ECOLE MAROCAINE DES SCIENCES DE L'INGENIEUR
Membre de HONORIS UNITED UNIVERSITIES

Compt Rendu

5ème année
Ingénieur Informatique & Réseaux

TP4

GraphQL

Réalisé par : *IMANE Chakrellah*
YASSINE Ech-chaoui

Prof : *Pr.Oussama*
Classe: *5IIR-11*

2025-2026

Table des matières

Tout les TPs Réalisés ce trouve dans le lien suivant qui contient le projet + CR :

https://github.com/CHAKRELLAH44/JEE-ARCHITECTURE.git

Objectif du TP

Ce TP a pour objectif de comprendre le fonctionnement de GraphQL en développant un service web bancaire basé sur Spring Boot.
L'étudiant doit :

- Comprendre la différence fondamentale entre REST et GraphQL.
- Créer un schéma GraphQL et des résolveurs (Query + Mutation).
- Mapper les entités, DTO, services et repositories.
- Gérer les transferts bancaires et les contraintes métiers.
- Utiliser GraphiQL pour tester toutes les opérations.
- Gérer les exceptions GraphQL de manière propre.

1. Architecture du projet

1.1 Structure du projet

L'architecture du TP contient les couches suivantes :

- **model (service.model)** : contient les entités JPA (Customer, User, BankAccount…).
- **dtos/** : données échangées avec le client GraphQL.
- **dao/** : interfaces Repository Spring Data JPA.
- **service/** : logique métier (virement, création client…).
- **presentation/** : contrôleurs GraphQL (QueryMapping, MutationMapping).
- **config/** : configuration de ModelMapper.
- **common/** : outils généraux (conversion dates).
- **resources/graphql/** : fichier **schema.graphqls** contenant Query & Mutation.

1.2 Description des composants

- **Entities (BO)** : représentent la base de données (User, Customer, BankAccount…).
- **DTOs** : ce que GraphQL renvoie au client.
- **Repositories** : accès H2 via JPA.
- **Services métier** :
  - Vérification des règles bancaires.
  - Gestion des identités uniques.
  - Gestion des soldes et statuts des comptes.
- **Controllers GraphQL** : exposent les opérations via Query & Mutation.
- **Schema.graphqls** : cœur du fonctionnement GraphQL (types, champs, inputs, enums).
- **ExceptionHandler GraphQL** : personnalise les erreurs renvoyées au client.

1.3 Rôle de chaque module

- **GraphQL schema** → définit tout ce qu'un client peut demander.
- **Resolvers (QueryMapping / MutationMapping)** → exécutent la logique.
- **Service** → applique les règles métiers (vérifications, calculs…).

- **Repository** → récupère et sauvegarde les données en H2.
- **ModelMapper** → convertit automatiquement BO vers DTO.

Le workflow GraphQL est :

Client GraphQL → Schema → Resolver → Service → Repository → Service → Resolver → Client

2. Notions importantes

2.1 Concepts théoriques

- **GraphQL** : langage de requête permettant au client de demander exactement les données qu'il veut.
- **Query** : équivalent GET, pour lire les données.
- **Mutation** : équivalent POST/PUT/DELETE, pour modifier les données.
- **Schema GraphQL** : décrit les types, inputs, enums et services exposés.
- **Resolver** : fonction qui répond à un champ du schéma.
- **Relations JPA** : OneToMany, ManyToOne entre clients, comptes, transactions.

2.2 Technologies utilisées

- **Spring Boot 3**
- **Spring GraphQL**
- **Spring Data JPA**
- **ModelMapper**
- **Base H2**
- **GraphiQL Explorer** (interface de test GraphQL)
- **Lombok**

2.3 Notions clés

- **Chaque requête GraphQL est POST**
- **Mapping BO ↔ DTO automatique via ModelMapper**
- **GraphQL utilise un seul endpoint** : `/graphql`
- **Respects des règles métier (statut compte, solde, identité unique)**
- **Gestion personnalisée des exceptions GraphQL**

3. Déroulement du TP

3.1 Étapes réalisées

1. Création du projet Maven + ajout des dépendances (GraphQL, JPA, H2).
2. Création des **Enums** (AccountStatus, TransactionType).
3. Création des entités JPA :
    - User, Customer, BankAccount, BankAccountTransaction.

4. Création des classes utilitaires (CommonTools).
5. Configuration de **ModelMapper** (mapping dates + conversions).
6. Création des **DTOs** : CustomerDto, BankAccountDto, TransactionDto…
7. Création des **repositories** JPA.
8. Implémentation des services métier :
    - Création clients
    - Création comptes
    - Virements
    - Consultation transactions
9. Initialisation de la base via CommandLineRunner.
10. Mise en place du fichier **schema.graphqls** (Query + Mutation).
11. Création des contrôleurs GraphQL :
    - CustomerGraphqlController
    - BankAccountGraphqlController
    - TransactionGraphqlController
12. Tests complets avec **GraphiQL** (customers, bankAccounts, mutations…).
13. Gestion des erreurs via GraphQlExceptionHandler.

4. Code source

4.1 Extraits essentiels

- **schema.graphqls** (types, données renvoyées, mutations).
- **BankAccountServiceImpl.wiredTransfer** :
    - Vérifie solde
    - Vérifie statut
    - Enregistre les transactions
- **GraphQlExceptionHandler** : renvoie messages d'erreur personnalisés.

4.2 Workflow détaillé (exemple d'une mutation)

Mutation :

```
mutation {
  addBankAccount(dto: {
    rib: "RIB_10",
    amount: 5000,
    customerIdentityRef: "A100"
  }) {
    message
    rib
  }
}
```

Fonctionnement :

1. Le client envoie la mutation à `/graphql`.
2. Spring GraphQL lit le schéma pour vérifier la validité.
3. Le resolver `addBankAccount()` est appelé.
4. Le service vérifie :

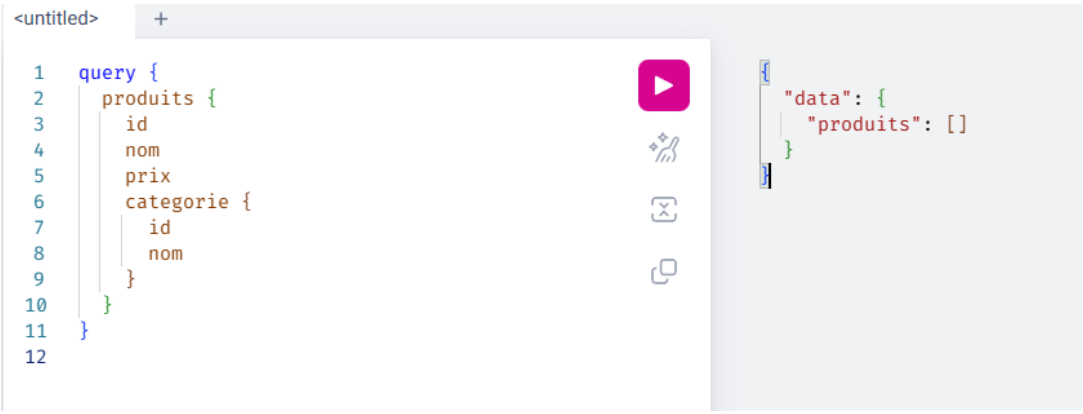
    - si le client existe
    - si le RIB est unique
5. Le BO est sauvegardé dans H2.
6. ModelMapper convertit BO → DTO.
7. Le DTO est renvoyé au client.

4.3 Rôle des principales classes

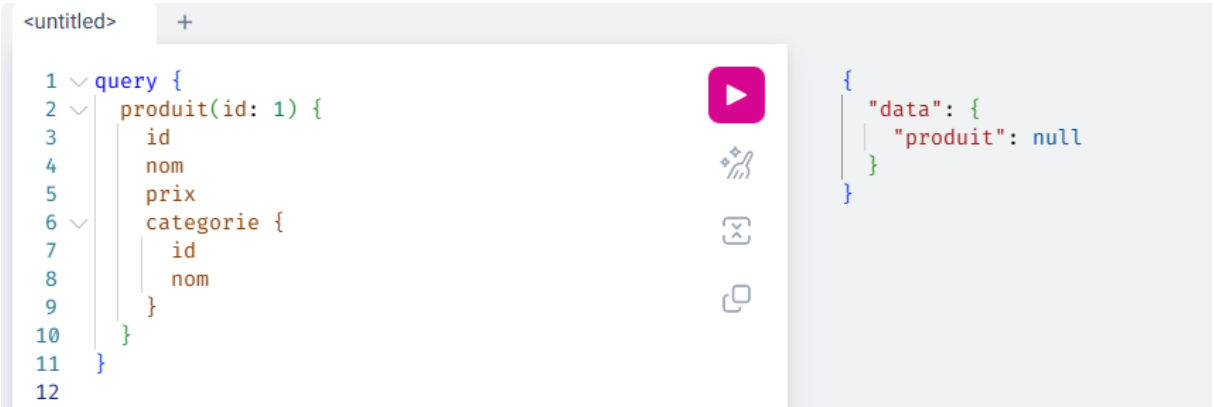
- **CustomerServiceImpl** : gère identités, mises à jour, suppression.
- **BankAccountServiceImpl** : création des comptes et validation du RIB.
- **TransactionServiceImpl** :
    - applique règles métier (solde suffisant, compte fermé/bloqué…).
    - enregistre les transactions.
- **GraphQl Controllers** : exposent les services.
- **GraphQlExceptionHandler** : capture BusinessException et renvoie un message lisible.

5. Tests GraphQL (GraphiQL)

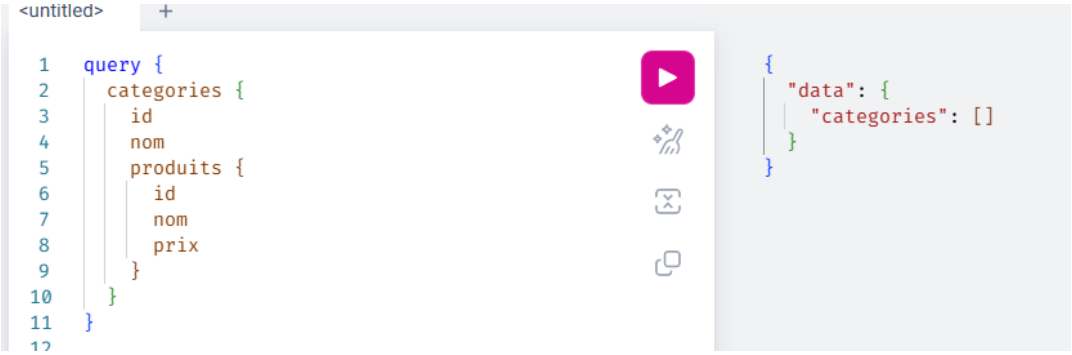
1. QUERY : récupérer tous les produits



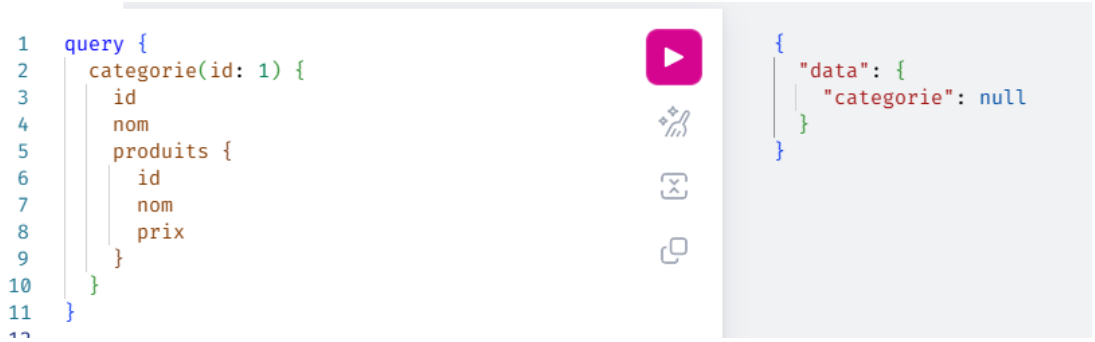
2. QUERY : récupérer un produit par ID (inexistant)



3. QUERY : récupérer toutes les catégories

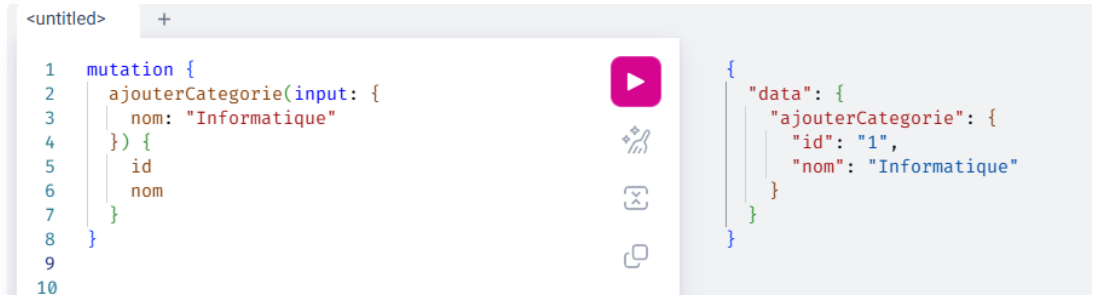


4. QUERY : récupérer une catégorie par ID

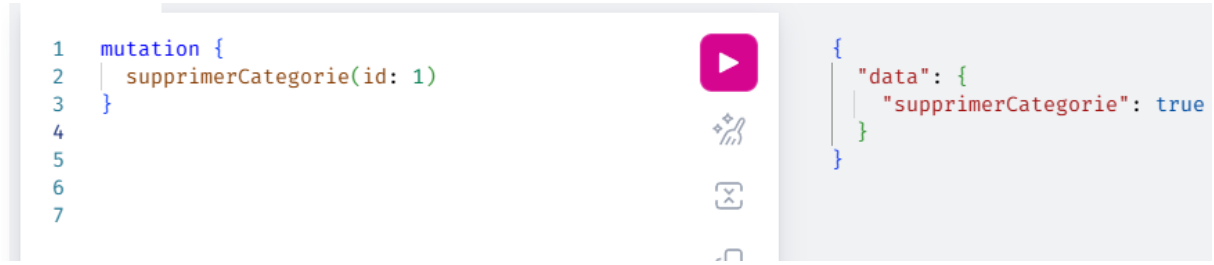


MUTATIONS

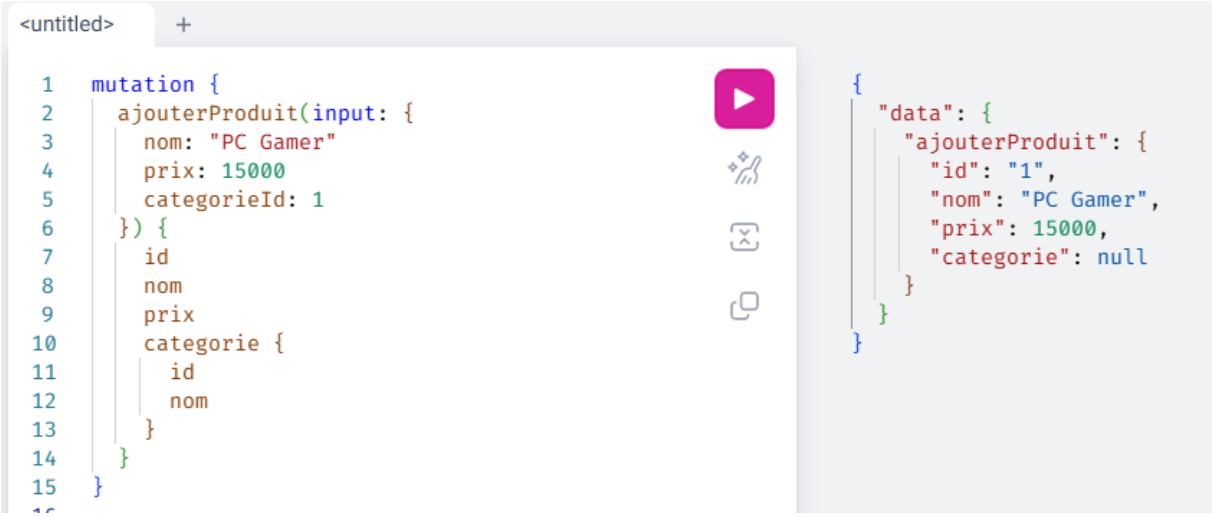
5. Mutation : ajouter une catégorie



6. Mutation : supprimer une catégorie



7. Mutation : ajouter un produit



```
mutation {
  ajouterProduit(input: {
    nom: "PC Gamer"
    prix: 15000
    categorieId: 1
  }) {
    id
    nom
    prix
    categorie {
      id
      nom
    }
  }
}
```

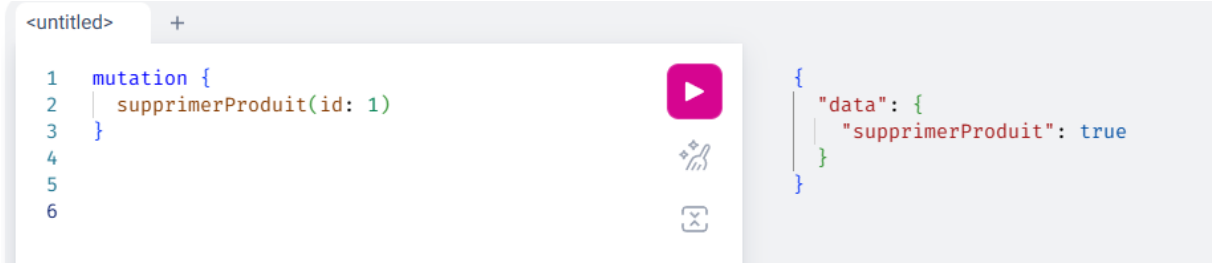
```
{
  "data": {
    "ajouterProduit": {
      "id": "1",
      "nom": "PC Gamer",
      "prix": 15000,
      "categorie": null
    }
  }
}
```

8. Mutation : modifier un produit

```
mutation {
  modifierProduit(id: 1, input: {
    nom: "PC Gamer Ultra"
    prix: 18000
    categorieId: 1
  }) {
    id
    nom
    prix
    categorie {
      id
      nom
    }
  }
}
```

```
{
  "data": {
    "modifierProduit": {
      "id": "1",
      "nom": "PC Gamer Ultra",
      "prix": 18000,
      "categorie": null
    }
  }
}
```

9. Mutation : supprimer un produit



```
mutation {
  supprimerProduit(id: 1)
}
```

```
{
  "data": {
    "supprimerProduit": true
  }
}
```

7. Outils de visualisation

- **GraphiQL Explorer**
- Permet :
    - Autocomplétion
    - Visualisation du schéma
    - Exécution interactive des requêtes
- Possibilité de voir la requête POST envoyée via l'onglet Network du navigateur.

8. Conclusion

Ce TP m'a permis d'acquérir une compréhension claire du fonctionnement d'un service GraphQL sous Spring Boot.
J'ai appris à :

- créer un schéma GraphQL complet
- exposer des Query et Mutation
- gérer une base H2 avec JPA
- appliquer des règles métiers complexes (virements bancaires)
- tester graphiquement grâce à GraphiQL
- gérer proprement les erreurs GraphQL

Ce TP constitue une étape essentielle pour comprendre les API modernes plus flexibles que REST.