

LOG430 - Rapport du laboratoire 02

ÉTS - LOG430 - Architecture logicielle - Hiver 2026 - Groupe 1

Étudiant: Yanni Haddar

Questions

💡 Question 1 : Dans la RFC 7231, nous trouvons que certaines méthodes HTTP sont considérées comme sûres (`safe`) ou idempotentes, en fonction de leur capacité à modifier (ou non) l'état de l'application. Lisez les sections 4.2.1 et 4.2.2 de la RFC 7231 et répondez : parmi les méthodes mentionnées dans l'activité 2, lesquelles sont sûres, non sûres, idempotentes et/ou non idempotentes?

Le GET (etapes 3 et 5) est sur et idempotent. Elle est `read only`, et ne modifie rien dans le serveur. Puisqu'elle est sure, elle est forcément idempotente.

Le POST (etapes 1, 2, 4) est non sur et non idempotent. Il modifie le serveur et rajoute des lignes, ou met à jour des compteurs. Il est aussi non-idempotent car la même requête de commande créera une deuxième commande.

Le DELETE (étape 6) est non sur mais idempotent. Elle altere des données sur le serveur, mais l'exécuter deux fois ne fera rien car la commande sera déjà supprimée et la base de données ne fera rien.

💡 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.

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

La méthode `join` est utilisée pour faire une jointure avec une clause `ON` explicite. Comme les tables `Product` et `Stock` sont liées par l'`ID` du produit, le `join` permet d'avoir toutes les données voulues.

💡 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.

```
{
  "data": {
```

```
        "product": null
    },
    "errors": null
}
```

Ceci est du au fait que le produit 1 (par défaut dans la collection Postman) n'est pas dans la cache redis car son stock n'a pas encore été mis à jour. Utilisons 13, qui a été interagit avec:

```
{
  "data": {
    "product": {
      "id": 13,
      "name": "Product 13",
      "quantity": 5
    }
  },
  "errors": null
}
```

On peut voir que toute les informations attendues sont présentes.

 Question 4 : Quelles lignes avez-vous changé dans update_stock_redis? Veuillez joindre du code afin d'illustrer votre réponse.

Les nouvelles lignes ont été marquées d'un +:

```
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:
+       session = get_sqlalchemy_session() # Utilisation d'une session
alchemistry
        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

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

```

        new_quantity = current_stock - quantity

+
+         p = session.execute(
+             text("SELECT name, sku, price FROM products WHERE id =
+:id"),
+             {"id": product_id}
+         ).fetchone()

+
+         pipeline.hset(f"stock:{product_id}", mapping={
+             "quantity": new_quantity,
+             "name": p.name,          # Ajouté
+             "sku": p.sku,            # Ajouté
+             "price": str(p.price) # Ajouté (converti en string pour
Redis)
+
+     })
+
+     _upsert_stock_to_redis(r, session, product_id, quantity)

         pipeline.execute()
+
+     session.close() # on doit maintenant fermer la session
else:
    _populate_redis_from_mysql(r)

```

Voici ce qui a été rajouté:

- get alchemy session et close alchemy session, évidemment on utilise maintenant alchemy, il faut l'ouvrir et le fermer.
- dans l'exécution, on rajoute maintenant les colonnes manquantes
- finalement, l'ajout de la méthode `_upsert_stock_to_redis`. Cette méthode a été créée à cause des complications Redis. Redis ne mettait sa cache à jour que pour les objets directement affectés par une requête. C'est à dire qu'après un Redis flush (si jamais certaines valeurs comme le prix ou le stock avaient changé) Redis n'était pas à jour sur tout les items sauf l'item directement affecté par une requête POST sur le stock. Par conséquent, les appels au endpoint graphql n'étaient jamais à jour, avec plusieurs produits manquants, et d'autres n'ayant pas les bonnes informations. Donc, j'ai créé une méthode qui rafraîchit tout les objets, et elle est appelée à chaque changement. C'est lourd, mais critique dans le contexte d'un laboratoire. Si c'était une base de données semi-statique, l'optimisation serait sensée, mais dans le cadre d'un laboratoire les valeurs sont mises à jour souvent et rapidement, et Redis doit refléter ces changements.

 Question 5 : 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.

Par défaut:

```
{
  "data": {
    "product": {
      "id": 13,
      "name": "Laptop X1 Carbon",
      "sku": "P1234567890",
      "price": 1200.0
    }
  }
}
```

```
        "name": "Product 13",
        "quantity": 5
    }
},
"errors": null
}
```

Changeons la requête pour avoir les nouvelles colonnes: { product(id: "13") { id name sku price quantity } }

Résultat:

```
{
  "data": {
    "product": {
      "id": 13,
      "name": "Product 13",
      "price": 0.0,
      "quantity": 5,
      "sku": ""
    }
  },
  "errors": null
}
```

Il s'agit du résultat attendu.

 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

Voici la section que les deux fins de fichier ont en commun:

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

Ces conteneurs peuvent se communiquer car ils sont sur le même network et forment une connexion bridge, qui signifie que il y a une transparence complète de réseau entre les deux conteneurs.

Déploiement

Pour commencer, voici le résultat du script de test de supplier app:

```
dusty@dusty-laptop:~/SchoolRepos/log430-labo3-mapleduck/scripts$ docker
compose up
Attaching to supplier_app-1
```

```
supplier_app-1 | 2026-02-12 23:57:49,212 - INFO - Starting periodic calls
to http://store_manager:5000/stocks/graphql-query every 10 seconds
supplier_app-1 | 2026-02-12 23:57:49,212 - INFO - Press Ctrl+C to stop
supplier_app-1 | 2026-02-12 23:57:49,213 - INFO - --- Call #1 ---
supplier_app-1 | 2026-02-12 23:57:49,213 - INFO - Calling
http://store_manager:5000/stocks/graphql-query (attempt 1/3)
supplier_app-1 | 2026-02-12 23:57:49,219 - INFO - Response: 200 - OK
supplier_app-1 | 2026-02-12 23:57:49,219 - INFO - Response body: {"data": {"product": {"id": 1, "name": "Laptop ABC", "price": 1999.99, "quantity": 1991, "sku": "LP12567"}}, "errors": null}
supplier_app-1 | ...
supplier_app-1 | 2026-02-12 23:57:49,219 - INFO - Waiting 10 seconds until
next call...
Gracefully Stopping... press Ctrl+C again to force
```

Succès.

Déploiement très similaire au dernier labo, il faut encore une fois spécifier les noms des conteneurs car docker leur donne un nom unique et ils ne peuvent donc pas se retrouver entre eux.

Les tests échouaient, puis je me suis rendu compte que lors de ce labo, il avait fallu créer la table stock, elle n'était pas créé par défaut (ce qui était un des buts du labo). Mais évidemment, il faut faire de même sur la VM, car la VM n'a pas non plus la base de données.

Ceci a donc été ajouté dans le `init.sql`:

```
-- Product stocks
-- Il faut automatiquement rajouter la table stock pour que le CI
fonctionne.
DROP TABLE IF EXISTS stocks;
CREATE TABLE stocks (
    product_id INT PRIMARY KEY,
    quantity INT NOT NULL DEFAULT 0,
    FOREIGN KEY (product_id) REFERENCES products(id) ON DELETE CASCADE
);

-- Mock data
INSERT INTO stocks (product_id, quantity) VALUES
(1, 10),
(2, 50),
(3, 100),
(4, 0);
```

Résultat final, les tests passent le CI via le runner:

The screenshot shows a CI pipeline interface for a project named "Deployment_Lab3". A specific job, "ci fix #17", is highlighted in blue. The job status is "succeeded 1 minute ago in 51s". The pipeline step "deploy" is selected, showing its detailed execution history:

Step	Description	Duration
> 1	Set up job	2s
> 2	Checkout depot	1s
> 3	Creer fichier .env	0s
> 4	Cleanup complet	13s
> 5	Demarrer les services	15s
> 6	Wait MySQL	2s
> 7	Attendre que les tables soient prêtes	11s
> 8	Executer les tests de validation	3s
> 9	Status	0s
> 10	Post Checkout depot	0s
> 11	Complete job	0s

On the right side of the interface, there is a search bar labeled "Search logs" and some additional icons.