réduit la quantité en stock à 3 unités. Ensuite, on supprime la commande, ce qui remet la quantité en stock à 5 unités.

Pour ce qui en est de l'article avec $id=2$, la quantité en stock est 500 unités.

GET `{{baseUrl}}/stocks/2` Send

Params Authorization Headers (6) Body Scripts Tests Settings Cookies

Query Params

Key	Value	Description	Bulk Edit
Key	Value	Description	

Body Cookies Headers (5) Test Results 201 CREATED • 16 ms • 203 B • Save Response

{ } JSON Preview Visualize

```

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.

Dans ce cas, la méthode `join` est utilisée pour effectuer une jointure entre les tables `Stock` et `Product` afin de récupérer des informations supplémentaires sur le produit associé à chaque stock. On utilise la clé étrangère `product_id` dans la table `Stock` pour faire la correspondance avec la clé primaire `id` dans la table `Item`. la méthode `join` est décrite dans la section [Joins to a Target with an ON Clause](#) de la documentation SQLAlchemy.

```

def get_stock_for_all_products():
    """Get stock quantity for all products"""
    session = get_sqlalchemy_session()

    results = (
        session.query(
            Stock.product_id,
            Stock.quantity,
            Product.name,
            Product.sku,
            Product.price,
        )
    )

```

```
        .join(Product, Stock.product_id == Product.id)
        .all()
    )

    stock_data = []
    for row in results:
        stock_data.append(
            {
                "Article": row.name,
                "Numéro SKU": row.sku,
                "Prix unitaire": float(row.price),
                "Unités en stock": int(row.quantity),
            }
        )

    return stock_data
```

Question 3

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

Avec la requête suggérée, j'ai obtenu un produit qui s'appelle Product 1 avec une quantité de 1000.

The screenshot shows a REST client interface with the following details:

- URL:** `{{baseUrl}} /stocks/graphql-query`
- Method:** POST
- Body:** GraphQL query:

```
1 {
2   product(productId: "1") {
3     id
4     name
5     quantity
6   }
7 }
8
```
- GraphQL Variables:** `1`
- Status:** 200 OK, 4 ms, 245 B
- Response Body (JSON):**

```
1 {
2   "data": {
3     "product": {
4       "id": 1,
5       "name": "Product 1",
6       "quantity": 1000
7     }
8   },
9   "errors": null
10 }
```

Question 4

Quelles lignes avez-vous changé dans `update_stock_redis`? Veuillez joindre du code afin d'illustrer votre réponse.

PROF

J'ai ajouté une ligne qui va chercher les informations du Product dans la bd SQL qui correspond au `product_id` du stock. Ensuite, j'ai modifié la ligne qui fait le `hset` pour ajouter les nouvelles colonnes `name`, `sku` et `price`. J'utilise le mapping du `hset` pour ajouter toutes les colonnes en une seule fois.

```
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:*"))
    session = get_sqlalchemy_session()

    if stock_keys:
        pipeline = r.pipeline()
        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"]

        product = session.query(Product).filter(Product.id ==
product_id).first()

        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

        pipeline.hset(
            f"stock:{product_id}",
            mapping={
                "quantity": new_quantity,
                "name": product.name,
                "sku": product.sku,
                "price": product.price,
            },
        )

        pipeline.execute()
    else:
        _populate_redis_from_mysql(r)

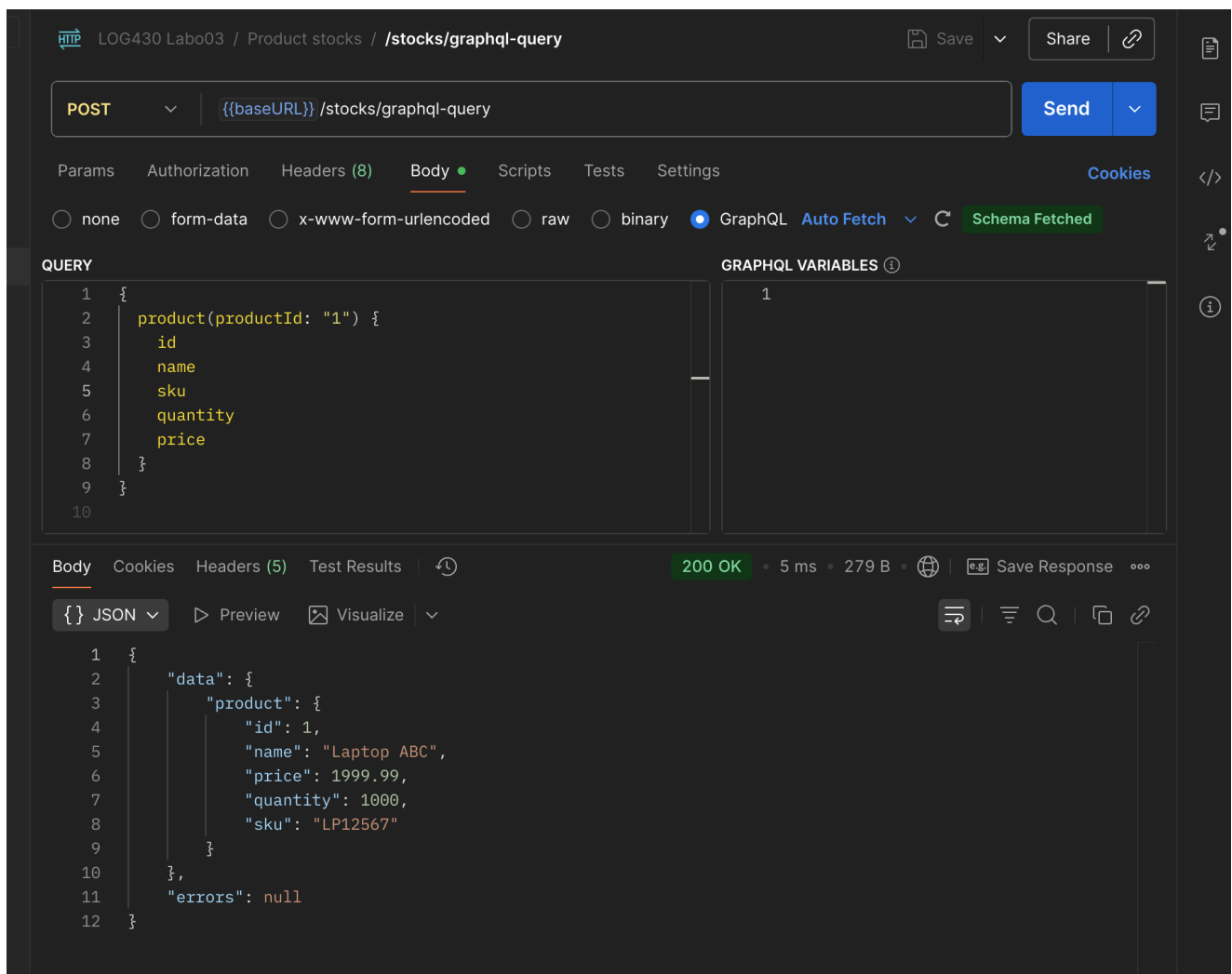
```

Question 5

PROF

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

Maintenant, on obtient les informations supplémentaires du produit, soit le nom, le sku et le prix. Le nom n'est plus Product {id}, mais bien son vrai nom.



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.

PROF Ils peuvent communiquer entre eux grâce au réseau créé avec la commande `docker network create labo03-network`. Ils peuvent donc se parler entre eux parce qu'on spécifie le réseau dans les deux fichiers `docker-compose.yml` avec ceci :

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

Supplier App script

Pour faire fonctionner le supplier app script, j'ai dû modifier l'URL de l'API pour qu'elle pointe vers le bon endpoint (`graphql` -> `graphql-query`). J'ai aussi ajouté `sku quantity price` dans le corps de la requête GraphQL pour obtenir les informations supplémentaires du produit. Voici une capture d'écran

du changement et une capture des logs du conteneur qui montre que les requêtes fonctionnent bien à partir du script.

```
ENDPOINT_URL = "http://store_manager:5000/stocks/graphql-query"

TEST_PAYLOAD = '{"query":{"{ product(productId: \\\"1\\\") { id name sku quantity price } }"},"variables":{}}'
INTERVAL_SECONDS = 10
TIMEOUT_SECONDS = 10
MAX_RETRIES = 3
```

```
root@log430-a25-labo3:/scripts# docker logs log430-a25-labo3-store_manager-1
08 check 1/5 failed: 2003 (HY000): Can't connect to MySQL server on 'mysql:3306' (111)
08 check 2/5 failed: 2003 (HY000): Can't connect to MySQL server on 'mysql:3306' (111)
08 check 3/5 failed: 2003 (HY000): Can't connect to MySQL server on 'mysql:3306' (111)
08 check 4/5 failed: 2003 (HY000): Can't connect to MySQL server on 'mysql:3306' (111)
4 enregistrements de stock ont été synchronisés avec Redis
Redis sync done
* Serving Flask app 'store_manager'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.18.0.4:5000
Press CTRL-C to quit
172.18.0.5 - - [01/Oct/2025 23:58:20] "POST /stocks/graphql-query HTTP/1.1" 200 -
172.18.0.5 - - [01/Oct/2025 23:58:30] "POST /stocks/graphql-query HTTP/1.1" 200 -
172.18.0.5 - - [01/Oct/2025 23:58:40] "POST /stocks/graphql-query HTTP/1.1" 200 -
root@log430-a25-labo3:/scripts#

root@log430-a25-labo3:/scripts# docker logs scripts-supplier_app-1
2025-10-01 23:58:20,105 - INFO - Starting periodic calls to http://store_manager:5000/stocks/graphql-query every 10 seconds
2025-10-01 23:58:20,105 - INFO - --- Call #1 ---
2025-10-01 23:58:20,105 - INFO - Press CTRL-C to stop
2025-10-01 23:58:20,105 - INFO - --- Call #1 ---
2025-10-01 23:58:20,106 - INFO - Calling http://store_manager:5000/stocks/graphql-query (attempt 1/3)
2025-10-01 23:58:20,111 - INFO - Response: 200 - OK
2025-10-01 23:58:20,111 - INFO - Response body: {"data":{"product":{"id":"1","name":"Laptop ABC","price":1999.99,"quantity":1000,"sku":"LP12567"}}, "errors":null}}
...
2025-10-01 23:58:20,111 - INFO - Waiting 10 seconds until next call...
2025-10-01 23:58:30,111 - INFO - --- Call #2 ---
2025-10-01 23:58:30,111 - INFO - Calling http://store_manager:5000/stocks/graphql-query (attempt 1/3)
2025-10-01 23:58:30,122 - INFO - Response: 200 - OK
2025-10-01 23:58:30,123 - INFO - Response body: {"data":{"product":{"id":"1","name":"Laptop ABC","price":1999.99,"quantity":1000,"sku":"LP12567"}}, "errors":null}}
...
2025-10-01 23:58:30,123 - INFO - Waiting 10 seconds until next call...
2025-10-01 23:58:40,123 - INFO - --- Call #3 ---
2025-10-01 23:58:40,123 - INFO - Calling http://store_manager:5000/stocks/graphql-query (attempt 1/3)
2025-10-01 23:58:40,127 - INFO - Response: 200 - OK
2025-10-01 23:58:40,127 - INFO - Response body: {"data":{"product":{"id":"1","name":"Laptop ABC","price":1999.99,"quantity":1000,"sku":"LP12567"}}, "errors":null}}
...
2025-10-01 23:58:40,127 - INFO - Waiting 10 seconds until next call...
root@log430-a25-labo3:/scripts#
```

CI/CD

Le CI/CD est configuré avec GitHub Actions. Le workflow est déclenché à chaque push ou pull request vers la branche `main`. Il lance Redis et MySQL, puis lance les tests (voir plus bas une capture d'écran des tests). De plus, avec un self host runner sur la VM, il a une étape qui fait le déploiement en continu de l'application. On a donc une pipeline CI/CD qui test, build et déploie l'application automatiquement.

```
Run tests inside app container

1 ▶ Run docker compose exec -T store_manager bash -c "PYTHONPATH=. pytest -v --cache-clear"
4 ===== test session starts =====
5 platform linux -- Python 3.11.13, pytest-8.4.2, pluggy-1.6.0 -- /usr/local/bin/python
6 cachedir: .pytest_cache
7 rootdir: /app
8 configfile: pyproject.toml
9 plugins: cov-7.0.0
10 collecting ... collected 2 items
11
12 tests/test_store_manager.py::test_health PASSED [ 50%]
13 tests/test_store_manager.py::test_stock_flow PASSED [100%]
14
15 ===== warnings summary =====
16 src/orders/models/base.py:10
17 /app/src/orders/models/base.py:10: MovedIn20Warning: The ``declarative_base()`` function is now available as sqlalchemy.orm.declarative_base(). (deprecated since: 2.0) (Background on SQLAlchemy 2.0 at: https://sqlalche.me/e/b8d9)
18 Base = declarative_base()
19
20 -- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
21 ===== 2 passed, 1 warning in 0.64s =====
```

Build + Integration Tests (GitHub runner)

succeeded 3 minutes ago in 1m 6s

- > ✓ Set up job
- > ✓ Checkout
- > ✓ Ensure Docker network exists
- > ✓ Write .env
- > ✓ Reset any previous stack
- > ✓ Build & start services
- > ✓ Wait for services to be healthy
- > ✓ Run tests inside app container
- > ✓ Cleanup
- > ✓ Post Checkout
- > ✓ Complete job

PROF

Rapport de stock

Le rapport de stock est généré en appelant l'endpoint `/stocks/report` de l'API. Voici une capture d'écran du rapport généré:

POST /proi

POST /proi

POST /orc

DEL /order

GET /healt

GET /stoc

POST /stoc

>

+

▼

No environment

▼

LOG430 Labo03 / Product stocks / /stocks/reports/overview-stocks

Save

▼

Share

🔗

GET

▼

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

Send

▼

Params

Authorization

Headers (6)

Body

Scripts

Tests

Settings

Cookies

Query Params

	Key	Value	Description	...	Bulk Edit
	Key	Value	Description		

Body

Cookies

Headers (5)

Test Results

🔄

200 OK

• 15 ms

• 586 B

• 🌐

• 📄 Save Response

...

{{}}

JSON

▼

▶ Preview

🖼 Visualize

▼

🔄

☰

🔍

📄

🔗

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

[

{

"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

},

{

"Article": "Gadget XYZ",

"Numéro SKU": "GG12345",

"Prix unitaire": 5.75,

"Unités en stock": 2

},

{

"Article": "27-inch Screen WYZ",

"Numéro SKU": "SC27289",

"Prix unitaire": 299.75,

"Unités en stock": 90

}

]

PROF

Tests

Avec le smoke test, j'ai pu vérifier que les principales fonctionnalités de l'API fonctionnaient correctement. Voici le code du test:

```

def test_stock_flow(client: Flask):
    # 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"
    )
    expected_status_code = 201

    assert (
        response.status_code == expected_status_code
    ), "1. `POST /products` wrong status code"
    data = response.get_json()
    assert data["product_id"] > 0

    # 2. Ajoutez 5 unités au stock de cet article (`POST /stocks`)
    product_id = data["product_id"]
    stock_quantity = 5
    stock_data = {"product_id": product_id, "quantity": stock_quantity}
    response = client.post(
        "/stocks", data=json.dumps(stock_data),
        content_type="application/json"
    )
    expected_status_code = 201

    assert (
        response.status_code == expected_status_code
    ), "2. `POST /stocks` wrong status code"
    data = response.get_json()
    assert f"rows added: {product_id}" in data["result"]

    # 3. Vérifiez le stock, votre article devra avoir 5 unités dans le
    stock (`GET /stocks/:id`)
    response = client.get(f"/stocks/{product_id}",
        content_type="application/json")
    expected_status_code = 201

    assert (
        response.status_code == expected_status_code
    ), "3. `GET /stocks/:id` wrong status code"
    data = response.get_json()
    assert stock_quantity == data["quantity"]

    # 4. Faites une commande de l'article que vous avez crée, 2 unités
    (`POST /orders`)
    order_quantity = 2
    product_data = {
        "user_id": 1,
        "items": [{"product_id": product_id, "quantity":
order_quantity}],
    }
    response = client.post(

```

```

        "/orders", data=json.dumps(product_data),
content_type="application/json"
    )
    expected_status_code = 201
    order_id = response.get_json()["order_id"]

    assert (
        response.status_code == expected_status_code
    ), "4. `POST /orders` wrong status code"

    # 5. Vérifiez le stock encore une fois (`GET /stocks/:id`)
    response = client.get(f"/stocks/{product_id}",
content_type="application/json")
    expected_status_code = 201

    assert (
        response.status_code == expected_status_code
    ), "5. `GET /stocks/:id` wrong status code"
    data = response.get_json()
    assert stock_quantity - order_quantity == data["quantity"]

    # 6. Étape extra: supprimez la commande et vérifiez le stock de
nouveau.
    # Le stock devrait augmenter après la suppression de la commande.
    response = client.delete(f"/orders/{order_id}")
    expected_status_code = 200

    assert (
        response.status_code == expected_status_code
    ), "6. `DELETE /orders/:id` wrong status code"
    data = response.get_json()
    assert data["deleted"]

    # Vérifiez le stock encore une fois (`GET /stocks/:id`)
    response = client.get(f"/stocks/{product_id}",
content_type="application/json")
    expected_status_code = 201

    assert (
        response.status_code == expected_status_code
    ), "6. `GET /stocks/:id` wrong status code"
    data = response.get_json()
    assert stock_quantity == data["quantity"]

```