

# Rapport sur les micro-services GraphQL

**ACHBAD Fatima**

8/11/2023

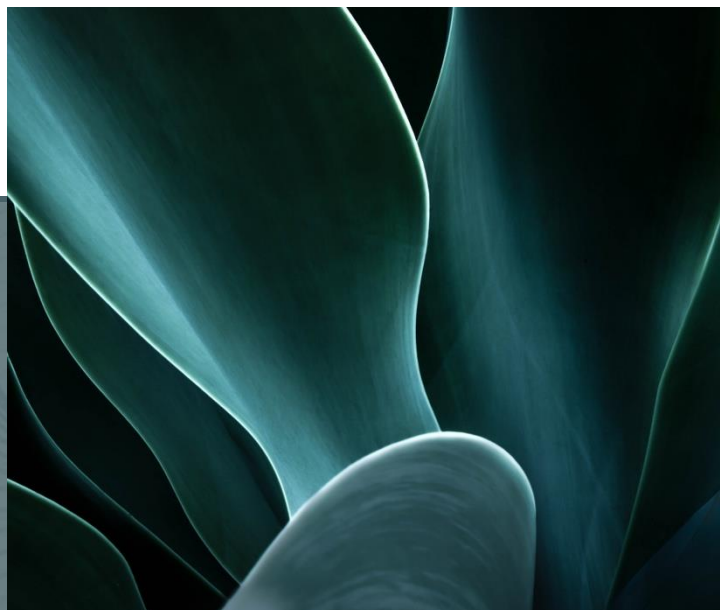
—

Mohamed YOUSSEFI

---

## Introduction

Les microservices sont une architecture logicielle qui consiste à découper une application en plusieurs services indépendants. Chaque service est responsable d'une fonction ou d'un ensemble de fonctions bien définies. Cette approche permet de rendre les applications plus flexibles, évolutives et faciles à maintenir. GraphQL est un langage de requête de données qui permet aux clients d'interroger des données d'un service GraphQL. GraphQL est un langage de requête déclaratif qui permet aux clients de spécifier les données qu'ils souhaitent obtenir. Cette approche permet de rendre les requêtes plus efficaces et faciles à écrire.





## Objectifs du rapport

---

Le principal objectif de l'utilisation des GraphQL est d'améliorer la communication entre le client et le serveur. Avec GraphQL, le client peut définir précisément les données dont il a besoin et obtenir uniquement celles-ci, évitant ainsi le problème d'overfetching ou d'underfetching rencontrés dans les API REST traditionnelles. Plus précisément, les objectifs de l'utilisation de GraphQL sont les suivants :

- Améliorer l'efficacité des requêtes : GraphQL permet au client de demander uniquement les données dont il a besoin, ce qui réduit la taille des requêtes et la charge du serveur.
  - Augmenter la flexibilité des requêtes : GraphQL permet au client de définir la structure des données qu'il souhaite obtenir, ce qui lui donne plus de liberté pour personnaliser ses requêtes.
  - Simplifier le développement : GraphQL fournit un modèle de données unifié, ce qui facilite le développement des clients et des serveurs.
- 

## Plan du rapport

Le rapport sera organisé comme suit :

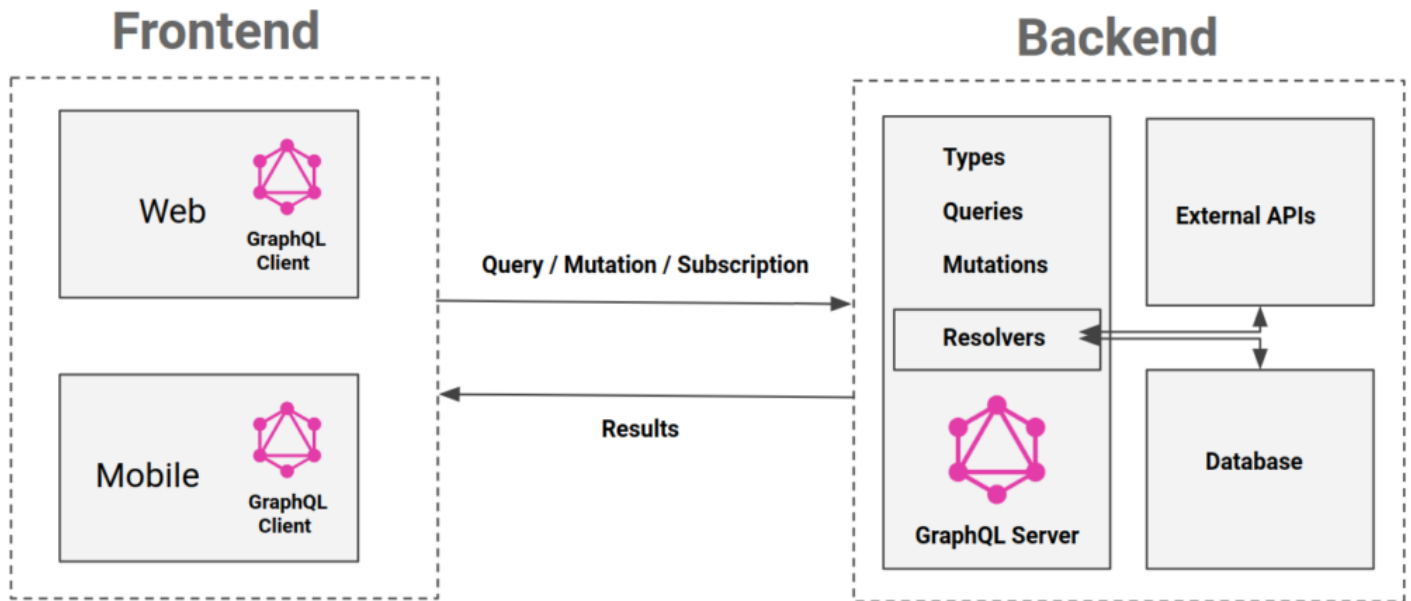
- + Introduction aux web services
- + Technologies : GraphQL , schémagraphqls
- + Cas pratique



## I. Introduction aux microservices REST

GraphQL est un langage de requête pour votre API, et un runtime côté serveur pour exécuter des requêtes en utilisant un système de type que vous définissez pour vos données. GraphQL n'est lié à aucune base de données ou moteur de stockage spécifique et est plutôt soutenu par votre code et vos données existants.

Un service GraphQL est créé en définissant des types et des champs sur ces types, puis en fournissant des fonctions pour chaque champ sur chaque type.



### Un micro service REST :

- **Flexibilité:** GraphQL permet aux clients de demander les données dont ils ont besoin, sans avoir à se soucier de la structure de l'API.
- **Efficacité:** GraphQL permet de réduire le nombre de requêtes HTTP nécessaires pour obtenir les données souhaitées.
- **Performance:** GraphQL peut améliorer les performances des applications, en réduisant le nombre de données qui doivent être transmises entre le client et le serveur.

## II. Les technologies

1. **GraphQL (Graph Query Language):** pour est un langage de requêtes et un environnement d'exécution, créé par Facebook en 2012, avant d'être publié comme projet open-source en 2015. Inscrit dans le modèle Client-Serveur, il propose une alternative aux API REST. La requête du client définit une structure de données, dont le stockage est éventuellement distribué, et le serveur suit cette structure pour retourner la réponse. Fortement typé, ce langage évite les problèmes de retour de données insuffisants (under-fetching) ou surnuméraires (over-fetching).
2. **Schéma GraphQL :** Le schéma GraphQL est la base de toute implémentation de serveur GraphQL. Chaque API GraphQL est définie par un **seul** schéma qui contient des types et des champs décrivant la manière dont les données des demandes seront renseignées. Les données qui transitent par votre API et les opérations effectuées doivent être validées par rapport au schéma.

Les schémas GraphQL sont écrits dans *Langage de définition du schéma*(SDL). SDL est composé de types et de champs dotés d'une structure établie :

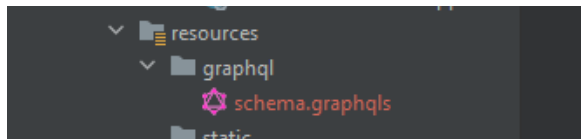
- **Les types:** Les types sont la façon dont GraphQL définit la forme et le comportement des données. GraphQL prend en charge une multitude de types qui seront expliqués plus loin dans cette section. Chaque type défini dans votre schéma contiendra sa propre portée. Le champ d'application comportera un ou plusieurs champs pouvant contenir une valeur ou une logique qui sera utilisée dans votre service GraphQL. Les types remplissent de nombreux rôles différents, les plus courants étant les objets ou les scalaires (types de valeurs primitives).
- **Champs:** Les champs existent dans le cadre d'un type et contiennent la valeur demandée au service GraphQL. Elles sont très similaires aux variables d'autres langages de programmation. La forme des données que vous définissez dans vos champs déterminera la manière dont les données sont structurées lors d'une opération de demande/réponse. Cela permet aux développeurs de prévoir ce qui sera renvoyé sans savoir comment le backend du service est implémenté.

# Cas pratique

Pour commencer ,on continue depuis le projet spring du BankAccount .Pour intégrer le graphql ,il faut ajouter la dépendance suivante :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-graphql</artifactId>
</dependency>
```

Et comme déjà signaler auparavant ,il nous faut un fichier pour configurer nos schémas GraphQL .



**Query :** Une *query* permet de **requêter le serveur GraphQL afin de récupérer un ensemble de données** : c'est donc une opération de lecture des données. Le *payload* de retour est en JSON et plusieurs *queries* peuvent s'exécuter en parallèle. Ces *queries* sont définies dans le schéma et traitées dans le *resolver*.

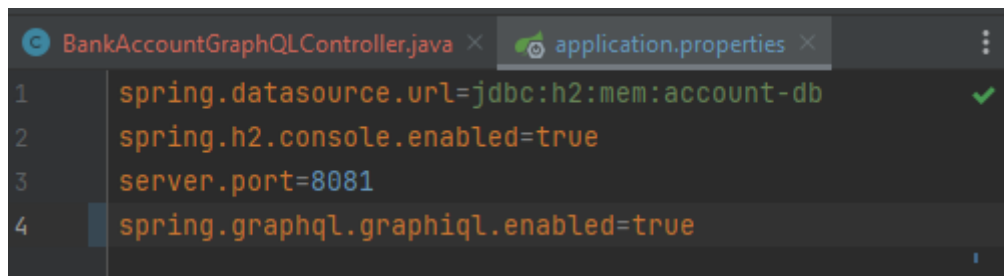
```
type Query {
  accountsList : [BankAccount]
}

type BankAccount {
  id : String!
  createdAt : Float!
  balance : Float!
  currency : String!
  type : String!
}
```

On crée donc une méthode qui va nous permettre de recevoir l'ensemble des comptes .

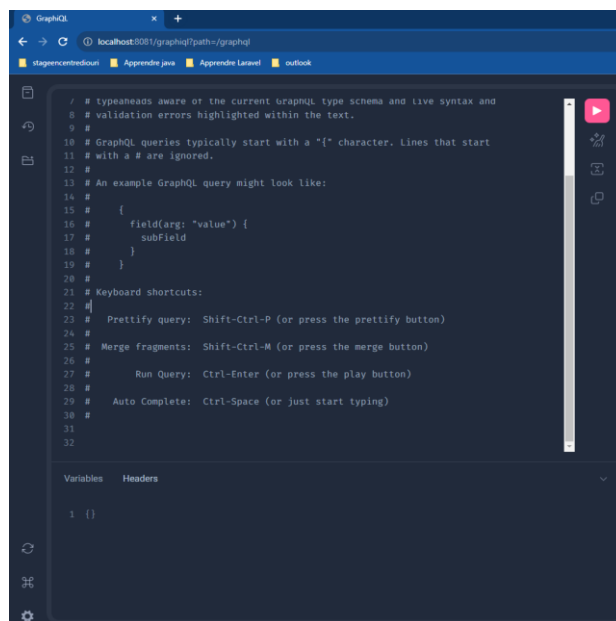
```
BankAccountGraphQLController.java
1 package com.example.bankaccountservice.web;
2
3 import com.example.bankaccountservice.entities.BankAccount;
4 import com.example.bankaccountservice.repositories.BankAccountRepository;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.graphql.data.method.annotation.QueryMapping;
7 import org.springframework.stereotype.Controller;
8
9 import java.util.List;
10
11 @Controller
12 public class BankAccountGraphQLController {
13     @Autowired
14     private BankAccountRepository bankAccountRepository;
15     @QueryMapping
16     public List<BankAccount> accountsList() { return bankAccountRepository.findAll(); }
17 }
```

Pour tester ,on active graphql depuis le fichier [application.properties](#)

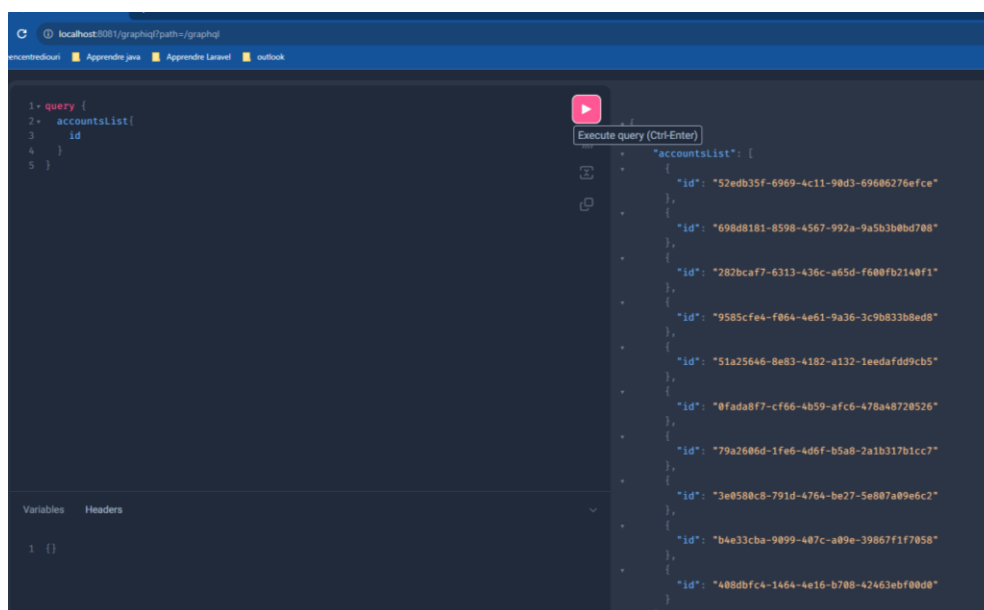


```
1 spring.datasource.url=jdbc:h2:mem:account-db ✓
2 spring.h2.console.enabled=true
3 server.port=8081
4 spring.graphql.graphiql.enabled=true
```

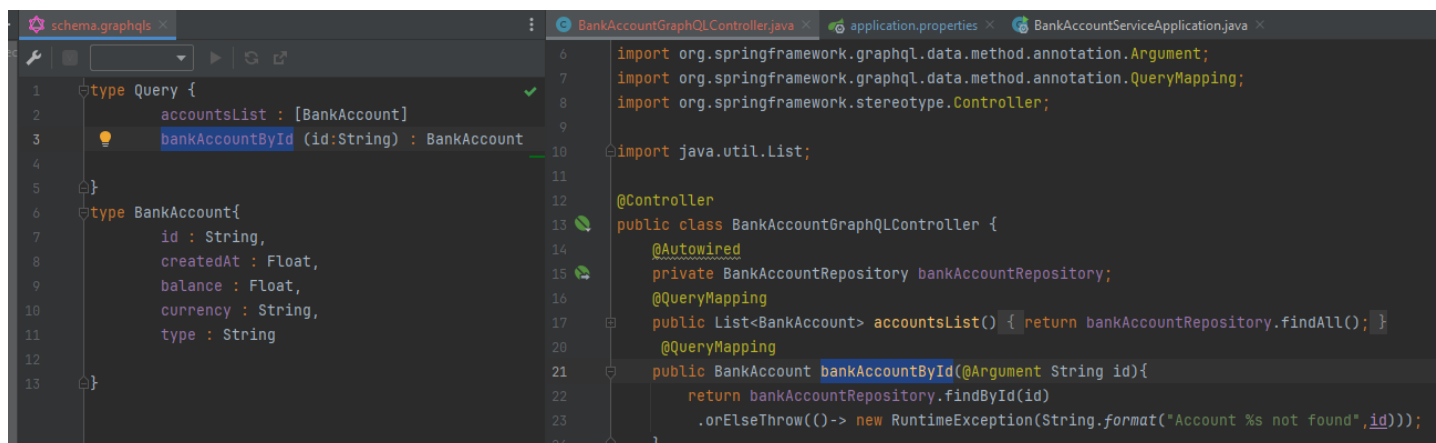
Comme première interface ,on a la suivante :



Pour tester notre première requête :



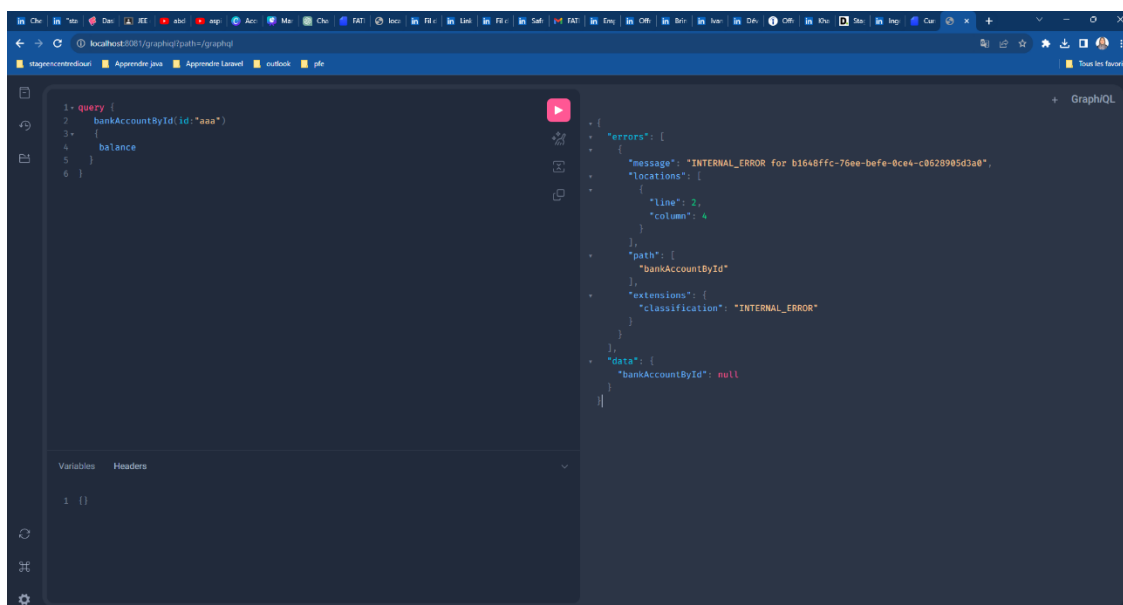
On crée un autre query par le meme principe , cette nouvelle méthode nous permet de récupérer un BankAccount par son id :



Pour le test :



Si j'incère un id non valide on aura le résultat comme suivant :

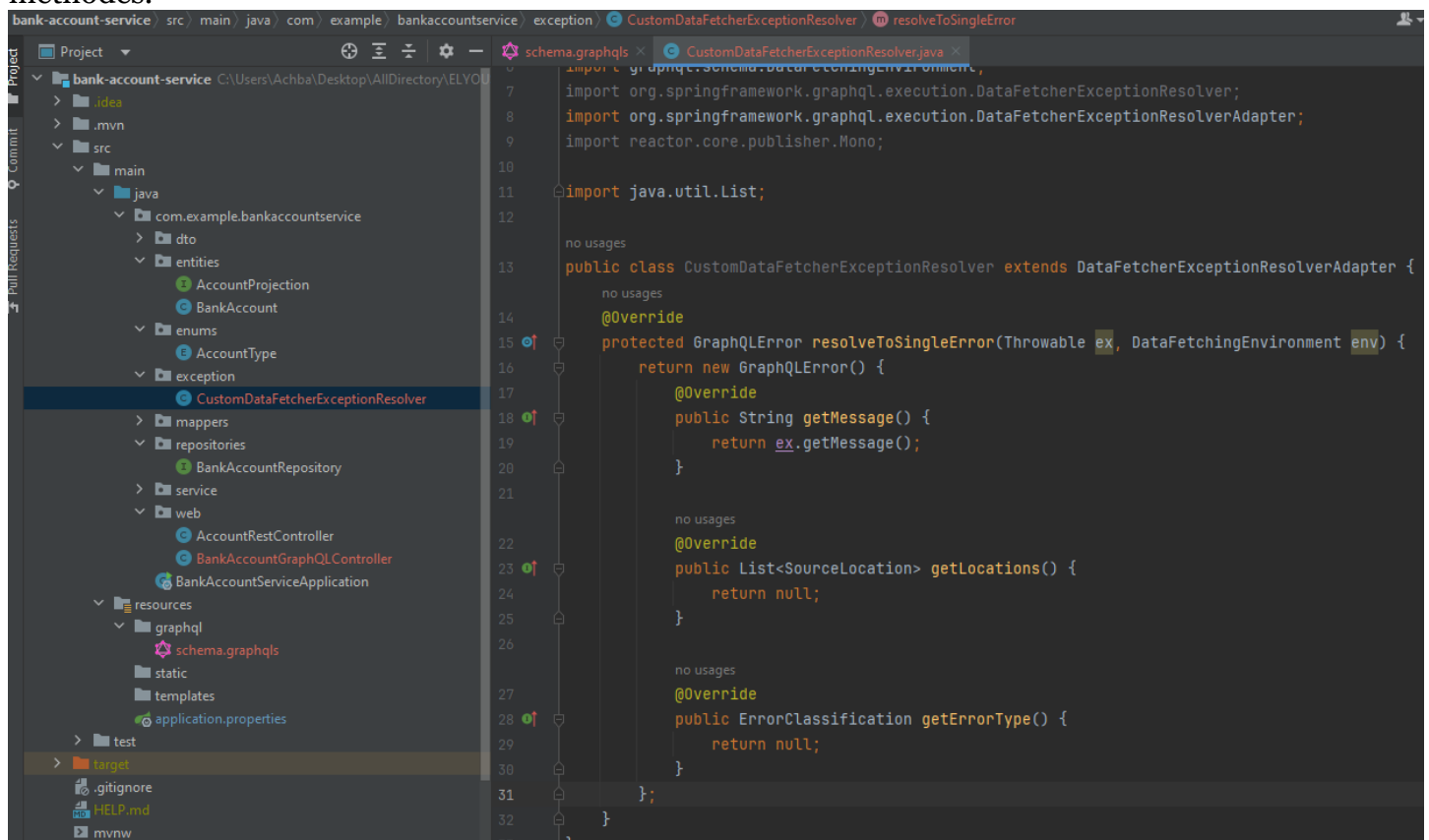




Pour gérer les exceptions, on crée un package avec nom exception .  
Avec Spring for GraphQL vous pouvez implémenter un `DataFetcherExceptionHandler` ou plus spécifiquement un `DataFetcherExceptionHandlerAdapter` que vous pouvez par exemple annoter avec `@Component` pour l'enregistrer automatiquement.

Le `DataFetcherExceptionHandler` de `graphql-java` est utilisé par Spring pour GraphQL en interne pour déléguer à vos classes `DataFetcherExceptionHandler`.

Dans votre propre `DataFetcherExceptionHandlerAdapter`, vous pouvez obtenir les informations qui sont disponibles comme `DataFetcherExceptionHandlerParameters` (chemin, `SourceLocation` et ainsi de suite) dans un `DataFetcherExceptionHandler` à partir du `DataFetchingEnvironment` qui est passé à `DataFetcherExceptionHandlerAdapter` **`resolveToSingleError`** et `resolveToMultipleErrors` méthodes.



```
import org.springframework.graphql.datafetcher.DataFetchingEnvironment;
import org.springframework.graphql.execution.DataFetcherExceptionHandler;
import org.springframework.graphql.execution.DataFetcherExceptionHandlerAdapter;
import reactor.core.publisher.Mono;

import java.util.List;

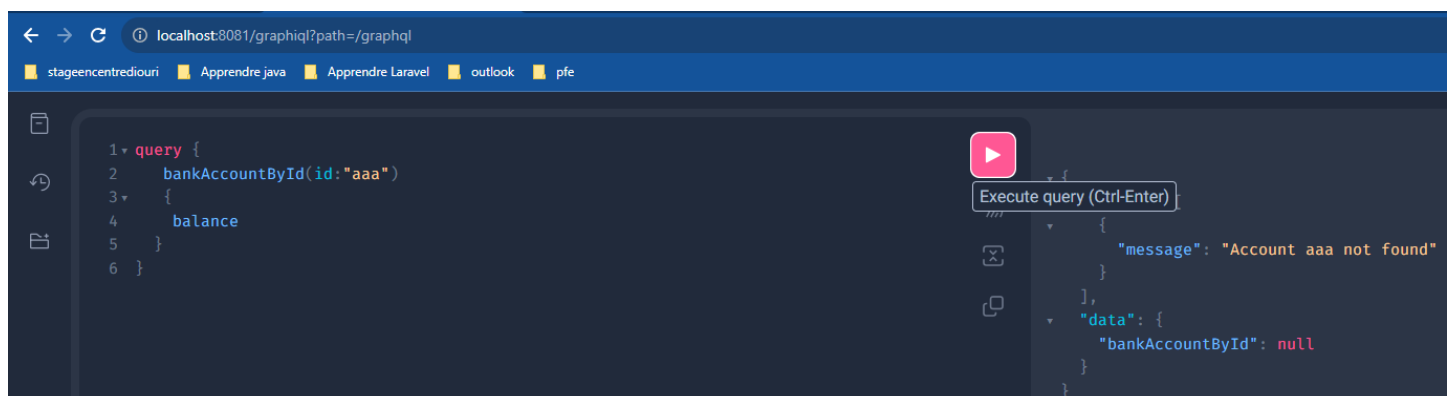
public class CustomDataFetcherExceptionHandlerAdapter extends DataFetcherExceptionHandlerAdapter {

    @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;
            }
        };
    }
}
```

On l'exploitant, on aura le résultat suivant :message plus significatif qu'avant :



```
query {
  bankAccountById(id:"aaa") {
    balance
  }
}
```

```
{
  "message": "Account aaa not found"
},
{
  "data": {
    "bankAccountById": null
  }
}
```

**Mutation :** Les *mutations* concernent tous les changements apportés aux données : ajout, modification, suppression. Leur fonctionnement est similaire aux *queries*, avec la définition de la *mutation* dans le schéma (dans le type *Mutation*) et la fonction associée dans le *resolver* (dans l'objet *Mutation*). Elle peut également retourner un objet, ce qui peut être utile pour récupérer l'état de l'objet mis à jour par cette *mutation*. Cependant, contrairement aux *queries*, les *mutations* s'exécutent en série, l'une après l'autre.

```
@MutationMapping
public BankAccountResponseDTO addAccount(@Argument BankAccountRequestDTO bankAccount ){
    return accountService.addAccount(bankAccount);
}

@MutationMapping
public BankAccountResponseDTO updateAccount(@Argument String id,@Argument BankAccountRequestDTO bankAccount){
    return accountService.updateAccount(id,bankAccount);
}

@MutationMapping
public boolean deleteAccount(@Argument String id){
    bankAccountRepository.deleteById(id);
    return true;
}
```

Le fichier de configuration est comme suivant :

```
type Mutation{
    addAccount(bankAccount : BankAccountDTO) :BankAccount,
    updateAccount(id : String, bankAccount : BankAccountDTO) : BankAccount,
    deleteAccount(id : String) : Boolean
}
```

Teste de méthode :

Comme dans l'image on peut donc utiliser les variables pour les passer aux paramètres de la fonction au lieu de travailler d'une manière spécifique .

```
1+ mutation($id:String,$t:String,$b:Float,$c:String){
2    updateAccount(
3      id : $id,
4+   bankAccount : {
5     type : $t,
6     balance : $b,
7     currency : $c
8   }
9+ ){
10   id,type,balance
11 }
12 }
```

Variables Headers

```
1+ { "t": "SAVING_ACCOUNT", "b": 50, "c": "MAD",
2   "id": "5b7c18b1-a281-4c16-a801-5bb932f490c6" }
```

Variables Headers

```
{
  "data": {
    "updateAccount": {
      "id": "5b7c18b1-a281-4c16-a801-5bb932f490c6",
      "type": "SAVING_ACCOUNT",
      "balance": 50
    }
  }
}
```

Pour tester les relations avec ce micro service , on crée une autre entité Customer .

```
package com.example.bankaccountservice.entities;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.List;

@Entity
@NoArgsConstructor @AllArgsConstructor @Data @Builder
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @OneToMany
    private List<BankAccount> bankAccounts;
}
```

Pour récupérer la liste des Customer ,on l'ajoute au niveau de Query ,ainsi que son type .

```
type Query {
  accountsList : [BankAccount]
  bankAccountById (id:String) : BankAccount
  customers : [Customer]
}

type Mutation {
  addAccount(bankAccount : BankAccountDTO) :BankAccount,
  updateAccount(id : String, bankAccount : BankAccountDTO) : BankAccount,
  deleteAccount(id : String) : Boolean
}

type BankAccount {
  id : String,
  createdAt : Float,
  balance : Float,
  currency : String,
  type : String
  customer : Customer
}

type Customer {
  id : Float,
  name : String,
  bankAccounts : [BankAccount]
}

input BankAccountDTO {
  balance : Float,
  currency : String,
  type : String
}
```

La méthode est comme suivante :

```
new *  
@QueryMapping  
public List<Customer> customers(){  
    return customerRepository.findAll();  
}
```

Pour tester :

The screenshot shows the GraphQL Playground interface. On the left, a query is defined:

```
1 query {  
2   customers {  
3     id, name, bankAccounts { balance, id, type }  
4   }  
5 }  
6 }
```

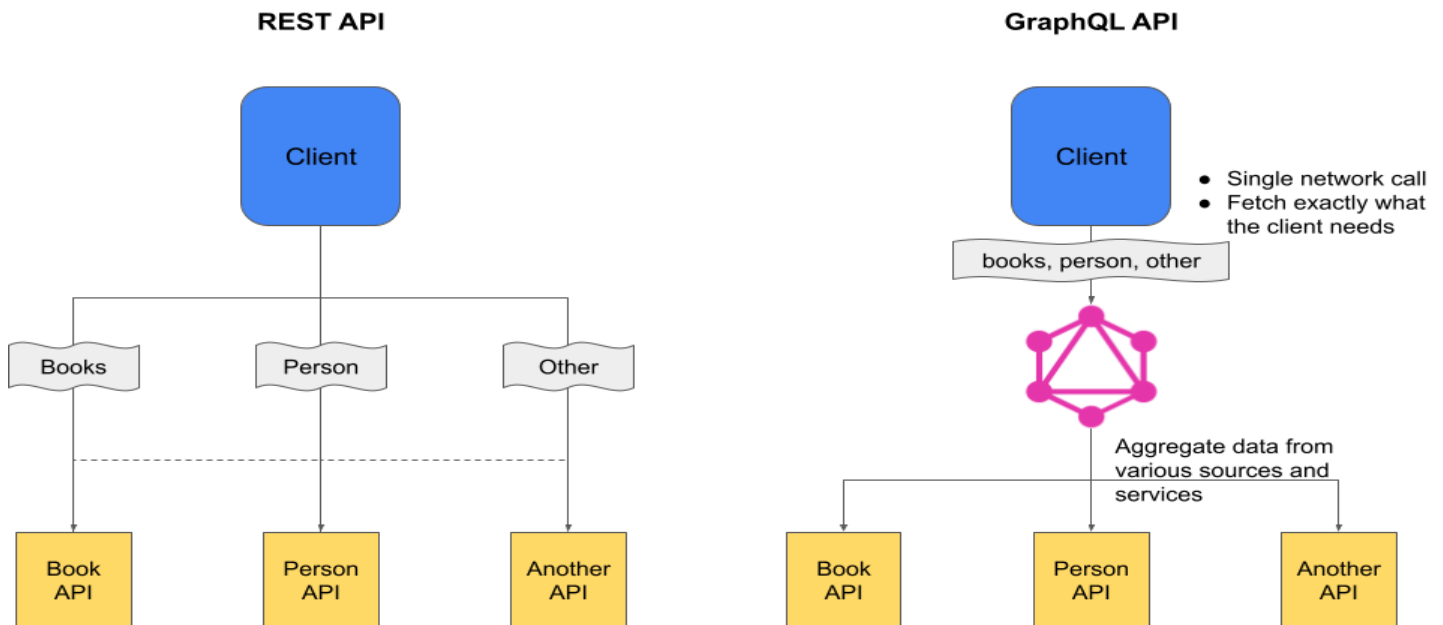
Below the query editor, the 'Variables' and 'Headers' tabs are visible, with 'Variables' showing an empty object: `{}`.

On the right, the 'Execute query (Ctrl-Enter)' button is highlighted. Below it, the JSON response is displayed:

```
{  
  "data": {  
    "customers": [  
      {  
        "id": 1,  
        "name": "Mohamed",  
        "bankAccounts": [  
          {  
            "balance": 59609.92918774076,  
            "id": "455fbf9b-25aa-402e-bc65-dce8fea21798",  
            "type": "CURRENT_ACCOUNT"  
          },  
          {  
            "balance": 57591.081854051095,  
            "id": "920f94a3-f3a3-4397-b763-c1786985ee7f",  
            "type": "CURRENT_ACCOUNT"  
          },  
          {  
            "balance": 87135.8683929164,  
            "id": "46f584e9-e062-4357-af24-e7d08c9cb081",  
            "type": "CURRENT_ACCOUNT"  
          },  
          {  
            "balance": 47363.71527711334,  
            "id": "5baa7b69-892a-4773-99e1-5aceaf7ed7f0",  
            "type": "CURRENT_ACCOUNT"  
          },  
          {  
            "balance": 95275.31460135372,  
            "id": "535e338e-abf2-4d77-a169-d0ff736bcd1b",  
            "type": "SAVING_ACCOUNT"  
          },  
          {  
            "balance": 56939.29638219523,  
            "id": "a3bc01ad-8d13-4308-8520-1aa6205a1a25",  
            "type": "CURRENT_ACCOUNT"  
          }  
        ]  
      }  
    ]  
  }  
}
```

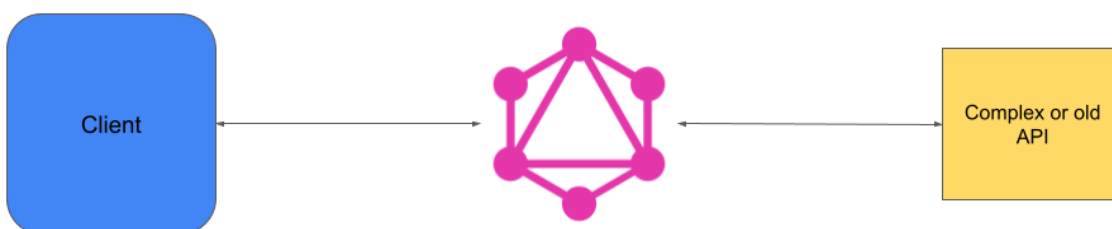
# Comparaison avec REST

En utilisant le formalisme de REST, le client doit se plier aux endpoints existants côté serveur pour récupérer les ressources dont il a besoin, et donc souvent appeler plusieurs endpoints pour récupérer la totalité des données (voir plus de données que nécessaire). Avec GraphQL, **un seul appel est suffisant** pour faire la même chose : le client effectuera cet unique appel au serveur GraphQL qui, lui, se chargera d'aller chercher toutes les ressources que la requête demande.



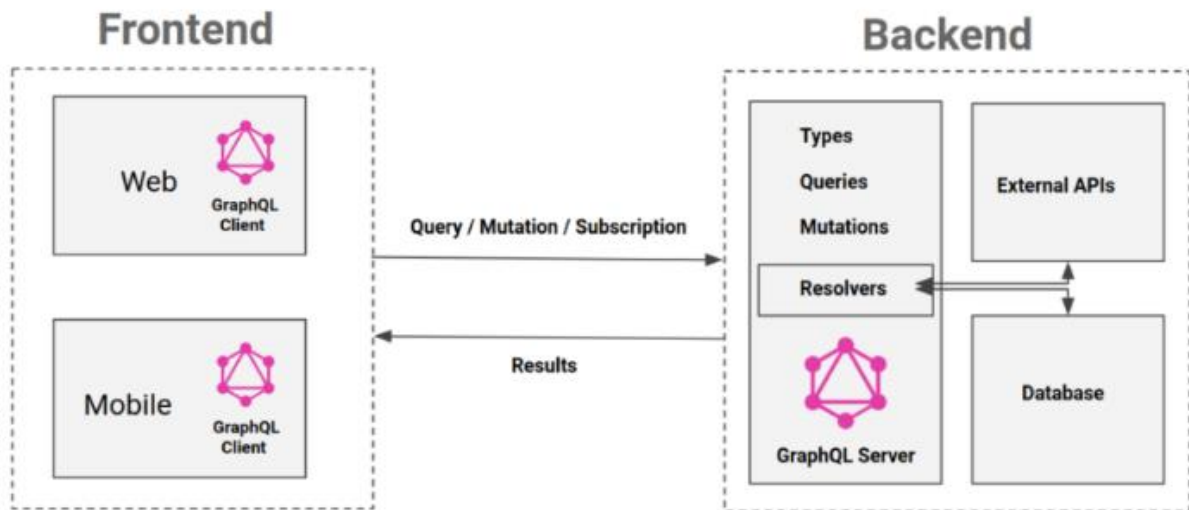
Grâce à GraphQL, **on va pouvoir récupérer plusieurs ressources en une seule requête** (pattern composite), ressources pouvant être stockées de différentes manières (API, BDD, fichiers, ...). Ce fonctionnement permet de demander exactement ce que l'on veut, sans récupérer trop de données (**over-fetching**) ou pas assez (**under-fetching**). Cela se traduit également par des appels plus légers et rapides entre le client et le serveur.

GraphQL peut également être utilisé dans d'autres circonstances. Par exemple, si l'on veut **simplifier l'utilisation d'une api complexe**, on peut ajouter une brique GraphQL entre le client et cette api (pattern facade). De la même manière, on peut inclure une couche graphql pour **enrichir une ancienne API avec une nouvelle fonctionnalité**, par exemple une couche d'authentification (pattern proxy)





## Schéma récapitulatif :



## Conclusion :

Les microservices sont une architecture logicielle qui consiste à découper une application en plusieurs services indépendants. Chaque service est responsable d'une partie bien définie du système et communique avec les autres services via des API.

GraphQL est un langage de requête open source qui permet d'interroger des données à partir de plusieurs sources de données. Il est conçu pour être flexible et évolutif, et permet aux clients de demander exactement les données dont ils ont besoin.

Les microservices GraphQL

L'association des microservices et de GraphQL offre un certain nombre d'avantages, notamment :

- **Flexibilité:** GraphQL permet aux clients de demander exactement les données dont ils ont besoin, ce qui permet aux services de fournir uniquement les données pertinentes.
- **Évolutivité:** GraphQL est conçu pour être évolutif, ce qui permet aux applications de s'adapter aux changements de besoins.
- **Simplicité:** GraphQL est un langage de requête simple à apprendre et à utiliser, ce qui facilite le développement d'applications.