Labo 03 – REST APIs, GraphQL

PAR

Marc CHARLEBOIS, CHAM65260301

RAPPORT DE LABORATOIRE PRÉSENTÉ À MONSIEUR FABIO PETRILLO DANS LE CADRE DU COURS *ARCHITECTURE LOGICIELLE* (LOG430-01)

MONTRÉAL, LE 3 OCTOBRE 2025

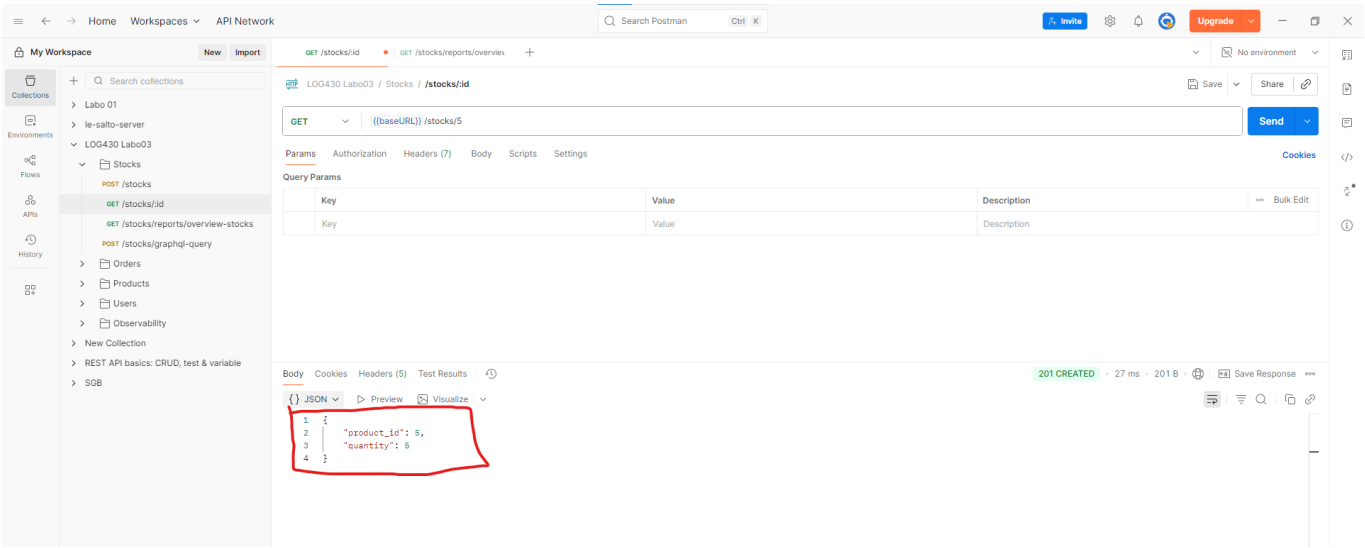
ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

Tables des matières

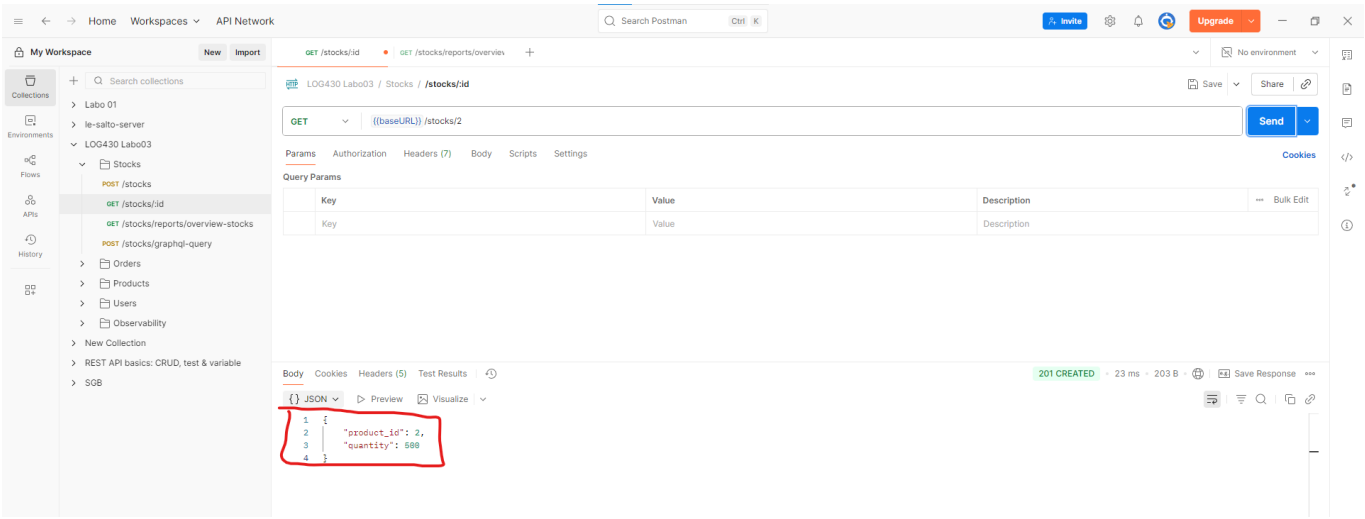
Question 1

> 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 suite du test du flux de stock, mon article nouvellement créé a bien vu son stock évoluer de façon cohérente. Après l'ajout initial de 5 unités, puis la création d'une commande de 2 unités, le stock est descendu à 3 unités. Lorsque j'ai supprimé cette commande (étape extra), le stock est revenu à 5 unités, ce qui confirme que le système met correctement à jour la base de données et reflète l'état attendu du stock en fonction des opérations effectuées.



Concernant l'article déjà présent dans la base avec l'id=2, celui-ci n'a pas été touché durant le scénario de test. Son stock est donc resté inchangé à 500 unités, tel qu'initialisé dans le script SQL fourni. Les résultats obtenus montrent ainsi que les opérations sur un produit donné n'affectent pas les stocks des autres articles de la base.



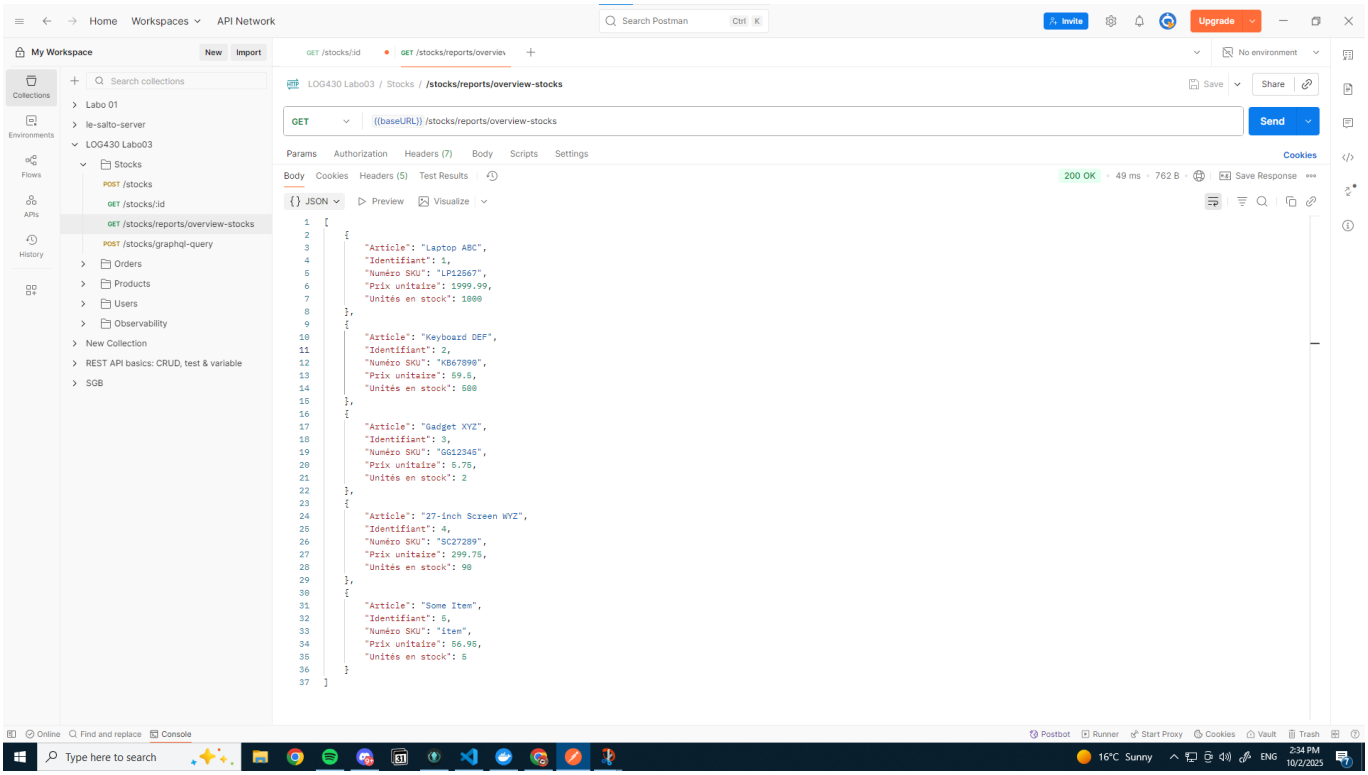
Question 2

> 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 demandés dans cette activité. Veuillez inclure le code pour illustrer votre réponse.

La méthode join de SQLAlchemy est utilisée ici pour relier la table `stocks` à la table `products` grâce à la clé étrangère `Stock.product_id` et la clé primaire `Product.id`. Cette jointure permet de compléter les informations de stock (quantité) avec des données du produit (`name`, `sku`, `price`) issues de la table `products`, ce qui correspond à une clause `INNER JOIN` en SQL. Ainsi, chaque ligne renvoyée contient à la fois les informations d'inventaire et les détails du produit, comme le montre le code ci-dessous :

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

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

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

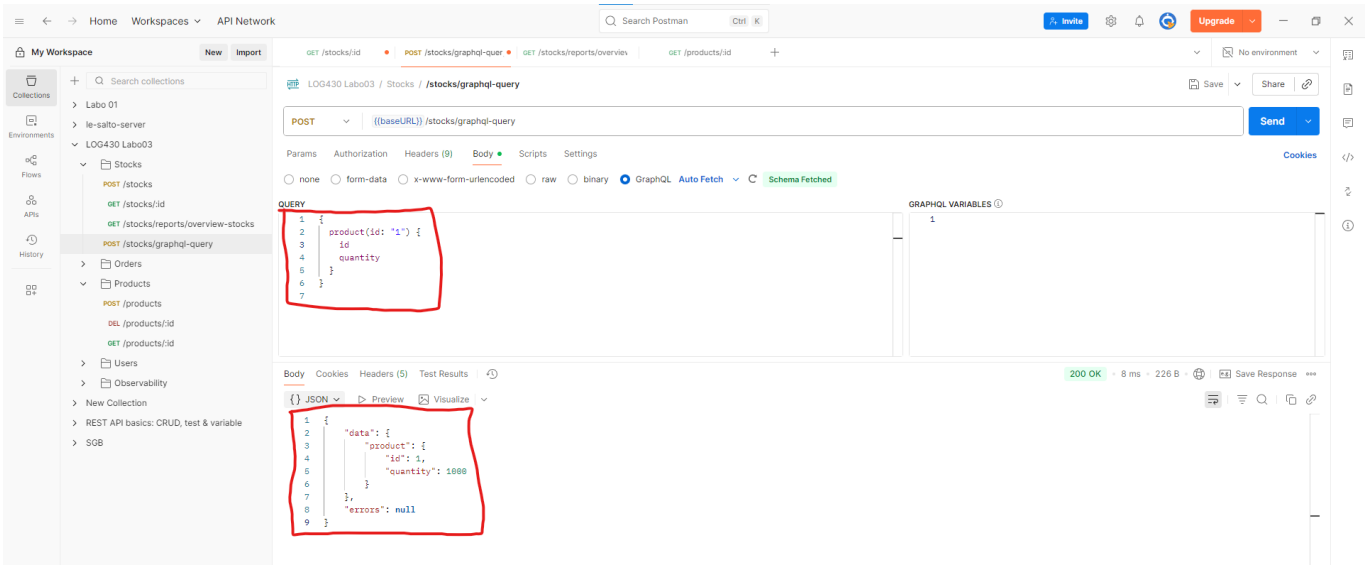
```
    return stock_data
```



Question 3

> Quels résultats avez-vous obtenus en utilisant l'endpoint POST /stocks/graphql 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-query` avec la requête GraphQL suggérée, nous avons obtenu en sortie un objet JSON contenant les informations du produit avec l'`id = 1`. Plus précisément, la réponse inclut les champs demandés `id` et `quantity`, ce qui montre la capacité de l'endpoint à gérer dynamiquement les attributs retournés. Le résultat était le suivant :



Question 4

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

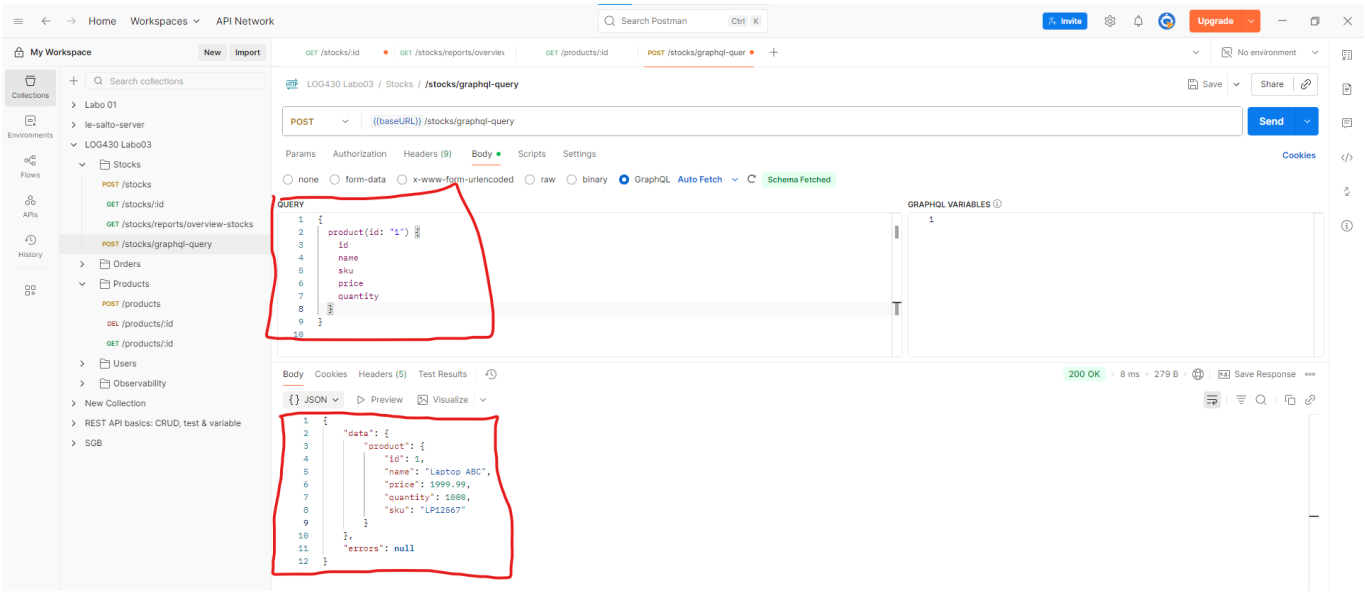
J'ai modifié la fonction update_stock_redis en ajoutant un appel à `get_product_by_id(product_id)` pour récupérer les informations complètes du produit (`name`, `sku`, `price`), puis je les ai intégrées directement dans le `pipeline.hset` en plus de la `quantity` qui était déjà présente. Ainsi, chaque entrée Redis pour un produit contient désormais toutes les informations nécessaires pour que l'endpoint GraphQL puisse les exposer.

```
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()
        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

            product = get_product_by_id(product_id)

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

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

    else:
        _populate_redis_from_mysql(r)
```

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.

Après l'ajout des champs name, sku et price dans le schéma GraphQL et l'enrichissement de Redis, l'endpoint `POST /stocks/graphql-query` retourne désormais toutes les informations demandées pour un produit. En exécutant la même requête, mais en ajoutant les nouveaux champs `sku`, `name` et `price`, ce qui m'a donné le résultat suivant:



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.

Les deux fichiers docker-compose.yml ont en commun qu'ils attachent tous leurs services au même réseau utilisateur `labo03-network` (déclaré en external et de type bridge). C'est précisément ce réseau Docker partagé qui permet la communication inter-conteneurs via la résolution DNS intégrée de Docker : chaque service est joignable par son nom de service (ex. store_manager) plutôt que par localhost. Ainsi, le conteneur supplier_app peut appeler l'API par `http://store_manager:5000/...` sans exposer de port supplémentaire entre conteneurs (les ports: ne servent que pour l'accès depuis la machine hôte).

Le service `store_manager`, `mysql` et `redis` sont reliés au réseau `labo03-network` pour être accessible par les autres conteneurs.

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

  mysql:
    networks:
      - labo03-network

  redis:
```

```
    networks:
      - labo03-network

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

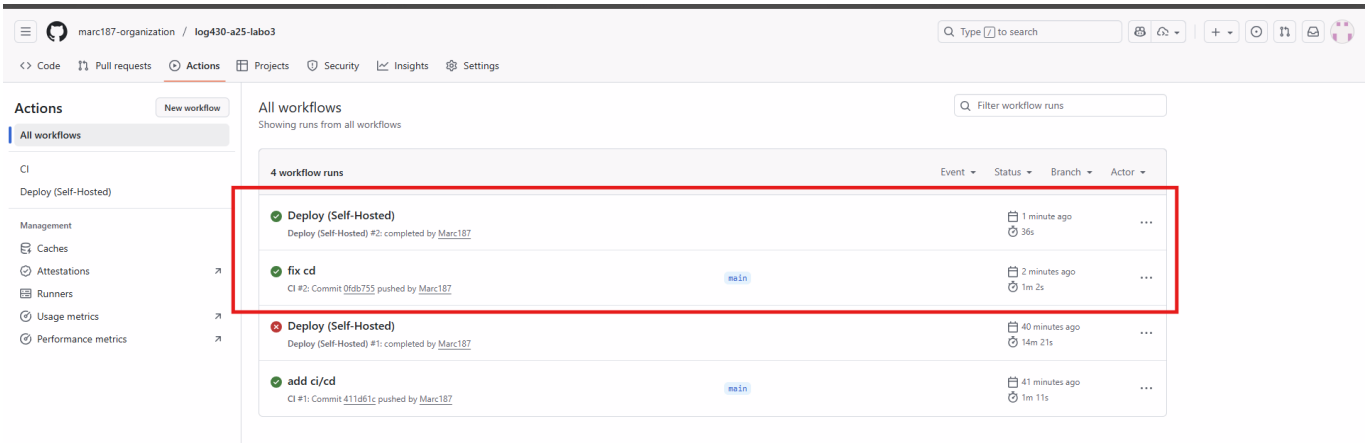
On réutilise le réseau `labo03-network` (déjà créé) pour garantir que `supplier_app` et les autres services sont dans le même espace réseau.

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

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

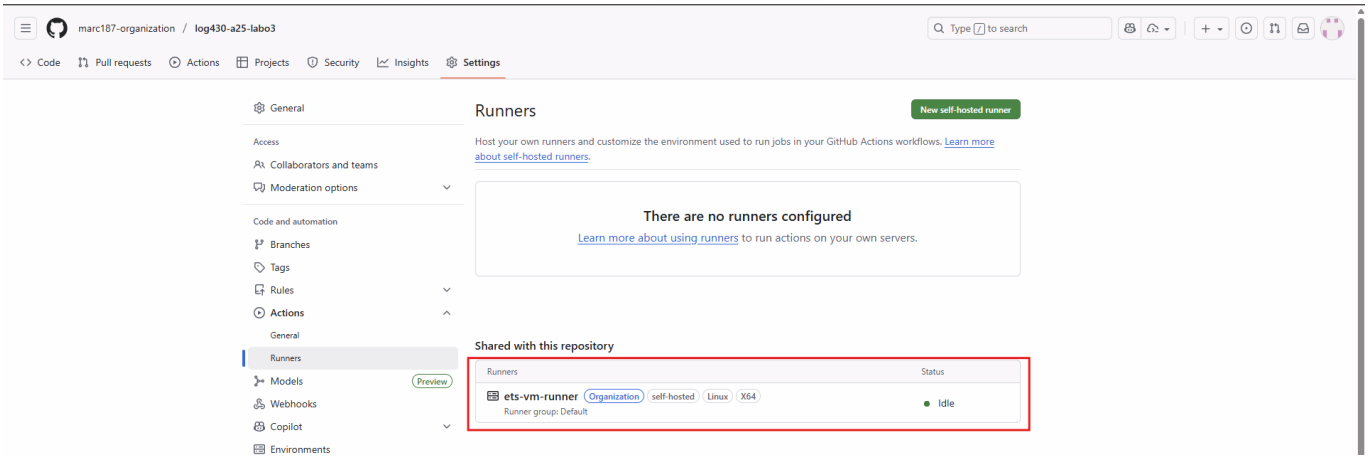
CI/CD

Mon pipeline CI/CD fonctionne ainsi : lors de chaque push ou pull request, mon script CI s'exécute sur GitHub Actions, lance un environnement avec MySQL et Redis, installe les dépendances et exécute les tests pour valider mon code. Si tout est correct, mon script CD se déclenche automatiquement via un runner self-hosted installé sur ma VM, qui récupère le dépôt, génère le fichier .env, construit et démarre les conteneurs avec Docker Compose, puis affiche l'état et les logs pour confirmer le déploiement.

On peut voir ci-dessous que les deux workflows se sont exécutés correctement, ce qui confirme que l'application a été testée puis déployée sans erreur.



Le déploiement s'effectue sur mon runner auto-hébergé configuré sur la VM, qui exécute directement les commandes Docker.



La commande `docker ps` montre que les conteneurs sont bien lancés sur la VM et que l'application est en fonctionnement.