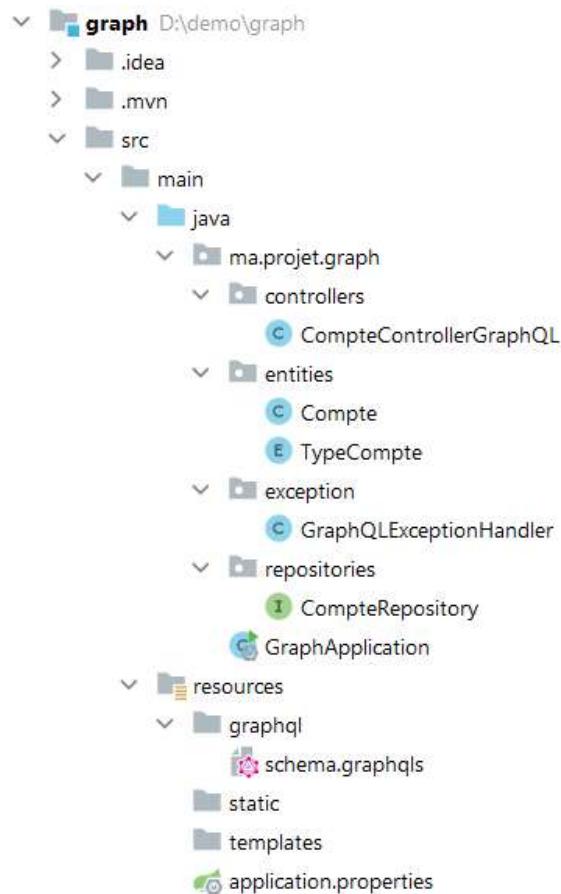


## 5.16 Activité pratique : Service GraphQL avec Spring Boot

### Etape 1 : Création du projet Spring Boot

1. Accéder à Spring Initializr, un outil en ligne permettant de générer des projets Spring Boot préconfigurés.
2. Renseigner les paramètres suivants :
  - **Project:** Sélectionner Maven Project.
  - **Language:** Choisir Java.
  - **Spring Boot Version:** Utiliser la version la plus récente compatible, comme 3.5.0.
  - **Group:** Saisir un nom de package, par exemple com.example.
  - **Artifact:** Donner un nom au projet, par exemple banque-service.
  - **Packaging:** Choisir Jar.
  - **Java Version:** Sélectionner 17 ou une version compatible avec votre JDK.
3. Ajouter les dépendances nécessaires :
  - Spring for GraphQL : permet de développer des APIs GraphQL en exploitant Spring Boot et GraphQL Java.
  - Spring Web : indispensable pour exposer des services web RESTful ou GraphQL.
  - Spring Data JPA : permet d'intégrer JPA pour la gestion des données relationnelles.
  - H2 Database : fournit une base de données légère et embarquée adaptée aux tests.
  - Lombok : réduit le code répétitif grâce à des annotations pour les getters, setters, constructeurs, etc.
4. Générer le projet et télécharger l'archive .zip. Extraire son contenu et importer le projet dans un IDE, tel qu'IntelliJ IDEA ou Eclipse.

La structure finale du projet est la suivante :



**Figure 5.14: Structure finale du projet.**

### Etape 2 : Configuration des propriétés de l'application

1. Ouvrir le fichier src/main/resources/application.properties.
2. Ajouter les configurations suivantes :

---

```
spring.datasource.url=jdbc:h2:mem:banque
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
server.port=8082
spring.jpa.hibernate.ddl-auto=update
spring.graphql.graphiql.enabled=true
```

---



Ces configurations assurent le bon fonctionnement du projet :

- `spring.datasource.url`: définit l'URL de la base H2 en mémoire. Cela signifie que les données sont volatiles et disparaissent à chaque redémarrage.
- `spring.datasource.driverClassName`: spécifie le pilote JDBC à utiliser pour H2.
- `spring.datasource.username` et `spring.datasource.password`: fournissent les identifiants par défaut (sa sans mot de passe).
- `spring.jpa.database-platform`: configure le dialecte Hibernate adapté à H2.
- `spring.h2.console.enabled`: active une console accessible via /h2-console pour explorer les données.
- `server.port`: modifie le port du serveur à 8082.
- `spring.jpa.hibernate.ddl-auto`: permet de créer ou de mettre à jour les tables automatiquement selon les entités JPA.
- `spring.graphql.graphiql.enabled`: active GraphiQL, une interface visuelle pour exécuter des requêtes GraphQL.

### Explications supplémentaires sur les dépendances ajoutées

- **Spring for GraphQL**: fournit des annotations comme `@QueryMapping` et `@MutationMapping`, permettant de créer facilement des APIs GraphQL.
- **Spring Web**: utilise le serveur web intégré Tomcat pour héberger les APIs.
- **Spring Data JPA**: facilite les opérations CRUD sur la base grâce à des interfaces comme `JpaRepository`.
- **H2 Database**: idéal pour un environnement de développement ou de test sans nécessiter une base externe.
- **Lombok**: supprime la nécessité d'écrire manuellement des méthodes comme `getters`, `setters`, ou des constructeurs, ce qui simplifie le code des entités.

### Vérification de la configuration

1. Démarrer le projet en exécutant la classe principale annotée avec `@SpringBootApplication`.
2. Accéder à la console H2 à l'URL suivante : <http://localhost:8082/h2-console>.
3. Vérifier que la connexion à la base est possible en entrant les informations suivantes :
  - **JDBC URL**: `jdbc:h2:mem:banque`.
  - **Username**: `sa`.
  - **Password**: laisser vide.
4. L'interface GraphiQL est également accessible à l'adresse : <http://localhost:8082/graphiql>.

### Etape 3 : Définition des entités

#### Classe Compte

La classe Compte représente une table dans la base de données, où chaque instance correspond à une ligne de la table. Voici sa définition complète avec les annotations JPA (Java Persistence API) :

---

```

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Compte {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private double solde;

    @Temporal(TemporalType.DATE)
    private Date dateCreation;

    @Enumerated(EnumType.STRING)
    private TypeCompte type;
}

```

---



- L'annotation `@Entity` indique que cette classe représente une table dans la base de données.
- `@Id` et `@GeneratedValue` gèrent la clé primaire.
- `@Temporal` indique que l'attribut `dateCreation` sera une date.
- `@Enumerated` indique que l'attribut `type` est une énumération.

### Enumération TypeCompte

L'énumération `TypeCompte` définit les différents types de comptes que l'application peut gérer.

---

```

public enum TypeCompte {
    COURANT, EPARGNE
}

```

---



L'énumération `TypeCompte` définit deux types de comptes possibles : `COURANT` et `EPARGNE`.

### Etape 4 : Définition du schéma GraphQL

Le schéma GraphQL constitue la structure principale de l'API. Il définit les types de données, les requêtes (Query) et les mutations (Mutation) accessibles aux utilisateurs.

#### Schéma GraphQL complet

Le schéma suivant, écrit en SDL (*Schema Definition Language*), présente tous les types et opérations disponibles dans le service GraphQL :

---

```

enum TypeCompte {
    COURANT
    EPARGNE
}

type Query {
    allComptes: [Compte]
    compteById(id : ID):Compte
    totalSolde: SoldeStats
}

```

---

```

type Compte {
    id: ID
    solde: Float
    dateCreation: String
    type: TypeCompte
}

type Mutation {
    saveCompte(compte:CompteRequest):Compte
}

input CompteRequest {
    solde: Float
    dateCreation: String
    type: TypeCompte
}

type SoldeStats {
    count: Int
    sum: Float
    average: Float
}

```

---

## Description détaillée des éléments du schéma

### Types principaux

- TypeCompte : une énumération représentant les types possibles de comptes. Elle permet de restreindre les valeurs possibles à :
  - COURANT : pour les comptes utilisés dans les opérations quotidiennes.
  - EPARGNE : pour les comptes d'épargne.
- Compte : un type représentant un compte bancaire avec les champs suivants :
  - id (ID) : identifiant unique du compte.
  - solde (Float) : solde du compte, exprimé en nombre décimal.
  - dateCreation (String) : date de création du compte, représentée sous forme de chaîne.
  - type (TypeCompte) : type du compte, basé sur l'énumération TypeCompte.
- SoldeStats : un type contenant des statistiques sur les soldes des comptes, comprenant :
  - count (Int) : nombre total de comptes.
  - sum (Float) : somme totale des soldes.
  - average (Float) : moyenne des soldes.

### Opérations définies dans le schéma

- **Requêtes (Query) :**
  - allComptes : retourne une liste de tous les comptes sous forme d'objets Compte.
  - compteById(id : ID) : retourne un compte spécifique identifié par son id.
  - totalSolde : retourne un objet SoldeStats contenant les statistiques globales sur les comptes.
- **Mutations (Mutation) :**
  - saveCompte(compte: CompteRequest) : permet de créer ou de modifier un compte en prenant un objet CompteRequest comme paramètre.

### Type d'entrée (*input type*)

- CompteRequest : un type d'entrée utilisé pour les mutations. Il contient les champs suivants :
  - solde (Float) : solde initial ou mis à jour du compte.

- dateCreation (String) : date de création du compte.
- type (TypeCompte) : type du compte (COURANT ou EPARGNE).

### Utilisation et flux d'interactions avec le schéma

Voici un exemple de flux interactif pour exploiter ce schéma :

- L'utilisateur exécute une requête pour récupérer un compte spécifique par son id :

---

```
query {
  compteById(id: 1) {
    id
    solde
    type
  }
}
```

---

- Le serveur GraphQL exécute cette requête et interroge la base de données pour retourner les données correspondantes sous la forme suivante :

```
{
  "data": {
    "compteById": {
      "id": 1,
      "solde": 1500.0,
      "type": "COURANT"
    }
  }
}
```

- Lors de la création d'un compte, une mutation est utilisée. Exemple :

---

```
mutation {
  saveCompte(compte: {
    solde: 2000.0,
    dateCreation: "2024-11-18",
    type: EPARGNE
  }) {
    id
    solde
    type
  }
}
```

---

- En réponse, le serveur retourne les informations du compte nouvellement créé :

```
{
  "data": {
    "saveCompte": {
      "id": 2,
      "solde": 2000.0,
      "type": "EPARGNE"
    }
  }
}
```



Ce schéma présente plusieurs avantages :

- **Flexibilité** : les requêtes GraphQL permettent de demander uniquement les données nécessaires.
- **Modularité** : les requêtes (**Query**) et les mutations (**Mutation**) sont bien séparées, facilitant l'ajout de nouvelles fonctionnalités.
- **Validation automatique** : les énumérations (**TypeCompte**) et les types d'entrée (**CompteRequest**) imposent des contraintes sur les données, réduisant les risques d'erreurs.
- **Centralisation des statistiques** : le type **SoldeStats** permet de calculer des statistiques globales avec une seule requête, évitant des appels multiples ou des calculs côté client.

### Étape 5 : Création du contrôleur GraphQL

Le contrôleur GraphQL est une classe qui gère les requêtes et les mutations définies dans le schéma GraphQL. Il agit comme un point d'entrée pour les utilisateurs, en mappant les requêtes aux fonctions spécifiques qui interagissent avec la couche de persistance.

Voici la définition complète du contrôleur **CompteControllerGraphQL** :

---

```

@Controller
@AllArgsConstructor
public class CompteControllerGraphQL {
    private CompteRepository compteRepository;

    @QueryMapping
    public List<Compte> allComptes(){
        return compteRepository.findAll();
    }

    @QueryMapping
    public Compte compteById(@Argument Long id){
        Compte compte = compteRepository.findById(id).orElse(null);
        if(compte == null) throw new RuntimeException(String.format("Compte %s not
            found", id));
        else return compte;
    }

    @MutationMapping
    public Compte saveCompte(@Argument Compte compte){
        return compteRepository.save(compte);
    }

    @QueryMapping
    public Map<String, Object> totalSolde() {
        long count = compteRepository.count();
        double sum = compteRepository.sumSolde();
        double average = count > 0 ? sum / count :0;

        return Map.of(
            "count", count,
            "sum", sum,
            "average", average
        );
    }
}

```

---

### Explications détaillées des annotations et méthodes

- `@Controller` : cette annotation indique que la classe est un composant Spring chargé de gérer les requêtes GraphQL.
- `@AllArgsConstructor` : générée par Lombok, cette annotation crée un constructeur prenant en argument tous les champs de la classe. Ici, elle permet d'injecter automatiquement une instance de `CompteRepository`.
- `@QueryMapping` : utilisée pour mapper une méthode Java à une requête (Query) GraphQL. Les requêtes définies dans le schéma sont liées aux méthodes portant cette annotation.
- `@MutationMapping` : utilisée pour mapper une méthode Java à une mutation (Mutation) GraphQL.
- `@Argument` : indique qu'un paramètre de méthode correspond à un argument passé par le client dans la requête GraphQL.

### Description des méthodes

1. `allComptes() : List<Compte>` :

- Cette méthode retourne une liste de tous les comptes stockés dans la base de données.
- Elle utilise la méthode `findAll()` du dépôt JPA (`CompteRepository`).
- Mappée à la requête `allComptes` définie dans le schéma.

2. `compteById(Long id) : Compte` :

- Cette méthode retourne un compte unique correspondant à l'identifiant (`id`) fourni.
- Si aucun compte n'est trouvé, une exception est levée avec un message personnalisé.
- Mappée à la requête `compteById` définie dans le schéma.
- Exemple d'utilisation :

---

```
query {
  compteById(id: 1) {
    id
    solde
    type
  }
}
```

---

3. `saveCompte(Compte compte) : Compte` :

- Cette méthode insère un nouveau compte ou met à jour un compte existant.
- Elle utilise la méthode `save()` du dépôt JPA.
- Le paramètre `Compte compte` est fourni par le client sous forme d'un objet GraphQL (type `CompteRequest`).
- Mappée à la mutation `saveCompte`.
- Exemple d'utilisation :

---

```
mutation {
  saveCompte(compte: {
    solde: 2000.0,
    dateCreation: "2024-11-18",
    type: COURANT
  }) {
    id
    solde
    type
  }
}
```

---

4. `totalSolde() : Map<String, Object>` :

- Cette méthode retourne des statistiques globales sur les comptes : nombre total, somme des soldes, et moyenne des soldes.
- Les statistiques sont calculées à l'aide des méthodes `count()` et `sumSolde()` du dépôt JPA (`CompteRepository`).
- Les résultats sont retournés sous forme de clé-valeur (`Map.of()`).
- Mappée à la requête `totalSolde`.
- Exemple d'utilisation :

---

```
query {
    totalSolde {
        count
        sum
        average
    }
}
```

---

- Exemple de réponse :

```
{
  "data": {
    "totalSolde": {
      "count": 5,
      "sum": 10000.0,
      "average": 2000.0
    }
  }
}
```

### Résumé des fonctionnalités

- `allComptes()` : permet d'afficher tous les comptes existants.
- `compteById(Long id)` : permet de rechercher un compte spécifique par son identifiant.
- `saveCompte(Compte compte)` : ajoute ou met à jour un compte dans la base.
- `totalSolde()` : calcule et retourne des statistiques agrégées sur les comptes.



Ce contrôleur GraphQL illustre les principes fondamentaux de l'API GraphQL :

- Une séparation claire entre les requêtes (`Query`) et les mutations (`Mutation`).
- Une simplicité d'interaction grâce aux annotations Spring Boot et aux méthodes de dépôt JPA.
- Une gestion facile des données grâce à une couche de persistance bien intégrée.

### Étape 6 : Gestion des exceptions GraphQL

La gestion des exceptions dans un service GraphQL est essentielle pour fournir des messages d'erreur clairs et utiles aux utilisateurs. Cette étape explique comment personnaliser la gestion des exceptions dans Spring Boot avec GraphQL.

#### Définition de la classe `GraphQLExceptionHandler`

La classe suivante étend `DataFetcherExceptionResolverAdapter` pour intercepter et personnaliser les erreurs déclenchées lors des requêtes GraphQL.

---

```
@Component
public class GraphQLExceptionHandler extends DataFetcherExceptionResolverAdapter
{
    @Override
```

```

protected GraphQLError resolveToSingleError(Throwable ex,
    DataFetchingEnvironment env) {
    return new GraphQLError() {
        @Override
        public String getMessage() {
            return ex.getMessage();
        }

        @Override
        public List<SourceLocation> getLocations() {
            return null;
        }

        @Override
        public ErrorClassification getErrorType() {
            return null;
        }
    };
}
}

```

---

### Explications détaillées des annotations et de la classe

- `@Component` :
  - Cette annotation marque la classe comme un composant Spring. Elle sera automatiquement détectée et gérée par le conteneur Spring.
  - Cela permet à Spring de l'enregistrer comme un résolveur d'erreurs pour les requêtes GraphQL.
- `DataFetcherExceptionResolverAdapter` :
  - Classe de base fournie par GraphQL Spring Boot pour personnaliser la gestion des exceptions.
  - Elle offre une méthode `resolveToSingleError` que l'on peut redéfinir pour retourner un objet `GraphQLError`.
- `resolveToSingleError(Throwable ex, DataFetchingEnvironment env)` :
  - Cette méthode est appelée lorsqu'une exception est levée dans le processus de résolution des requêtes GraphQL.
  - Elle prend deux paramètres :
    - \* `Throwable ex` : l'exception qui a été levée.
    - \* `DataFetchingEnvironment env` : contient le contexte d'exécution, comme les arguments de la requête.
  - La méthode retourne un objet anonyme implémentant `GraphQLError`.
- `GraphQLError` :
  - Interface qui définit la structure des erreurs retournées au client.
  - Les trois méthodes redéfinies dans cet exemple sont :
    - \* `getMessage()` : retourne le message d'erreur à afficher. Ici, il reprend le message de l'exception levée.
    - \* `getLocations()` : retourne les emplacements dans la requête où l'erreur s'est produite (ici, `null` pour simplifier).
    - \* `getErrorType()` : retourne le type d'erreur (ici, `null` pour simplifier).

### Exemple de fonctionnement

1. Lorsqu'une méthode de résolution d'une requête GraphQL lève une exception, comme dans le cas de `compteById` :

---

```
if (compte == null) {
    throw new RuntimeException(String.format("Compte %s not found", id));
}
```

---

2. L'exception est interceptée par GraphQLExceptionHandler.
3. La méthode resolveToSingleError est appelée pour créer un objet GraphQLError.
4. Le message d'erreur est renvoyé au client GraphQL sous la forme suivante :

```
{
  "errors": [
    {
      "message": "Compte 42 not found",
      "locations": null,
      "path": null
    }
  ]
}
```

### Avantages de cette gestion des erreurs

- **Clarté** : les utilisateurs reçoivent des messages d'erreur explicites, adaptés à leur requête.
- **Personnalisation** : chaque type d'erreur peut être formaté selon les besoins spécifiques du projet.
- **Sécurité** : en masquant les détails internes de l'exception, on évite de divulguer des informations sensibles sur l'application.



Cette classe permet une personnalisation complète de la gestion des erreurs dans les requêtes GraphQL, en assurant des retours d'erreur compréhensibles pour les clients tout en facilitant le débogage côté serveur.

### Etape 7 : Tests avec GraphiQL

Cette section explique comment tester les fonctionnalités de l'API GraphQL à l'aide de l'interface GraphiQL, qui permet d'exécuter des requêtes et des mutations de manière interactive.

#### Accéder à l'interface GraphiQL

1. Lancer l'application Spring Boot.
2. Ouvrir un navigateur web et accéder à l'URL suivante : <http://localhost:8082/graphiql>.
3. Une interface utilisateur s'affiche, permettant de saisir et d'exécuter des requêtes GraphQL.

#### Exécuter des requêtes et mutations

Pour tester les fonctionnalités, saisir les requêtes et mutations suivantes dans l'interface GraphiQL.

#### Requête 1 : Récupérer tous les comptes

Cette requête permet de récupérer la liste de tous les comptes enregistrés dans la base de données.

---

```
query {
  allComptes {
    id
    solde
    dateCreation
    type
  }
}
```

---

The screenshot shows a GraphQL playground interface. On the left, there is a code editor containing the following GraphQL query:

```

1 * query {
2   allComptes {
3     id
4     solde
5     dateCreation
6     type
7   }
8 }

```

On the right, the results of the query are displayed as a JSON object. The response shows three accounts (allComptes) with their respective details:

```

{
  "data": {
    "allComptes": [
      {
        "id": "1",
        "solde": 8271.796491312965,
        "dateCreation": "2024-11-18",
        "type": "EPARGNE"
      },
      {
        "id": "2",
        "solde": 3672.765137493308,
        "dateCreation": "2024-11-18",
        "type": "COURANT"
      },
      {
        "id": "3",
        "solde": 422.171034829543,
        "dateCreation": "2024-11-18",
        "type": "EPARGNE"
      }
    ]
  }
}

```

Below the code editor, there are tabs for "Variables" and "Headers".

- **Explication :**

- Le mot-clé `query` est utilisé pour exécuter une requête GraphQL.
- Le champ `allComptes` correspond à la méthode `allComptes()` définie dans le contrôleur GraphQL.
- Les champs `id`, `solde`, `dateCreation`, et `type` sont les données demandées pour chaque compte.

- **Exemple de réponse :**

```
{
  "data": {
    "allComptes": [
      {
        "id": 1,
        "solde": 1500.0,
        "dateCreation": "2024-11-18",
        "type": "COURANT"
      },
      {
        "id": 2,
        "solde": 3000.0,
        "dateCreation": "2024-11-17",
        "type": "EPARGNE"
      }
    ]
  }
}
```

### Requête 2 : Récupérer un compte par identifiant (`id`)

Cette requête permet de récupérer un compte spécifique en utilisant son `id`.

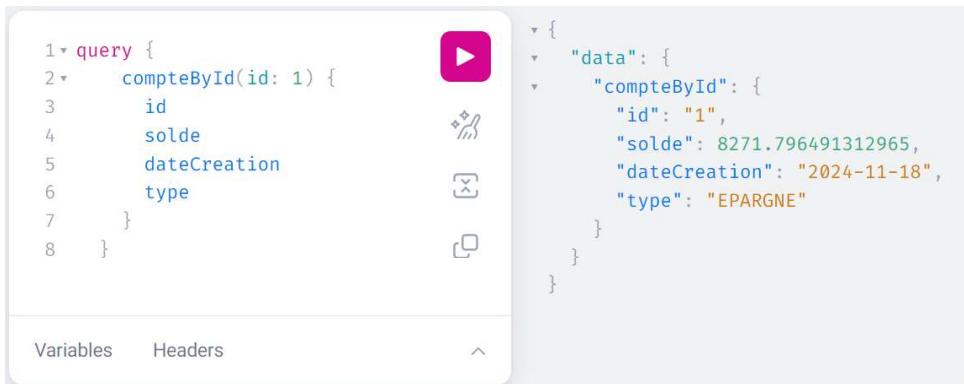
---

```
query {
  compteById(id: 1) {
    id
    solde
  }
}
```

```

    dateCreation
    type
}
}

```



The screenshot shows a GraphQL playground interface. On the left, there is a code editor containing the following GraphQL query:

```

1 * query {
2   *   compteById(id: 1) {
3     id
4     solde
5     dateCreation
6     type
7   }
8 }

```

On the right, there is a JSON response window showing the results of the query. The response is:

```

{
  "data": {
    "compteById": {
      "id": "1",
      "solde": 8271.796491312965,
      "dateCreation": "2024-11-18",
      "type": "EPARGNE"
    }
  }
}

```

Below the code editor, there are tabs for "Variables" and "Headers".

- **Explication :**

- Le champ `compteById` prend un argument `id`.
- Cet argument est utilisé pour rechercher un compte spécifique dans la base de données.
- Les champs demandés (`id`, `solde`, `dateCreation`, `type`) sont retournés pour le compte correspondant.

- **Exemple de réponse :**

```
{
  "data": {
    "compteById": {
      "id": 1,
      "solde": 1500.0,
      "dateCreation": "2024-11-18",
      "type": "COURANT"
    }
  }
}
```

- **En cas d'erreur :** Si l'identifiant ne correspond à aucun compte, une erreur est renvoyée :

```
{
  "errors": [
    {
      "message": "Compte 42 not found",
      "locations": null,
      "path": ["compteById"]
    }
  ]
}
```

### Requête avec paramètre

Une requête GraphQL peut inclure des paramètres dynamiques pour faciliter la recherche ou le filtrage des données. Cette section présente l'intégration d'une requête avec un paramètre.

#### Requête GraphQL avec paramètre :

---

```
query($id: ID) {
  compteById(id: $id) {
    id
    type
  }
}
```

---

- \$id est une variable passée comme paramètre à la requête.
- compteById est le champ correspondant à la méthode définie dans le contrôleur GraphQL.
- Les champs demandés (id, type) sont ceux à retourner pour le compte identifié.

**Exemple d'exécution avec GraphiQL :**

1. Dans l'interface GraphiQL, saisir la requête suivante :

---

```
query($id: ID) {
  compteById(id: $id) {
    id
    type
  }
}
```

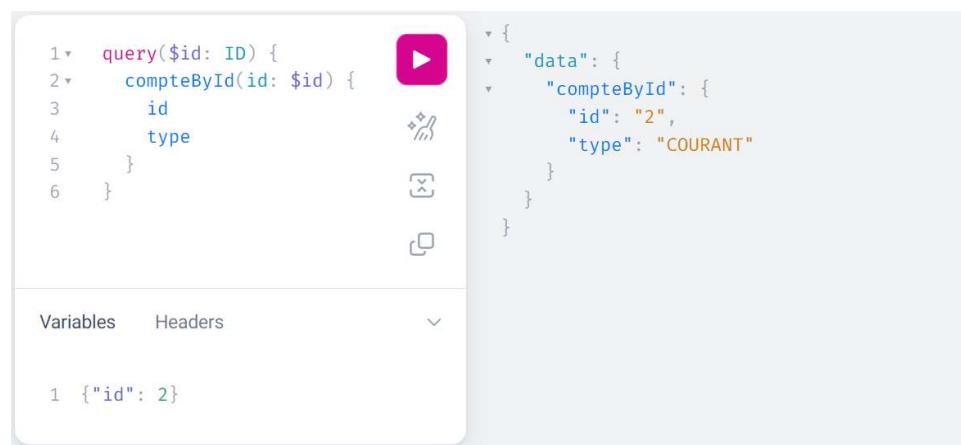
---

2. Passer la variable \$id dans un objet JSON, comme suit :

```
{
  "id": 1
}
```

**Exemple de réponse pour id = 1 :**

```
{
  "data": {
    "compteById": {
      "id": 1,
      "type": "COURANT"
    }
  }
}
```



**Explication des étapes**

- La requête utilise une variable (\$id) pour rendre l'appel plus flexible.

- Dans GraphiQL, les variables sont définies sous forme de JSON dans un espace dédié sous la requête.
- Lors de l'exécution, la valeur de \$id est injectée dans la requête pour rechercher le compte correspondant.

### Avantages de l'utilisation des paramètres

- Permet de rendre les requêtes réutilisables sans modifier le code source.
- Simplifie les tests en passant différents paramètres sans changer la structure de la requête.
- Améliore la lisibilité et la modularité du code GraphQL.

### Mutation 1 : Ajouter un nouveau compte

Cette mutation permet de créer un nouveau compte dans la base de données.

```
mutation {
  saveCompte(compte: {
    soldes: 1500.0,
    dateCreation: "2024-11-18",
    type: COURANT
  }) {
    id
    soldes
    type
  }
}
```

The screenshot shows the GraphiQL interface with two panes. The left pane contains the GraphQL mutation code:

```
1 mutation {
2   saveCompte(compte: {
3     soldes: 1500,
4     type: COURANT
5   }) {
6     id
7     soldes
8     type
9   }
10 }
```

The right pane shows the JSON response from the mutation:

```
{
  "data": {
    "saveCompte": {
      "id": "5",
      "soldes": 1500,
      "type": "COURANT"
    }
  }
}
```

Below the panes, there are tabs for "Variables" and "Headers".

- **Explication :**
  - Le mot-clé `mutation` est utilisé pour indiquer que cette opération modifie les données.
  - Le champ `saveCompte` prend un objet de type `CompteRequest`.
  - Les champs définis dans l'objet (`soldes`, `dateCreation`, `type`) sont utilisés pour créer le compte.
  - Les champs retournés (`id`, `soldes`, `type`) permettent de vérifier que l'opération a réussi.
- **Exemple de réponse :**

```
{
  "data": {
    "saveCompte": {
      "id": 3,
      "soldes": 1500.0,
```

```

        "type": "COURANT"
    }
}
}
}
```

### Requête 3 : Calculer des statistiques sur les comptes

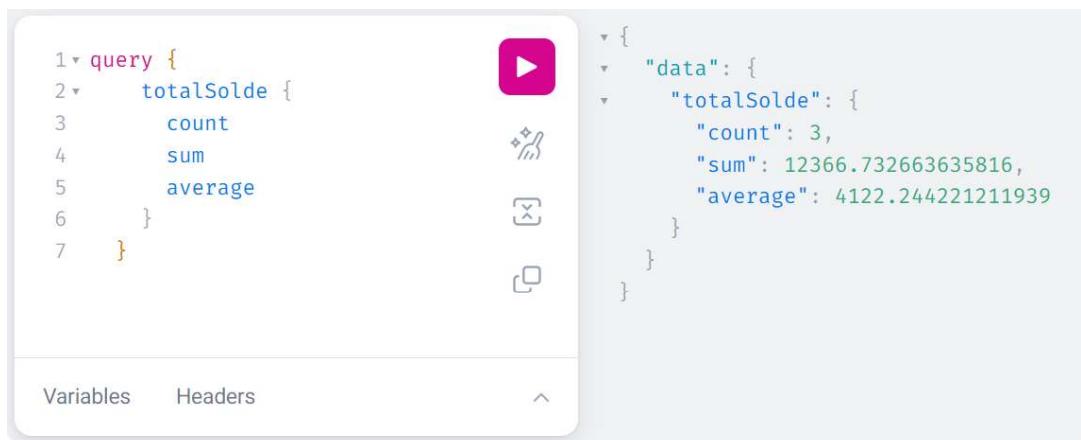
Cette requête permet d'obtenir des statistiques globales sur les soldes des comptes.

---

```

query {
  totalSolde {
    count
    sum
    average
  }
}
```

---



The screenshot shows a GraphQL playground interface. On the left, there is a code editor with the following query:

```

1 query {
2   totalSolde {
3     count
4     sum
5     average
6   }
7 }
```

On the right, there is a JSON response with a tree view:

```

{
  "data": {
    "totalSolde": {
      "count": 3,
      "sum": 12366.732663635816,
      "average": 4122.244221211939
    }
  }
}
```

Below the code editor, there are tabs for "Variables" and "Headers".

- **Explication :**

- Le champ `totalSolde` retourne un objet contenant des statistiques calculées sur tous les comptes.
- Les champs demandés incluent :
  - \* `count` : le nombre total de comptes.
  - \* `sum` : la somme totale des soldes.
  - \* `average` : la moyenne des soldes.

- **Exemple de réponse :**

```
{
  "data": {
    "totalSolde": {
      "count": 3,
      "sum": 6000.0,
      "average": 2000.0
    }
  }
}
```

### Résumé des tests

- La requête `allComptes` vérifie que tous les comptes sont correctement enregistrés et récupérables.

- La requête `compteById` permet de s'assurer qu'un compte spécifique peut être retrouvé à partir de son `id`.
- La mutation `saveCompte` valide l'ajout ou la mise à jour des comptes.
- La requête `totalSolde` calcule des statistiques précises sur les soldes des comptes.



Ces tests permettent de valider le bon fonctionnement de l'API GraphQL et de garantir que les données sont manipulées conformément aux attentes.

### Etape 8 : Ajout de la gestion des transactions

Cette section enrichit l'application en intégrant la gestion des transactions associées aux comptes bancaires. Chaque transaction représente une opération financière, telle qu'un dépôt ou un retrait, et est directement liée à un compte.

#### Objectifs

- Permettre l'ajout d'une transaction (dépôt ou retrait) à un compte existant.
- Afficher toutes les transactions associées à un compte donné.
- Calculer dynamiquement le solde d'un compte en fonction de ses transactions.
- Proposer des statistiques globales sur toutes les transactions (nombre total, somme des dépôts, somme des retraits).

#### Schéma GraphQL enrichi

Le schéma GraphQL doit inclure les nouvelles fonctionnalités suivantes :

- **Requêtes (Query) :**
  - `compteTransactions(id: ID)` : retourne toutes les transactions associées à un compte identifié par `id`.
  - `allTransactions` : retourne l'ensemble des transactions enregistrées.
  - `transactionStats` : retourne des statistiques globales sur les transactions (nombre total, somme des dépôts, somme des retraits).
- **Mutations (Mutation) :**
  - `addTransaction(transaction: TransactionRequest)` : permet d'ajouter une transaction à un compte existant.

#### Étapes de réalisation

##### 1. Crédation des entités nécessaires

Une entité `Transaction` représente chaque opération financière. Chaque transaction inclut les informations suivantes :

- Un montant.
- Une date.
- Un type (DEPOT ou RETRAIT).
- Un lien vers un compte existant.

##### 2. Implémentation des méthodes dans le contrôleur GraphQL

###### Ajout d'une transaction :

La méthode suivante ajoute une transaction à un compte identifié par `compteId` :

---

```
@MutationMapping
public Transaction addTransaction(@Argument TransactionRequest
    transactionRequest) {
    Compte compte = compteRepository.findById(transactionRequest.getCompteId())
        .orElseThrow(() -> new RuntimeException("Compte not found"));
    Transaction transaction = new Transaction();
    transaction.setMontant(transactionRequest.getMontant());
    transaction.setDate(transactionRequest.getDate());
```

---

```

        transaction.setType(transactionRequest.getType());
        transaction.setCompte(compte);
        transactionRepository.save(transaction);
        return transaction;
    }

```

---

**Affichage des transactions d'un compte :**

Cette méthode retourne la liste des transactions associées à un compte donné :

---

```

@QueryMapping
public List<Transaction> compteTransactions(@Argument Long id) {
    Compte compte = compteRepository.findById(id)
        .orElseThrow(() -> new RuntimeException("Compte not found"));
    return transactionRepository.findByCompte(compte);
}

```

---

**Calcul des statistiques globales sur les transactions :**

Les statistiques globales incluent le nombre total de transactions, la somme des dépôts, et la somme des retraits :

---

```

@QueryMapping
public Map<String, Object> transactionStats() {
    long count = transactionRepository.count();
    double sumDepots = transactionRepository.sumByType(TypeTransaction.DEPOT);
    double sumRetraits = transactionRepository.sumByType(TypeTransaction.RETRAIT);
    return Map.of(
        "count", count,
        "sumDepots", sumDepots,
        "sumRetraits", sumRetraits
    );
}

```

---

**3. Tests avec GraphiQL**

GraphiQL permet d'exécuter des requêtes et mutations pour tester les fonctionnalités ajoutées.

**Ajout d'une transaction :**

La mutation suivante ajoute un dépôt à un compte identifié par compteId :

---

```

mutation {
    addTransaction(transaction: {
        compteId: 1,
        montant: 500.0,
        date: "2024-11-18",
        type: DEPOT
    }) {
        id
        montant
        type
        compte {
            id
        }
    }
}

```

---

**Affichage des transactions d'un compte :**

La requête suivante retourne toutes les transactions associées à un compte :

---

```
query {
    compteTransactions(id: 1) {
        id
        montant
        date
        type
    }
}
```

---

#### Calcul des statistiques globales :

La requête suivante retourne les statistiques globales sur les transactions :

---

```
query {
    transactionStats {
        count
        sumDepots
        sumRetraits
    }
}
```

---

#### Résumé des fonctionnalités ajoutées

- Ajout d'une transaction avec la mutation addTransaction.
- Récupération des transactions d'un compte avec la requête compteTransactions.
- Calcul des statistiques globales avec la requête transactionStats.
- Gestion des erreurs avec des messages clairs en cas de compte ou transaction introuvable.



Cette extension améliore significativement l'application en ajoutant une gestion complète des transactions. Les fonctionnalités fournissent une vue claire sur les opérations financières, facilitent le suivi des comptes, et offrent des outils d'analyse globaux sur l'ensemble des transactions.