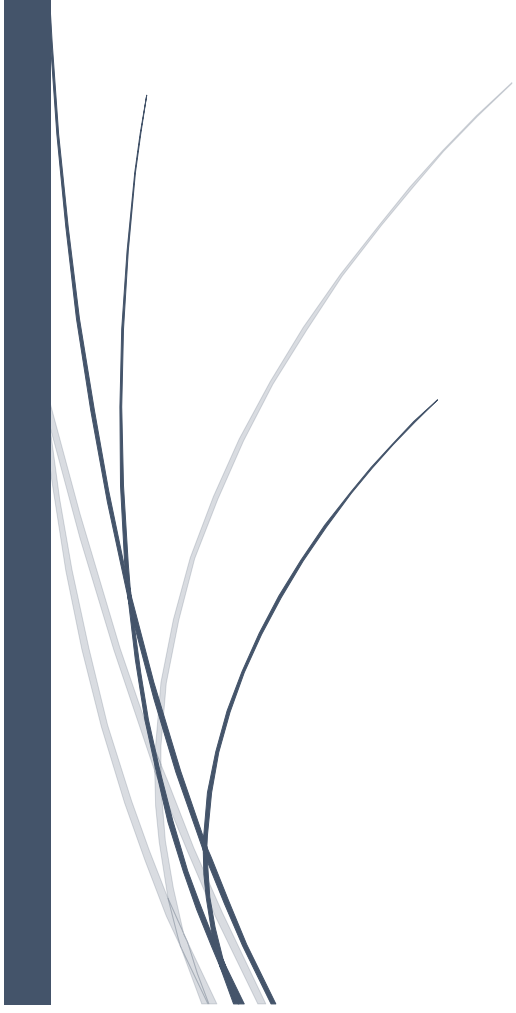
TP : Développer un service web avec GraphQL et Spring Boot



Architecture des composants d’entreprise

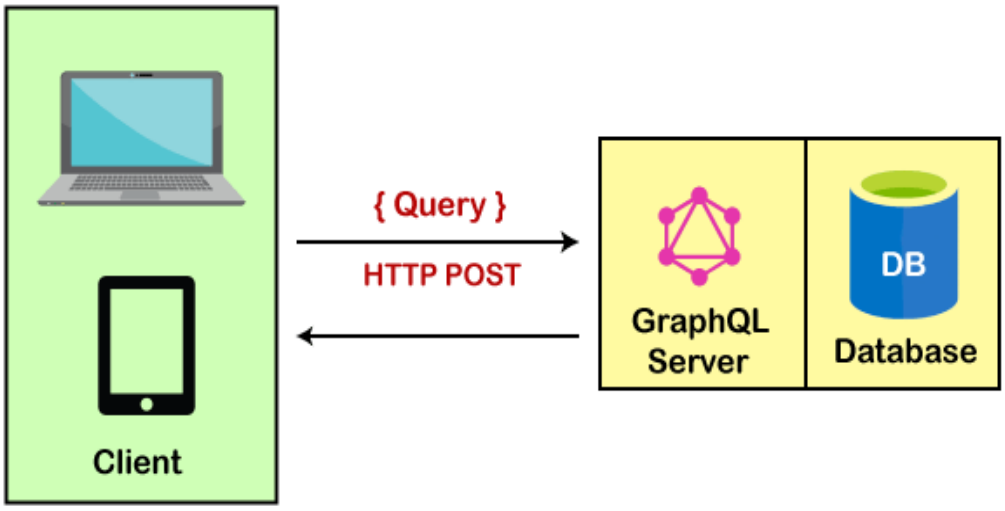
Table des matières

I. Objectif du TP

- Comprendre l'architecture du standard GraphQL.
- Maitriser la syntaxe du schéma GraphQL.
- Gérer les exceptions avec GraphQL.

II. Architecture de GraphQL

1. GraphQL : définition



- ✓ ***GraphQL*** (**Graph Query Language**) a été développé par Facebook, qui a commencé à l'utiliser pour les applications mobiles en 2012. La spécification de GraphQL est devenue Open Source en 2015. Elle est désormais supervisée par GraphQL Foundation (https://graphql.org/foundation/).

- ✓ GraphQL est un langage de requête et un environnement d'exécution côté serveur pour les APIs qui permet de fournir aux clients ***uniquement les données qu'ils ont demandées***.

- ✓ GraphQL est conçu pour mettre à la disposition des développeurs des API rapides, flexibles et faciles à utiliser.

- ✓ Utilisé à la place de REST, *GraphQL* permet aux développeurs de créer des requêtes qui extraient les données de plusieurs sources à l'aide d'un seul appel d'API.

- ✓ En outre, avec *GraphQL*, les équipes chargées de la maintenance des API peuvent librement ajouter ou retirer des champs sans perturber les requêtes existantes. De leur côté, les développeurs peuvent créer des API en utilisant les méthodes de leur choix. La spécification de GraphQL garantit que leur fonctionnement sera prévisible auprès des clients.

2. Schémas, résolveurs

```
type Query{
    customers:[CustomerDto]
}

type Mutation {
    createCustomer(dto:AddCustomerRequest):AddCustomerResponse
}

type CustomerDto {
    username : String,
    identityRef : String,
    firstname:String,
    lastname:String,
}


input AddCustomerRequest {
    username : String,
    identityRef : String,
    firstname:String,
    lastname:String
}

type AddCustomerResponse {
    message:String,
    username:String,
    identityRef:String,
    firstname:String,
    lastname:String
}
```

- customers est le nom du service pour consulter les clients.

- CustomerDto est le type des données qui seront retournés au clients.

- Pour les requêtes Select, le mot clé utilisé est Query.

- Toutes les requêtes autres que Select, le mot clé utilisé est Mutation.

- createCustomer est le nom du service pour créer un nouveau client.

- La requête createCustomer prends en paramètre un objet de type AddCustomerRequest.

- L'objet résultat de la requête createCustomer est AddCustomerResponse.

- Pour déclarer les objets de type paramètre, on utilise le mot clé "input".

- Pour déclarer les ojets de type response, on utilise le mot clé "type".

```
@Controller
public class CustomerGraphqlController {
    4 usages
    private final ICustomerService customerService;

    public CustomerGraphqlController(ICustomerService customerService) {...}

    @QueryMapping
    List<CustomerDto> customers() {
        return customerService.getAllCustomers();
    }

    @MutationMapping
    public AddCustomerResponse createCustomer(@Argument("dto") AddCustomerRequest dto) {
        return customerService.createCustomer(dto);
    }
```

- customers() est le nom du resolver qui sera exécuté par le serveur GraphQL lorsque la requête customers est envoyée.

- Pour les requêtes GraphQL de type Query, on annote les resolver avec l'annotation @QueryMapping.

- createCustomer() est le nom du resolver qui sera exécuté par le serveur GraphQL lorsque la requête createCustomer est envoyée.

-Le paramètre dto doit être annoté par @Argument afin de mapper le nom du paramètre en Input de la requête createCustomer avec le paramètre passé à la méthode createCustomer.

- Toutes les méthodes GraphQL (type Query ou bien Mutation) sont de type POST.

- ✓ Les développeurs d'API utilisent *GraphQL* pour créer ***un schéma*** qui liste toutes les données que les clients peuvent demander par l'intermédiaire de ce service.
- ✓ Un schéma *GraphQL* est constitué de types d'objets, qui définissent le genre d'objet qu'il est possible de demander et les champs qu'il contient.
- ✓ Lorsque les requêtes arrivent, GraphQL les compare au schéma, puis exécute celles qui ont été validées.
- ✓ Le développeur de l'API associe chaque champ d'un schéma à une fonction nommée résolveur. Le résolveur est appelé pour produire une valeur au cours de l'exécution.
- ✓ Toutes les requêtes dans GraphQL sont de type POST.

III. Prérequis

- IntelliJ IDEA ;
- JDK version 17 ;
- Une connexion Internet pour permettre à Maven de télécharger les librairies.

NB : *Ce TP a été réalisé avec IntelliJ IDEA 2023.2.3 (Ultimate Edition).*

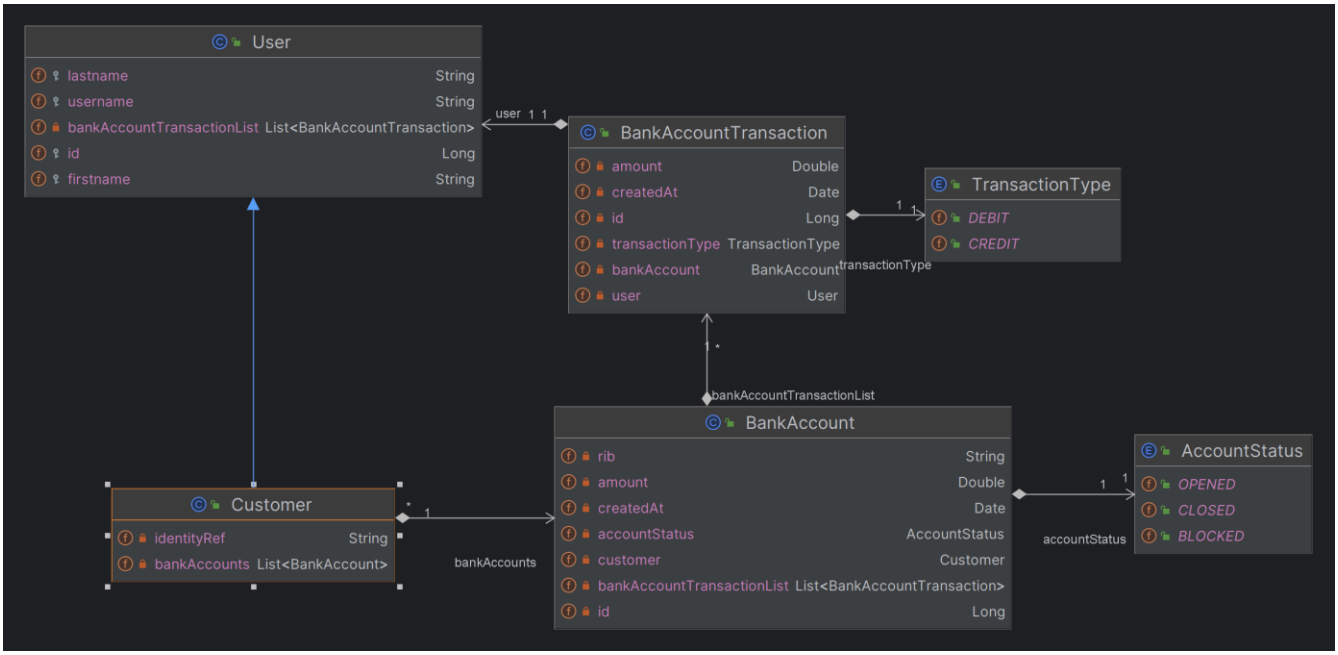
IV. Développement de l'application

a. Le modèle de données

Nous allons développer un service web avec *GraphQL* nommé *bank-service* pour une banque donnée qui offre les services suivants :

- Consulter la liste des clients de la banque.
- Consulter un client par son numéro d'identité.
- Modifier un client par son numéro d'identité.
- Supprimer un client par son numéro d'identité.
- Consulter la liste des comptes bancaires.
- Consulter un compte bancaire par son RIB.
- Effecteur des virements d'un compte vers un autre compte.

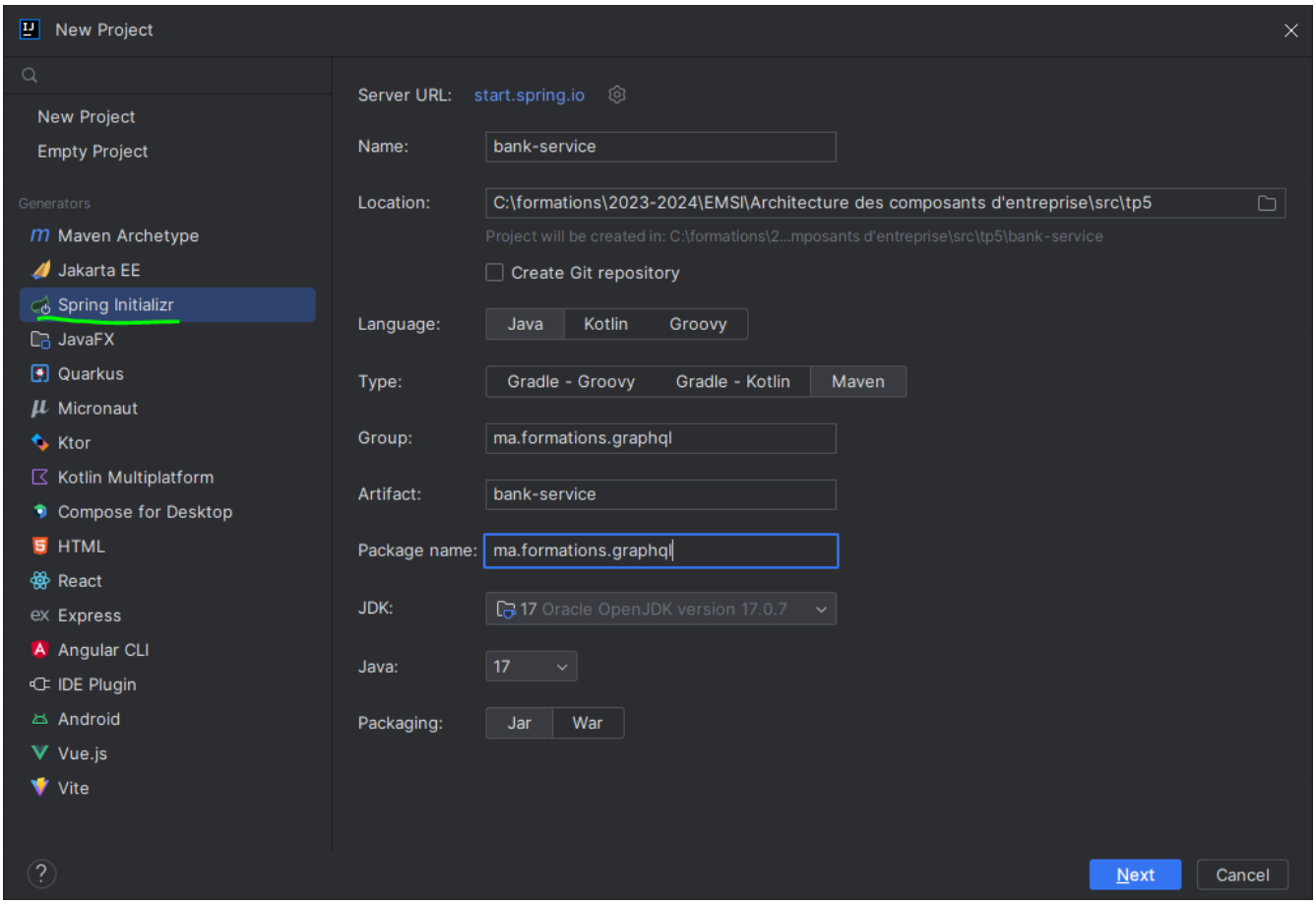
Le modèle de données que nous allons implémenter est le suivant :



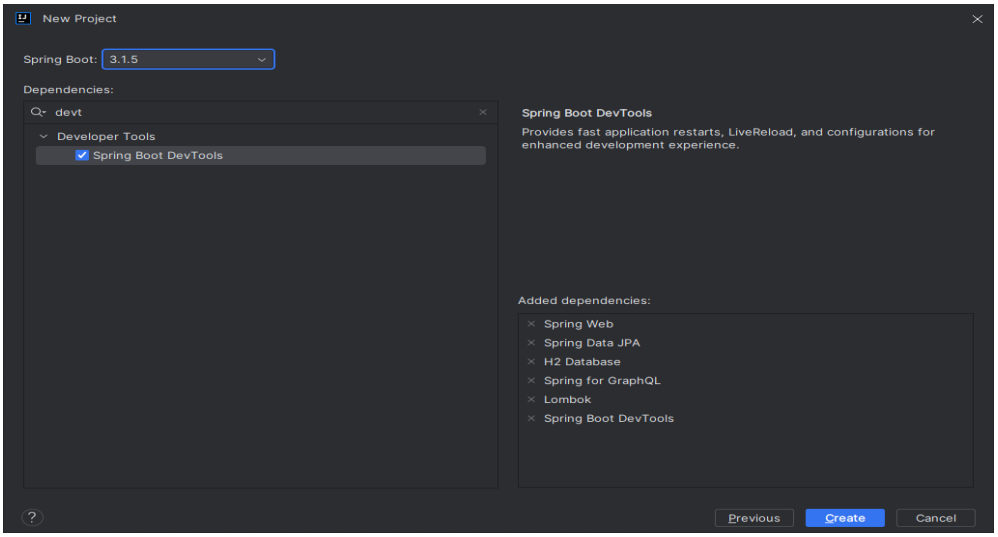
- ✓ Un client peut avoir un ou plieurs comptes bancaires.
- ✓ Sur un compte bancaire, le client peut effectuer une ou plusieurs transactions.
- ✓ Un client est un utilisateur.
- ✓ Un utilisateur (exemple : agent guichet) peut effectuer une ou plusieurs transactions.
- ✓ Un compte bancaire peut avoir les statuts suivants : OPENED, CLOSED ou bien BLOCKED.
- ✓ Une transaction est de deux types : DEBIT ou CREDIT.
- ✓ Aucune transaction ne peut être effectuée sur un compte « CLOSED » ou bien « BLOCKED ».
- ✓ Pour effectuer un virement, le solde du compte bancaire de l'émetteur doit être supérieur au montant du virement.
- ✓ L'identité du client est unique.
- ✓ Le nom d'utilisateur (*username*) est unique.

b. Création du projet Maven

- Créer un nouveau projet comme le montre la fenêtre suivante :



1. Entrer le nom de votre projet.
2. Dans Language, cliquer sur Java.
3. Dans Type, cliquer sur Maven.
4. Entrer le Group et l'Artifact.
5. Entrer le nom du package.
6. Choisir JDK 17.
7. Dans Packaging, cliquer sur Jar. Enfin, cliquer sur Next. La fenêtre suivante s'affiche :



- Ajouter les dépendances suivantes : Spring Web, Spring Data JPA, H2 Database, Spring for GraphQL, Lombok et Spring Boot DevTools et cliquer ensuite sur Create.

c. Le fichier pom.xml

- Ajouter au niveau du fichier pom.xml la dépendance de la librairie ***ModelMapper*** suivante :

```
<dependency>
  <groupId>org.modelmapper</groupId>
  <artifactId>modelmapper</artifactId>
  <version>3.2.0</version>
</dependency>
```

Explications :

- ✓ ModelMapper permet de convertir les objets DTO (Data Transfer Object) vers les objets BO (Business Object) et vice versa. L'objectif est de ne pas utiliser les BO dans la couche présentation.

d. Développement des classes Model

- Créer le package ***ma.formations.graphql.enums*** et ensuite créer les enums suivants :
  - ✓ L'enum ***AccountStatus*** :

```
package ma.formations.graphql.enums;

public enum AccountStatus {
  OPENED, CLOSED, BLOCKED
}
```

  - ✓ L'enum ***TransactionType*** :

```
package ma.formations.graphql.enums;

public enum TransactionType {
  CREDIT,DEBIT
}
```

- Créer le package ***ma.formations.graphql.service.model*** et ensuite créer les classes suivantes :
  - ✓ La classe ***User*** :

```
package ma.formations.graphql.service.model;

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

```java
import java.util.List;

@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
@Inheritance(strategy = InheritanceType.JOINED)
public class User {
   @Id
   @GeneratedValue
   protected Long id;
   protected String username;
   protected String firstname;
   protected String lastname;

   @OneToMany(mappedBy = "user")
   private List<BankAccountTransaction> bankAccountTransactionList;

   public User(String username) {
      this.username = username;

   }
}
```

- ✓ La classe ***Customer*** :

```java
package ma.formations.graphql.service.model;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.OneToMany;
import jakarta.persistence.PrimaryKeyJoinColumn;
import lombok.*;

import java.util.List;

@Entity
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Getter
@Setter
@PrimaryKeyJoinColumn(name = "id")
public class Customer extends User {
    @Column(unique = true)
    private String identityRef;
    @OneToMany(mappedBy = "customer")
    private List<BankAccount> bankAccounts;
}
```

- ✓ La classe ***BankAccount*** :

```java
package ma.formations.graphql.service.model;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import ma.formations.graphql.enums.AccountStatus;

import java.util.Date;
import java.util.List;

@Entity
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Data
public class BankAccount {
    @Id
    @GeneratedValue
    private Long id;
    private String rib;
    private Double amount;
    private Date createdAt;
    @Enumerated(EnumType.STRING)
    private AccountStatus accountStatus;

    @ManyToOne
    private Customer customer;
    @OneToMany(mappedBy = "bankAccount")
    private List<BankAccountTransaction> bankAccountTransactionList;


}
```

- ✓ La classe ***BankAccountTransaction*** :

```java
package ma.formations.graphql.service.model;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import ma.formations.graphql.enums.TransactionType;

import java.util.Date;

@Entity
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Data
public class BankAccountTransaction {
    @Id
    @GeneratedValue
    private Long id;

    @Temporal(TemporalType.TIMESTAMP)
    private Date createdAt;
```

```
    @Enumerated(EnumType.STRING)
    private TransactionType transactionType;
    private Double amount;
    @ManyToOne
    private BankAccount bankAccount;

    @ManyToOne
    private User user;
}
```

✓ La classe ***GetTransactionListBo*** :

```
package ma.formations.graphql.service.model;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.Date;

@NoArgsConstructor
@AllArgsConstructor
@Data
@Builder
public class GetTransactionListBo {
    private String rib;
    private Date dateTo;
    private Date dateFrom;
}
```

e. Développement des classes utilitaires

- Créer le package *ma.formations.graphql.common*.
- Dans ce dernier, créer la classe ***CommonTools*** suivante :

```
package ma.formations.graphql.common;

import lombok.Data;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

@Component
@Data
public class CommonTools {

    @Value("${graphql.date.format}")
    private String dateFormat;

    public String dateToString(Date date) {
        SimpleDateFormat formatter = new SimpleDateFormat(dateFormat);
        return formatter.format(date);
    }

    public Date stringToDate(String date) throws ParseException {
```

```
        SimpleDateFormat formatter = new SimpleDateFormat(dateFormat);
        return formatter.parse(date);
    }
}
```

Explications :

- ✓ L'annotation ***@Value*** permet d'injecter la valeur de la clé *graphql.date.format* qui existe au niveau du fichier application.properties.
- ✓ La syntaxe ***${key}*** est de Spring EL (Spring Expression Language).

f. Configuration de ModelMapper

- Créer le package *ma.formations.graphql.config.*
- Dans ce dernier, créer la classe **ModelMapperConfig** suivante :

```
package ma.formations.graphql.config;

import lombok.AllArgsConstructor;
import ma.formations.graphql.common.CommonTools;
import ma.formations.graphql.service.exception.BusinessException;
import org.modelmapper.AbstractConverter;
import org.modelmapper.Converter;
import org.modelmapper.ModelMapper;
import org.modelmapper.convention.MatchingStrategies;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.text.ParseException;
import java.util.Date;

@Configuration
@AllArgsConstructor
public class ModelMapperConfig {
  private CommonTools tools;

  @Bean
  public ModelMapper modelMapper() {
    ModelMapper modelMapper = new ModelMapper();
    modelMapper.getConfiguration().
        setMatchingStrategy(MatchingStrategies.LOOSE).
        setFieldMatchingEnabled(true).
        setSkipNullEnabled(true).
        setFieldAccessLevel(org.modelmapper.config.Configuration.AccessLevel.PRIVATE);
    Converter<Date, String> dateToStringConverter = new AbstractConverter<>() {
      @Override
      public String convert(Date date) {
        return tools.dateToString(date);
      }
    };
    Converter<String, Date> stringToDateConverter = new AbstractConverter<>() {
      @Override
      public Date convert(String s) {
        try {
```

```
                return tools.stringToDate(s);
            } catch (ParseException e) {
                throw new BusinessException(String.format("the date %s doesn't respect the format %s ", s,
tools.getDateFormat()));
            }
        }
    };
    modelMapper.addConverter(dateToStringConverter);
    modelMapper.addConverter(stringToDateConverter);

    return modelMapper;
  }
}
```

Explications :

- ✓ La stratégie ***LOOSE*** permet le mappage entre des champs portant des noms et des types différents, tandis que la stratégie ***STRICT*** applique des règles de mappage strictes et déclenche une exception lorsqu'un mappage ne peut pas être effectué.
- ✓ ***setSkipNullEnabled(true)*** permet au ModelMapper d'ignorer les champs nuls dans la conversion.
- ✓ ***setFieldMatchingEnabled(true)*** et ***setFieldAccessLevel(AccessLevel.PRIVATE)*** sont nécessaires lorsque vous souhaitez convertir une JavaBean vers des classes qui sont des Builders. Dans ce cas, le ModelMapper va passer les valeurs en utilisant les variables d'instances directement sans passer par les Setters. Remarquer qu'il n'y a pas les Setters dans les classes qui respectent le Design Pattern Builder.
- ✓ Remarquer que nous avons ajouté deux convertisseurs (Converter interface) qui seront utilisés par *ModelMapper* pour convertir les dates en format String et vice versa.

g. Développement des classes DTO (Data Transfer Object)

- Créer les packages suivantes :
  - ✓ ma.formations.graphql.dtos.customer
  - ✓ ma.formations.graphql.dtos.bankaccount
  - ✓ ma.formations.graphql.dtos.transaction
  - ✓ ma.formations.graphql.dtos.user
- Dans le package *ma.formations.graphql.dtos.customer*, créer les classes suivantes : ***CustomerDto***, ***AddCustomerRequest, AddCustomerResponse***, ***UpdateCustomerRequest*** et ***UpdateCustomerResponse***
  - ❖ La classe ***CustomerDTO*** :

```
package ma.formations.graphql.dtos.customer;

import lombok.AllArgsConstructor;
```

```
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.List;

@NoArgsConstructor
@AllArgsConstructor
@Builder
@Data
public class CustomerDto {
    private Long id;
    private String username;
    private String identityRef;
    private String firstname;
    private String lastname;
}
```

- La classe ***AddCustomerRequest*** :

```
package ma.formations.graphql.dtos.customer;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@NoArgsConstructor
@AllArgsConstructor
@Builder
@Data
public class AddCustomerRequest {
   private String username;
   private String identityRef;
   private String firstname;
   private String lastname;
}
```

- La classe ***AddCustomerResponse*** :

```
package ma.formations.graphql.dtos.customer;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@NoArgsConstructor
@AllArgsConstructor
@Builder
@Data
public class AddCustomerResponse {
   private String message;
   private Long id;
   private String username;
   private String identityRef;
   private String firstname;
```

```
    private String lastname;
}
```

- Dans le package *ma.formations.graphql.dtos.bankaccount*, créer les classes suivantes :

  BankAccountDto, AddBankAccountRequest et AddBankAccountResponse :

  - La classe ***BankAccountDto :***

```
package ma.formations.graphql.dtos.bankaccount;

import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import ma.formations.graphql.dtos.customer.CustomerDto;
import ma.formations.graphql.enums.AccountStatus;

@NoArgsConstructor
@AllArgsConstructor
@Builder
@Data
public class BankAccountDto {
    private Long id;
    private String rib;
    private Double amount;
    private String createdAt;
    private AccountStatus accountStatus;
    private CustomerDto customer;
}
```

  - La classe ***AddBankAccountRequest :***

```
package ma.formations.graphql.dtos.bankaccount;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@NoArgsConstructor
@AllArgsConstructor
@Builder
@Data
public class AddBankAccountRequest {
    private String rib;
    private Double amount;
    private String customerIdentityRef;
}
```

  - La classe ***AddBankAccountResponse :***

```
package ma.formations.graphql.dtos.bankaccount;

import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import lombok.AllArgsConstructor;
import lombok.Builder;
```

```
import lombok.Data;
import lombok.NoArgsConstructor;
import ma.formations.graphql.dtos.customer.CustomerDto;
import ma.formations.graphql.enums.AccountStatus;

@NoArgsConstructor
@AllArgsConstructor
@Builder
@Data
public class AddBankAccountResponse {
    private String message;
    private Long id;
    private String rib;
    private Double amount;
    private String createdAt;
    private AccountStatus accountStatus;
    private CustomerDto customer;
}
```

- Dans le package *ma.formations.graphql.dtos.transaction*, créer les classes suivantes :

  TransactionDto, AddWirerTransferRequest, AddWirerTransferResponse et GetTransactionListRequest :

  - La classe ***TransactionDto :***

```
package ma.formations.graphql.dtos.transaction;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import ma.formations.graphql.dtos.bankaccount.BankAccountDto;
import ma.formations.graphql.dtos.user.UserDto;

@NoArgsConstructor
@AllArgsConstructor
@Builder
@Data
public class TransactionDto {
    private Long id;
    private String createdAt;
    private String transactionType;
    private Double amount;
    private BankAccountDto bankAccount;
    private UserDto user;
}
```

  - La classe ***AddWirerTransferRequest :***

```
package ma.formations.graphql.dtos.transaction;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
```

```
public class AddWirerTransferRequest {
    private String ribFrom;
    private String ribTo;
    private Double amount;
    private String username;
}
```

- La classe ***AddWirerTransferResponse :***

```
package ma.formations.graphql.dtos.transaction;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class AddWirerTransferResponse {
    private String message;
    private TransactionDto transactionFrom;
    private TransactionDto transactionTo;
}
```

- La classe ***GetTransactionListRequest :***

```
package ma.formations.graphql.dtos.transaction;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@NoArgsConstructor
@AllArgsConstructor
@Data
@Builder
public class GetTransactionListRequest {
    private String rib;
    private String dateTo;
    private String dateFrom;
}
```

- Dans le package ma.formations.graphql.dtos.user, créer la classe UserDto suivante :

```
package ma.formations.graphql.dtos.user;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@NoArgsConstructor
@AllArgsConstructor
@Data
@Builder
public class UserDto {
```

```
    protected String username;
    protected String firstname;
    protected String lastname;
}
```

h. Développement de la couche DAO (Data Access Object)

- Créer le package ***ma.formations.graphql.dao*** et créer ensuite les interfaces suivantes :

  ✓ L'interface ***UserRepository*** :

```
package ma.formations.graphql.dao;

import ma.formations.graphql.service.model.User;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);
}
```

  ✓ L'interface ***CustomerRepository*** :

```
package ma.formations.graphql.dao;

import ma.formations.graphql.service.model.Customer;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface CustomerRepository extends JpaRepository<Customer, Long> {
    Optional<Customer> findByIdentityRef(String identityRef);
    Optional<Customer> findByUsername(String username);
}
```

  ✓ L'interface ***BankAccountRepository*** :

```
package ma.formations.graphql.dao;

import ma.formations.graphql.service.model.BankAccount;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface BankAccountRepository extends JpaRepository<BankAccount, Long>
{
    Optional<BankAccount> findByRib(String rib);
}
```

  ✓ L'interface ***BankAccountTransactionRepository*** :

```
package ma.formations.graphql.dao;

import ma.formations.graphql.service.model.BankAccountTransaction;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Date;
import java.util.List;
```

```
public interface BankAccountTransactionRepository extends
JpaRepository<BankAccountTransaction, Long> {
    List<BankAccountTransaction> findByBankAccount_RibAndCreatedAtBetween(String
rib, Date from, Date to);
}
```

i. Paramétrage de la base de données H2

- Modifier le fichier ***application.properties*** comme suit :

```
# The name of the H2 database :
spring.datasource.url=jdbc:h2:mem:testdb
# The H2 Driver :
spring.datasource.driverClassName=org.h2.Driver
spring.data.jpa.repositories.bootstrap-mode=default
spring.datasource.username=sa
spring.datasource.password=
# The Dialect : java => SQL compatible with H2
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
# automatic creation and modification of tables
spring.jpa.hibernate.ddl-auto=update
# Activate the H2 console :
spring.h2.console.enabled=true
# For customizing the console URL
spring.h2.console.path=/h2
spring.jpa.properties.hibernate.globally_quoted_identifiers=true
graphql.date.format=yyyy-MM-dd HH:mm:ss
```

Explications :

- ✓ Dans ce fichier, nous configurons l'URL de la base de données, le Driver, le nom d'utilisateur, le mot de passe, le Dialect, la stratégie pour la génération du schéma, l'activation de la console H2 et le path pour y accéder. Pour rappel, Spring Boot gère par défaut ces paramètres au cas ou vous ne les renseignez pas. Dans ce cas de figure :
  - le nom de la base de données sera généré automatiquement.
  - Le path par défaut pour accéder à la console est /h2-console.
- ✓ La clé **spring.jpa.properties.hibernate.globally_quoted_identifiers** permet de résoudre le problème de l'utilisation des noms réservés (déjà hutilisés par Hibernate) dans les noms des identificateurs. Dans notre cas, le nom de la classe User est réservé. En ajoutant ce paramètre, Hibernate mettra le nom de la classe User entre deux quotes (" user ").

j. Développement de la couche Service (la couche Métier)

- Créer le package ***ma.formations.graphql.service.exception*** et créer ensuite la classe ***BusinessException*** suivante :

```
package ma.formations.graphql.service.exception;

public class BusinessException extends RuntimeException {
```

```
    public BusinessException(String message) {
        super(message);
    }
}
```

- Dans le package ***ma.formations.graphql.service***, créer les interfaces ICustomerService, IBankAccountService, ITransactionService et les classes CustomerServiceImpl, BankAccountServiceImpl, et TransactionServiceImpl suivantes :

  - ❖ L'interface ICustomerService :

```
package ma.formations.graphql.service;

import ma.formations.graphql.dtos.customer.*;

import java.util.List;

public interface ICustomerService {
  List<CustomerDto> getAllCustomers();
  AddCustomerResponse createCustomer(AddCustomerRequest addCustomerRequest);
  UpdateCustomerResponse updateCustomer(String identityRef, UpdateCustomerRequest updateCustomerRequest);
  CustomerDto getCustomByIdentity(String identity);
  String deleteCustomerByIdentityRef(String identityRef);
}
```

  - ❖ L'interface IBankAccountService :

```
package ma.formations.graphql.service;

import ma.formations.graphql.dtos.bankaccount.AddBankAccountRequest;
import ma.formations.graphql.dtos.bankaccount.AddBankAccountResponse;
import ma.formations.graphql.dtos.bankaccount.BankAccountDto;

import java.util.List;

public interface IBankAccountService {
    AddBankAccountResponse saveBankAccount(AddBankAccountRequest dto);
    List<BankAccountDto> getAllBankAccounts();
    BankAccountDto getBankAccountByRib(String rib);
}
```

  - ❖ L'interface ITransactionService :

```
package ma.formations.graphql.service;

import ma.formations.graphql.dtos.transaction.AddWirerTransferRequest;
import ma.formations.graphql.dtos.transaction.AddWirerTransferResponse;
import ma.formations.graphql.dtos.transaction.GetTransactionListRequest;
import ma.formations.graphql.dtos.transaction.TransactionDto;

import java.util.Date;
import java.util.List;
public interface ITransactionService {
    AddWirerTransferResponse wiredTransfer(AddWirerTransferRequest dto);
    List<TransactionDto> getTransactions(GetTransactionListRequest dto);
}
```

❖ La classe CustomerServiceImpl :

```
package ma.formations.graphql.service;

import lombok.AllArgsConstructor;
import ma.formations.graphql.dao.CustomerRepository;
import ma.formations.graphql.dtos.customer.*;
import ma.formations.graphql.service.exception.BusinessException;
import ma.formations.graphql.service.model.Customer;
import org.modelmapper.ModelMapper;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;
import java.util.stream.Collectors;

@Service
@Transactional
@AllArgsConstructor
public class CustomerServiceImpl implements ICustomerService {

   private final CustomerRepository customerRepository;
   private final ModelMapper modelMapper;

   @Override
   public List<CustomerDto> getAllCustomers() {
      return customerRepository.findAll().stream().
            map(customer -> modelMapper.map(customer, CustomerDto.class)).
            collect(Collectors.toList());
   }

   @Override
   public AddCustomerResponse createCustomer(AddCustomerRequest addCustomerRequest) {
      Customer bo = modelMapper.map(addCustomerRequest, Customer.class);
      String identityRef = bo.getIdentityRef();
      String username = bo.getUsername();

      customerRepository.findByIdentityRef(identityRef).ifPresent(a ->
            {
               throw new BusinessException(String.format("Customer with the same identity [%s] exist", identityRef));
            }
      );

      customerRepository.findByUsername(username).ifPresent(a ->
            {
               throw new BusinessException(String.format("The username [%s] is already used", username));
            }
      );
      AddCustomerResponse response = modelMapper.map(customerRepository.save(bo), AddCustomerResponse.class);
      response.setMessage(String.format("Customer : [identity= %s,First Name= %s, Last Name= %s, username= %s] was
created with success",
            response.getIdentityRef(), response.getFirstname(), response.getLastname(), response.getUsername()));
      return response;
   }


   @Override
```

```
    public UpdateCustomerResponse updateCustomer(String identityRef, UpdateCustomerRequest
updateCustomerRequest) {
        Customer customerToPersist = modelMapper.map(updateCustomerRequest, Customer.class);
        Customer customerFound = customerRepository.findAll().stream().filter(bo ->
bo.getIdentityRef().equals(identityRef)).findFirst().orElseThrow(
                () -> new BusinessException(String.format("No Customer with identity [%s] exist !", identityRef))
        );
        customerToPersist.setId(customerFound.getId());
        customerToPersist.setIdentityRef(identityRef);
        UpdateCustomerResponse updateCustomerResponse =
modelMapper.map(customerRepository.save(customerToPersist), UpdateCustomerResponse.class);
        updateCustomerResponse.setMessage(String.format("Customer identity %s is updated with success", identityRef));
        return updateCustomerResponse;
    }

    @Override
    public CustomerDto getCustomByIdentity(String identity) {
        return modelMapper.map(customerRepository.findByIdentityRef(identity).orElseThrow(
                    () -> new BusinessException(String.format("No Customer with identity [%s] exist !", identity))),
                CustomerDto.class);
    }

    @Override
    public String deleteCustomerByIdentityRef(String identityRef) {
        if (identityRef == null || identityRef.isEmpty())
            throw new BusinessException("Enter a correct identity customer");
        Customer customerFound = customerRepository.findAll().stream().filter(customer ->
customer.getIdentityRef().equals(identityRef)).findFirst().orElseThrow(
                () -> new BusinessException(String.format("No customer with identity %s exist in database", identityRef))
        );
        customerRepository.delete(customerFound);
        return String.format("Customer with identity %s is deleted with success", identityRef);

    }
}
```

- La classe BankAccountServiceImpl :

```
package ma.formations.graphql.service;


import lombok.AllArgsConstructor;
import ma.formations.graphql.dao.BankAccountRepository;
import ma.formations.graphql.dao.CustomerRepository;
import ma.formations.graphql.dtos.bankaccount.AddBankAccountRequest;
import ma.formations.graphql.dtos.bankaccount.AddBankAccountResponse;
import ma.formations.graphql.dtos.bankaccount.BankAccountDto;
import ma.formations.graphql.enums.AccountStatus;
import ma.formations.graphql.service.exception.BusinessException;
import ma.formations.graphql.service.model.BankAccount;
import ma.formations.graphql.service.model.Customer;
import org.modelmapper.ModelMapper;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.Date;
```

```
import java.util.List;
import java.util.stream.Collectors;

@Service
@Transactional
@AllArgsConstructor
public class BankAccountServiceImpl implements IBankAccountService {
   private final BankAccountRepository bankAccountRepository;
   private final CustomerRepository customerRepository;
   private ModelMapper modelMapper;


   @Override
   public AddBankAccountResponse saveBankAccount(AddBankAccountRequest dto) {
      BankAccount bankAccount = modelMapper.map(dto, BankAccount.class);
      Customer customerP =
customerRepository.findByIdentityRef(bankAccount.getCustomer().getIdentityRef()).orElseThrow(
            () -> new BusinessException(String.format("No customer with the identity: %s exist",
dto.getCustomerIdentityRef())));
      bankAccount.setAccountStatus(AccountStatus.OPENED);
      bankAccount.setCustomer(customerP);
      bankAccount.setCreatedAt(new Date());
      AddBankAccountResponse response = modelMapper.map(bankAccountRepository.save(bankAccount),
AddBankAccountResponse.class);
      response.setMessage(String.format("RIB number [%s] for the customer [%s] has been successfully created",
dto.getRib(), dto.getCustomerIdentityRef()));
      return response;
   }

   @Override
   public List<BankAccountDto> getAllBankAccounts() {
      return bankAccountRepository.findAll().stream().
            map(bankAccount -> modelMapper.map(bankAccount, BankAccountDto.class)).
            collect(Collectors.toList());
   }

   @Override
   public BankAccountDto getBankAccountByRib(String rib) {
      return modelMapper.map(bankAccountRepository.findByRib(rib).orElseThrow(
            () -> new BusinessException(String.format("No Bank Account with rib [%s] exist", rib))), BankAccountDto.class);
   }
}
```

- La classe **TransactionServiceImpl** :

```
package ma.formations.graphql.service;

import lombok.AllArgsConstructor;
import ma.formations.graphql.dao.BankAccountRepository;
import ma.formations.graphql.dao.BankAccountTransactionRepository;
import ma.formations.graphql.dao.UserRepository;
import ma.formations.graphql.dtos.transaction.AddWirerTransferRequest;
import ma.formations.graphql.dtos.transaction.AddWirerTransferResponse;
import ma.formations.graphql.dtos.transaction.GetTransactionListRequest;
import ma.formations.graphql.dtos.transaction.TransactionDto;
import ma.formations.graphql.enums.AccountStatus;
```

```java
import ma.formations.graphql.enums.TransactionType;
import ma.formations.graphql.service.exception.BusinessException;
import ma.formations.graphql.service.model.BankAccount;
import ma.formations.graphql.service.model.BankAccountTransaction;
import ma.formations.graphql.service.model.GetTransactionListBo;
import ma.formations.graphql.service.model.User;
import org.modelmapper.ModelMapper;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.Date;
import java.util.List;
import java.util.stream.Collectors;

@Service
@Transactional
@AllArgsConstructor
public class TransactionServiceImpl implements ITransactionService {

    private final BankAccountRepository bankAccountRepository;
    private final BankAccountTransactionRepository bankAccountTransactionRepository;

    private final UserRepository userRepository;

    private ModelMapper modelMapper;


    @Override
    public AddWirerTransferResponse wiredTransfer(AddWirerTransferRequest dto) {

        BankAccountTransaction transactionFrom = BankAccountTransaction.builder().
                amount(dto.getAmount()).
                transactionType(TransactionType.DEBIT).
                bankAccount(BankAccount.builder().rib(dto.getRibFrom()).build()).
                user(new User(dto.getUsername())).
                build();

        BankAccountTransaction transactionTo = BankAccountTransaction.builder().
                amount(dto.getAmount()).
                transactionType(TransactionType.CREDIT).
                bankAccount(BankAccount.builder().rib(dto.getRibTo()).build()).
                user(new User(dto.getUsername())).
                build();

        String username = transactionFrom.getUser().getUsername();
        String ribFrom = transactionFrom.getBankAccount().getRib();
        String ribTo = transactionTo.getBankAccount().getRib();
        Double amount = transactionFrom.getAmount();

        User user = userRepository.findByUsername(username).
                orElseThrow(() -> new BusinessException(String.format("User [%s] doesn't exist", username)));

        BankAccount bankAccountFrom = bankAccountRepository.findByRib(ribFrom).
                orElseThrow(() -> new BusinessException(String.format("No bank account have the rib %s", ribFrom)));

        BankAccount bankAccountTo = bankAccountRepository.findByRib(ribTo).
                orElseThrow(() -> new BusinessException(String.format("No bank account have the rib %s", ribTo)));
```

```
        checkBusinessRules(bankAccountFrom, bankAccountTo, amount);
        //On débite le compte demandeur
        bankAccountFrom.setAmount(bankAccountFrom.getAmount() - amount);
        //On crédite le compte destinataire
        bankAccountTo.setAmount(bankAccountTo.getAmount() + amount);

        transactionFrom.setCreatedAt(new Date());
        transactionFrom.setUser(user);
        transactionFrom.setBankAccount(bankAccountFrom);

        transactionTo.setCreatedAt(new Date());
        transactionTo.setUser(user);
        transactionTo.setBankAccount(bankAccountTo);
        bankAccountTransactionRepository.save(transactionFrom);
        bankAccountTransactionRepository.save(transactionTo);
        return AddWirerTransferResponse.builder().
                message(String.format("the transfer of an amount of %s from the %s bank account to %s was carried out
successfully",
                        dto.getAmount(), dto.getRibFrom(), dto.getRibTo())).
                transactionFrom(modelMapper.map(transactionFrom, TransactionDto.class)).
                transactionTo(modelMapper.map(transactionTo, TransactionDto.class)).
                build();
    }

    private void checkBusinessRules(BankAccount bankAccountFrom, BankAccount bankAccountTo, Double amount) {

        if (bankAccountFrom.getAccountStatus().equals(AccountStatus.CLOSED))
            throw new BusinessException(String.format("the bank account %s is closed !!", bankAccountFrom.getRib()));

        if (bankAccountFrom.getAccountStatus().equals(AccountStatus.BLOCKED))
            throw new BusinessException(String.format("the bank account %s is blocked !!", bankAccountFrom.getRib()));

        if (bankAccountTo.getAccountStatus().equals(AccountStatus.CLOSED))
            throw new BusinessException(String.format("the bank account %s is closed !!", bankAccountTo.getRib()));

        if (bankAccountTo.getAccountStatus().equals(AccountStatus.BLOCKED))
            throw new BusinessException(String.format("the bank account %s is blocked !!", bankAccountTo.getRib()));

        if (bankAccountFrom.getAmount() < amount)
            throw new BusinessException(String.format("the balance of account number %s is less than %s",
bankAccountFrom.getRib(), amount));
    }


    @Override
    public List<TransactionDto> getTransactions(GetTransactionListRequest requestDTO) {
        GetTransactionListBo data = modelMapper.map(requestDTO, GetTransactionListBo.class);
        return bankAccountTransactionRepository.findByBankAccount_RibAndCreatedAtBetween(
                        data.getRib(), data.getDateFrom(), data.getDateTo()).
                stream().map(bo -> modelMapper.map(bo, TransactionDto.class)).collect(Collectors.toList());
    }
}
```

k. Initialisation de la base de données

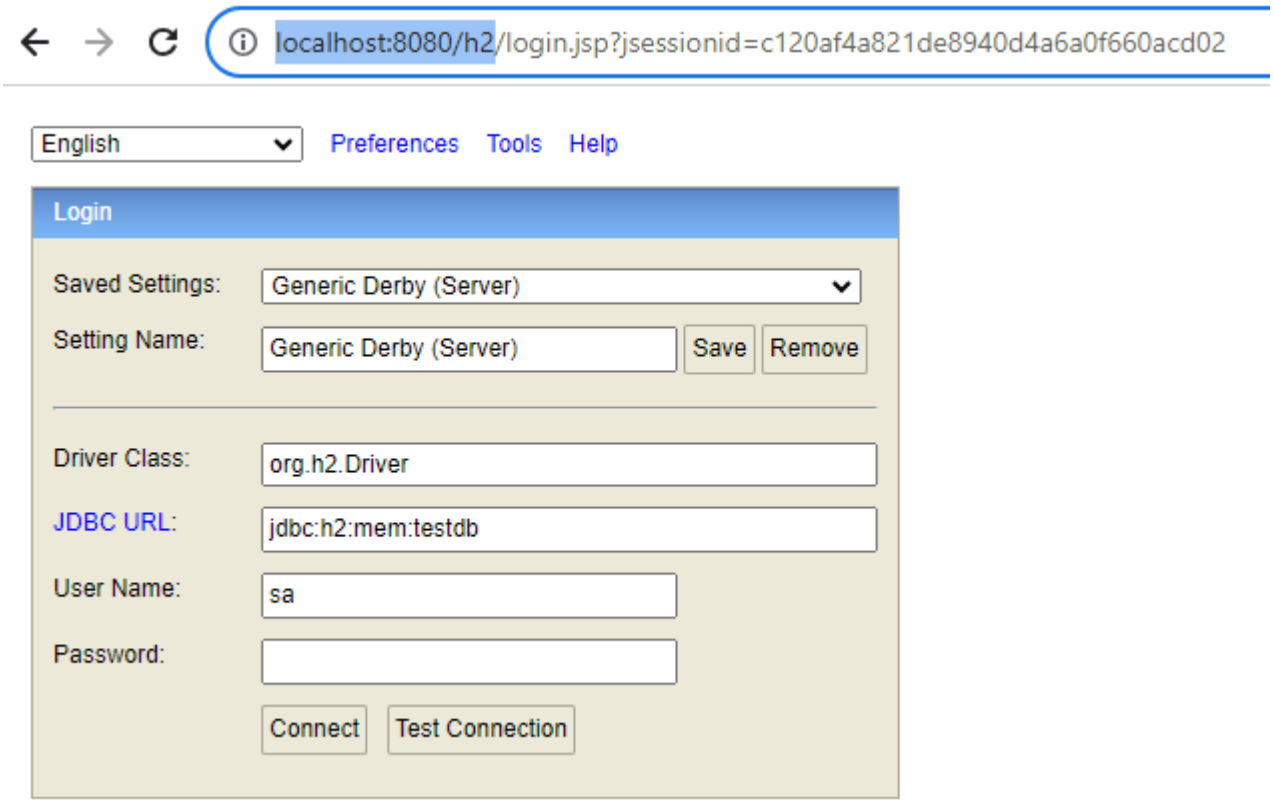
- Au niveau de la classe de démarrage de *Spring Boot*, ajouter la méthode suivante :

```
@Bean
CommandLineRunner initDataBase(ICustomerService customerService,
                               IBankAccountService bankAccountService,
                               ITransactionService transactionService) {

  return args -> {
    customerService.createCustomer(AddCustomerRequest.builder().
          username("user1").
          identityRef("A100").
          firstname("FIRST_NAME1").
          lastname("LAST_NAME1").
          build());

    bankAccountService.saveBankAccount(AddBankAccountRequest.builder().
          rib("RIB_1").
          amount(1000000d).
          customerIdentityRef("A100").
          build());
    bankAccountService.saveBankAccount(AddBankAccountRequest.builder().
          rib("RIB_11").
          amount(2000000d).
          customerIdentityRef("A100").
          build());

    customerService.createCustomer(AddCustomerRequest.builder().
          username("user2").
          identityRef("A200").
          firstname("FIRST_NAME2").
          lastname("LAST_NAME2").
          build());

    bankAccountService.saveBankAccount(AddBankAccountRequest.builder().
          rib("RIB_2").
          amount(2000000d).
          customerIdentityRef("A200").
          build());


    customerService.createCustomer(AddCustomerRequest.builder().
          username("user3").
          identityRef("A900").
          firstname("FIRST_NAME9").
          lastname("LAST_NAME9").
          build());

    bankAccountService.saveBankAccount(AddBankAccountRequest.builder().
          rib("RIB_9").
          amount(-25000d).
          customerIdentityRef("A900").
          build());


    customerService.createCustomer(AddCustomerRequest.builder().
          username("user4").
          identityRef("A800").
```

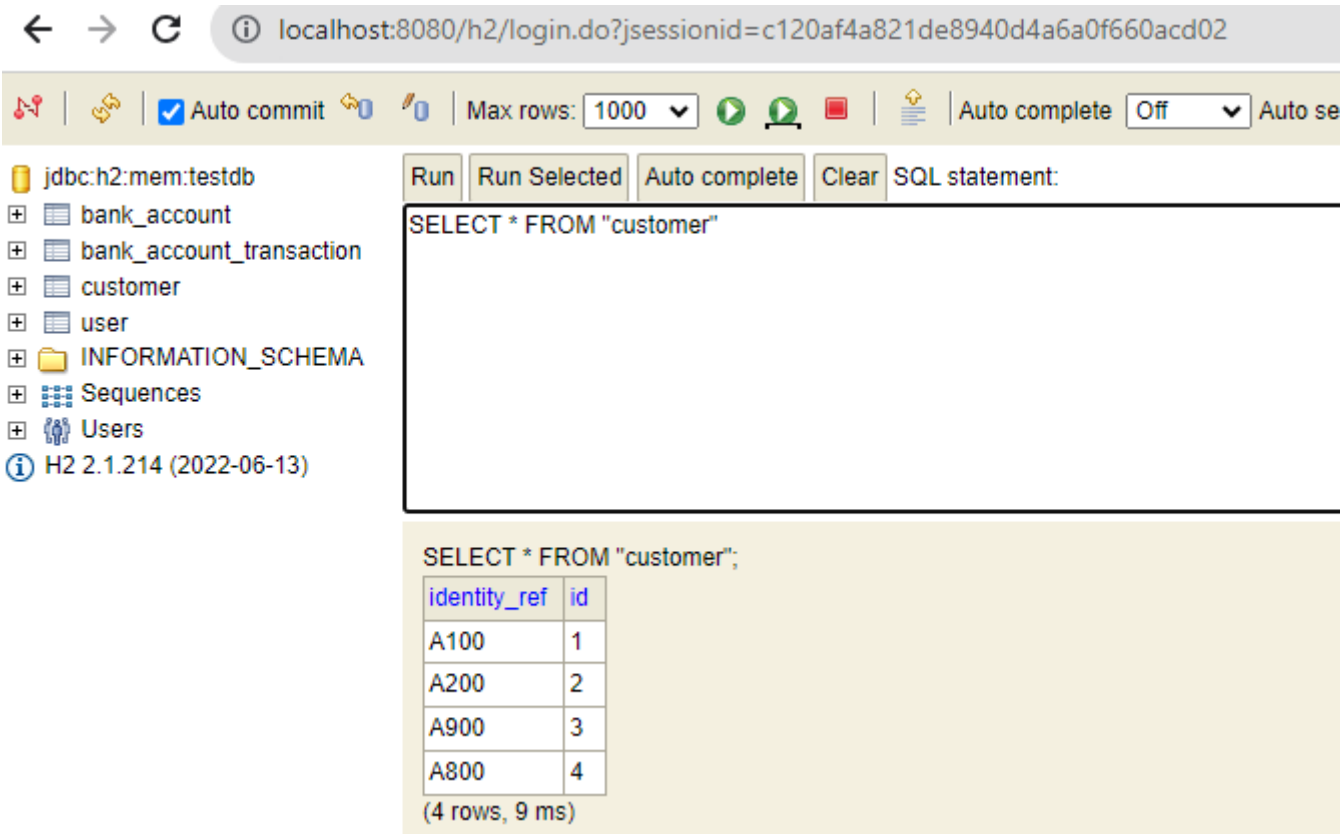
```
            firstname("FIRST_NAME8").
            lastname("LAST_NAME8").
            build());

    bankAccountService.saveBankAccount(AddBankAccountRequest.builder().
            rib("RIB_8").
            amount(0.0).
            customerIdentityRef("A800").
            build());

    transactionService.wiredTransfer(AddWirerTransferRequest.builder().
            ribFrom("RIB_1").
            ribTo("RIB_2").
            amount(10000.0).
            username("user1").
            build());

    transactionService.wiredTransfer(AddWirerTransferRequest.builder().
            ribFrom("RIB_1").
            ribTo("RIB_9").
            amount(20000.0).
            username("user1").
            build());

    transactionService.wiredTransfer(AddWirerTransferRequest.builder().
            ribFrom("RIB_1").
            ribTo("RIB_8").
            amount(500.0).
            username("user1").
            build());


    transactionService.wiredTransfer(AddWirerTransferRequest.builder().
            ribFrom("RIB_2").
            ribTo("RIB_11").
            amount(300.0).
            username("user2").
            build());
  };
}
```

I. Tester la création des tables

- Exécuter la méthode main de votre classe de démarrage et vérifier que les tables ont été bien crées en suivant les étapes suivantes :
    - ✓ Aller au lien http://localhost:8080/h2, la fenêtre suivante s'affiche :



- Cliquer sur *Connect* (vérifier avant de cliquer sur Connect que vous avez entré la bonne classe Driver de la base de données H2 (***org.h2.Driver***) et la bonne URL (***jdbc:h2:mem:testdb***)) : la console de H2 s'affiche :



- Vérifier la création des 04 tables (USER, CUSTOMER, BANK_ACCOUNT et BANK_ACCOUNT_TRANSACTION).
- Vérifier également que les tables ont été bien initialisées avec les données.

V. Configuration de GraphQL

a. Activation du GraphQL

- Ajouter la ligne suivante au niveau du fichier *application.properties* :

```
spring.graphql.graphiql.enabled=true
```

- Dans ***resources***, créer le dossier ***graphql*** et dans ce dernier, créer le fichier ***schema.graphqls*** suivant :

```
type Query{
   customers:[CustomerDto]
   customerByIdentity(identity:String):CustomerDto
   bankAccounts : [BankAccountDto]
   bankAccountByRib (rib:String):BankAccountDto
   getTransactions (dto:GetTransactionListRequest):[TransactionDto]
}

type Mutation {
   createCustomer(dto:AddCustomerRequest):AddCustomerResponse
   addBankAccount(dto:AddBankAccountRequest):AddBankAccountResponse
   addWirerTransfer(dto:AddWirerTransferRequest):AddWiredTransferResponse
   updateCustomer(identityRef:String,dto:UpdateCustomerRequest):UpdateCustomerResponse
   deleteCustomer(identityRef:String):String
}

type CustomerDto {
   username : String,
   identityRef : String,
   firstname:String,
   lastname:String,
}

type BankAccountDto {
   rib:String,
   amount:Float,
   createdAt:String,
   accountStatus:AccountStatus,
   customer:CustomerDto,
}

type TransactionDto {
   createdAt:String,
   transactionType:TransactionType,
   amount:Float,
   bankAccount:BankAccountDto
   user:userDto
}

type userDto {
   username:String
```

```
    firstname:String
    lastname:String
}

enum AccountStatus {
    OPENED, CLOSED, BLOCKED
}
enum TransactionType {
    CREDIT,DEBIT
}

input AddCustomerRequest {
    username : String,
    identityRef : String,
    firstname:String,
    lastname:String
}

input UpdateCustomerRequest {
    username : String,
    firstname:String,
    lastname:String
}

type UpdateCustomerResponse {
    id:Int,
    message:String,
    username:String,
    identityRef:String,
    firstname:String,
    lastname:String
}

type AddCustomerResponse {
    message:String,
    username:String,
    identityRef:String,
    firstname:String,
    lastname:String
}

input AddBankAccountRequest {
    rib:String,
    amount:Float,
    customerIdentityRef:String
}

type AddBankAccountResponse {
    message:String,
    rib:String,
    amount:Float,
    createdAt:String,
    accountStatus:AccountStatus,
    customer:CustomerDto
}

input AddWirerTransferRequest {
```

```
    ribFrom:String,
    ribTo:String,
    amount:Float,
    username:String
}

type AddWiredTransferResponse {
    message:String,
    transactionFrom:TransactionDto,
    transactionTo:TransactionDto
}

input GetTransactionListRequest {
    rib:String,
    dateTo : String,
    dateFrom:String
}
```

- Créer le package ***ma.formations.graphql.presentation*** et ensuite créer les classes suivantes : CustomerGraphqlController, BankAccountGraphqlController et TransactionGraphqlController :

❖ La classe **CustomerGraphqlController** :

```
package ma.formations.graphql.presentation;

import ma.formations.graphql.dtos.customer.*;
import ma.formations.graphql.service.ICustomerService;
import org.springframework.graphql.data.method.annotation.Argument;
import org.springframework.graphql.data.method.annotation.MutationMapping;
import org.springframework.graphql.data.method.annotation.QueryMapping;
import org.springframework.stereotype.Controller;

import java.util.List;

@Controller
public class CustomerGraphqlController {

    private final ICustomerService customerService;

    public CustomerGraphqlController(ICustomerService customerService) {
        this.customerService = customerService;
    }

    @QueryMapping
    List<CustomerDto> customers() {
        return customerService.getAllCustomers();
    }

    @QueryMapping
    CustomerDto customerByIdentity(@Argument String identity) {
        return customerService.getCustomByIdentity(identity);
    }

    @MutationMapping
```

```
    public AddCustomerResponse createCustomer(@Argument("dto") AddCustomerRequest dto) {
        return customerService.createCustomer(dto);
    }

    @MutationMapping
    public UpdateCustomerResponse updateCustomer(@Argument("identityRef") String identityRef, @Argument("dto")
UpdateCustomerRequest dto) {
        return customerService.updateCustomer(identityRef, dto);
    }

    @MutationMapping
    public String deleteCustomer(@Argument("identityRef") String identityRef) {
        return customerService.deleteCustomerByIdentityRef(identityRef);
    }

}
```

- La classe **BankAccountGraphqlController** :

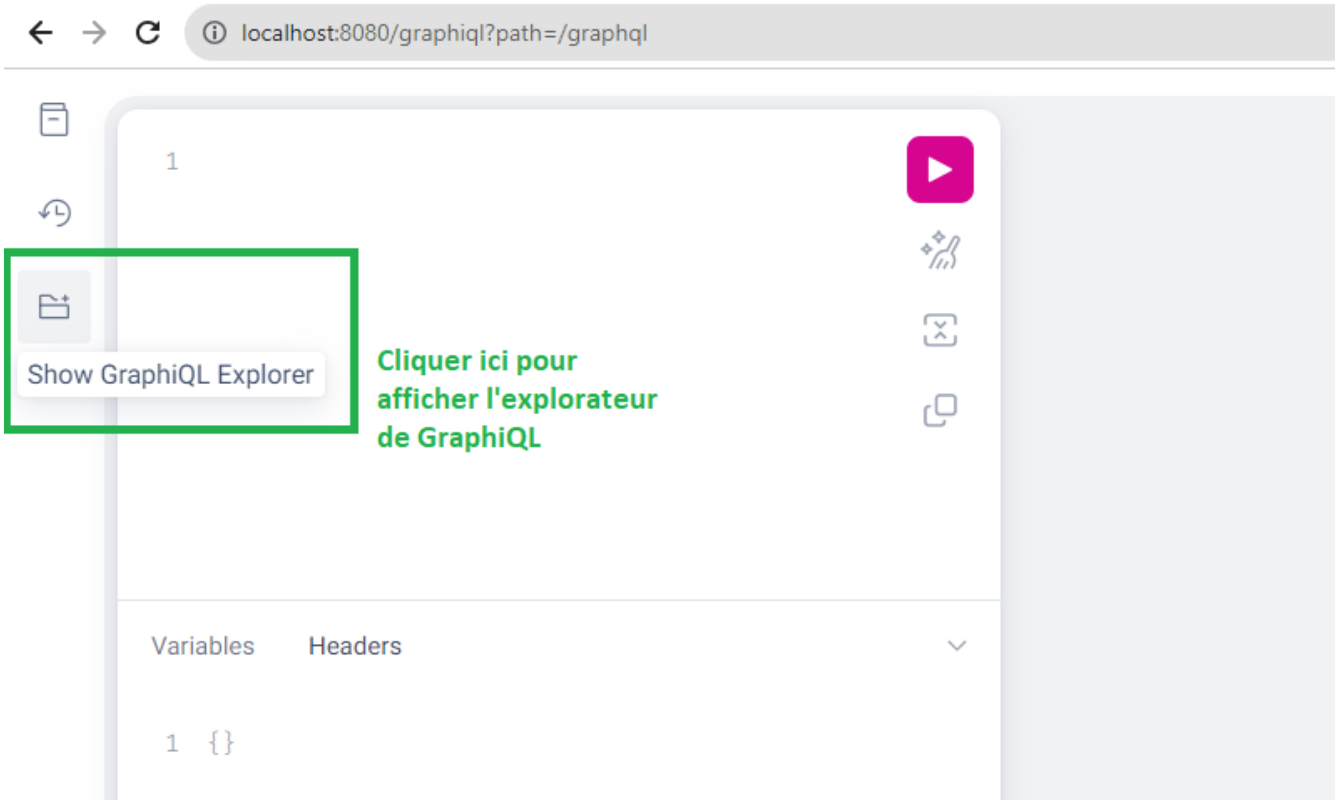
```
package ma.formations.graphql.presentation;

import lombok.AllArgsConstructor;
import ma.formations.graphql.dtos.bankaccount.AddBankAccountRequest;
import ma.formations.graphql.dtos.bankaccount.AddBankAccountResponse;
import ma.formations.graphql.dtos.bankaccount.BankAccountDto;
import ma.formations.graphql.service.IBankAccountService;
import org.springframework.graphql.data.method.annotation.Argument;
import org.springframework.graphql.data.method.annotation.MutationMapping;
import org.springframework.graphql.data.method.annotation.QueryMapping;
import org.springframework.stereotype.Controller;

import java.util.List;

@Controller
@AllArgsConstructor
public class BankAccountGraphqlController {
    private final IBankAccountService bankAccountService;

    @QueryMapping
    List<BankAccountDto> bankAccounts() {
        return bankAccountService.getAllBankAccounts();
    }

    @QueryMapping
    BankAccountDto bankAccountByRib(@Argument String rib) {
        return bankAccountService.getBankAccountByRib(rib);
    }


    @MutationMapping
    public AddBankAccountResponse addBankAccount(@Argument("dto") AddBankAccountRequest dto) {
        return bankAccountService.saveBankAccount(dto);
    }

}
```

❖ La classe **TransactionGraphqlController** :

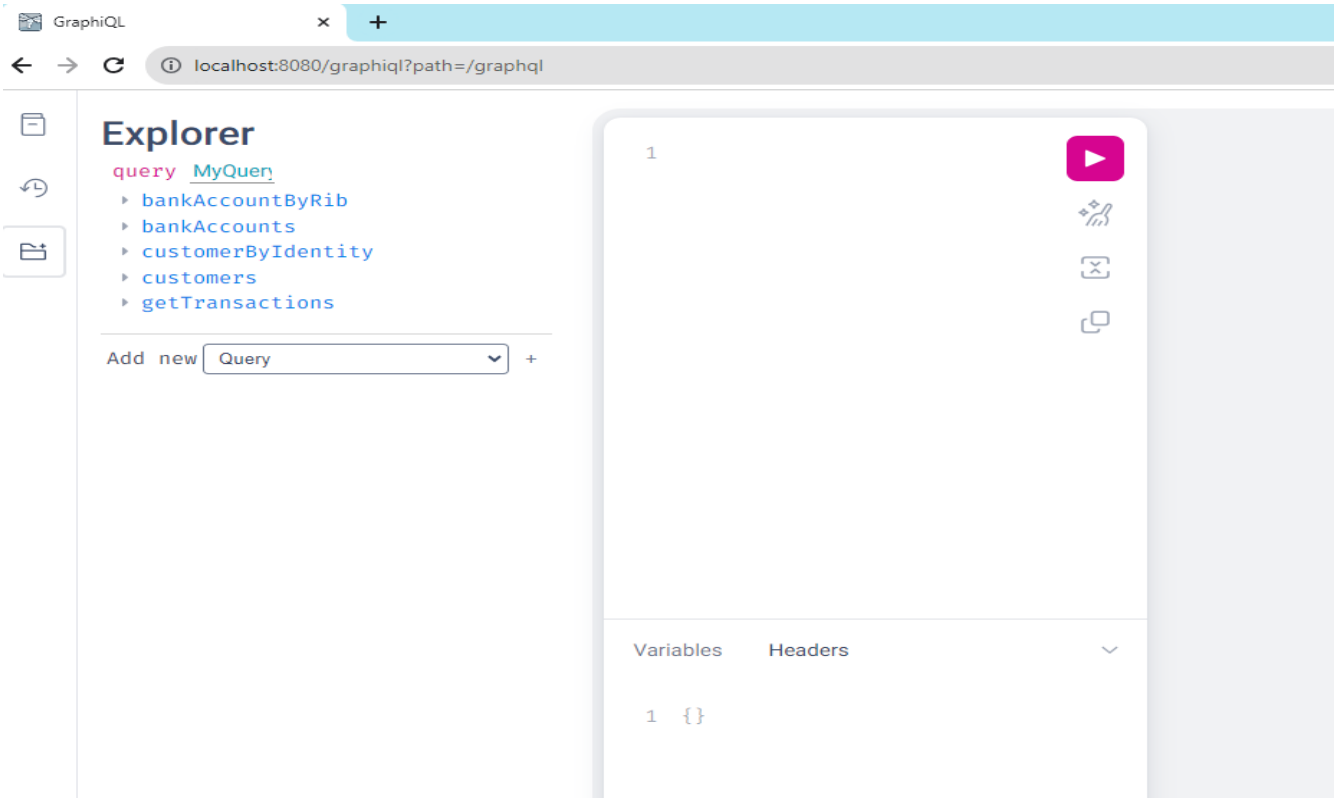
```
package ma.formations.graphql.presentation;

import lombok.AllArgsConstructor;
import ma.formations.graphql.common.CommonTools;
import ma.formations.graphql.dtos.transaction.AddWirerTransferRequest;
import ma.formations.graphql.dtos.transaction.AddWirerTransferResponse;
import ma.formations.graphql.dtos.transaction.GetTransactionListRequest;
import ma.formations.graphql.dtos.transaction.TransactionDto;
import ma.formations.graphql.service.ITransactionService;
import ma.formations.graphql.service.exception.BusinessException;
import org.springframework.graphql.data.method.annotation.Argument;
import org.springframework.graphql.data.method.annotation.MutationMapping;
import org.springframework.graphql.data.method.annotation.QueryMapping;
import org.springframework.stereotype.Controller;

import java.util.Date;
import java.util.List;

@Controller
@AllArgsConstructor

public class TransactionGraphqlController {

   private ITransactionService transactionService;
   private CommonTools commonTools;

   @MutationMapping
   public AddWirerTransferResponse addWirerTransfer(@Argument("dto") AddWirerTransferRequest dto) {
      return transactionService.wiredTransfer(dto);
   }

   @QueryMapping
   public List<TransactionDto> getTransactions(@Argument GetTransactionListRequest dto) {
      return transactionService.getTransactions(dto);
   }
}
```

b. Accéder à l'explorateur de GraphQL (l'interface web GraphiQL)

- Démarrer la méthode *main* et aller à l'url suivante : http://localhost:8080/graphiql?path=/graphql, la page suivante sera affichée :



- Cliquer sur l'icône ci-dessus pour afficher l'explorateur de GraphiQL :

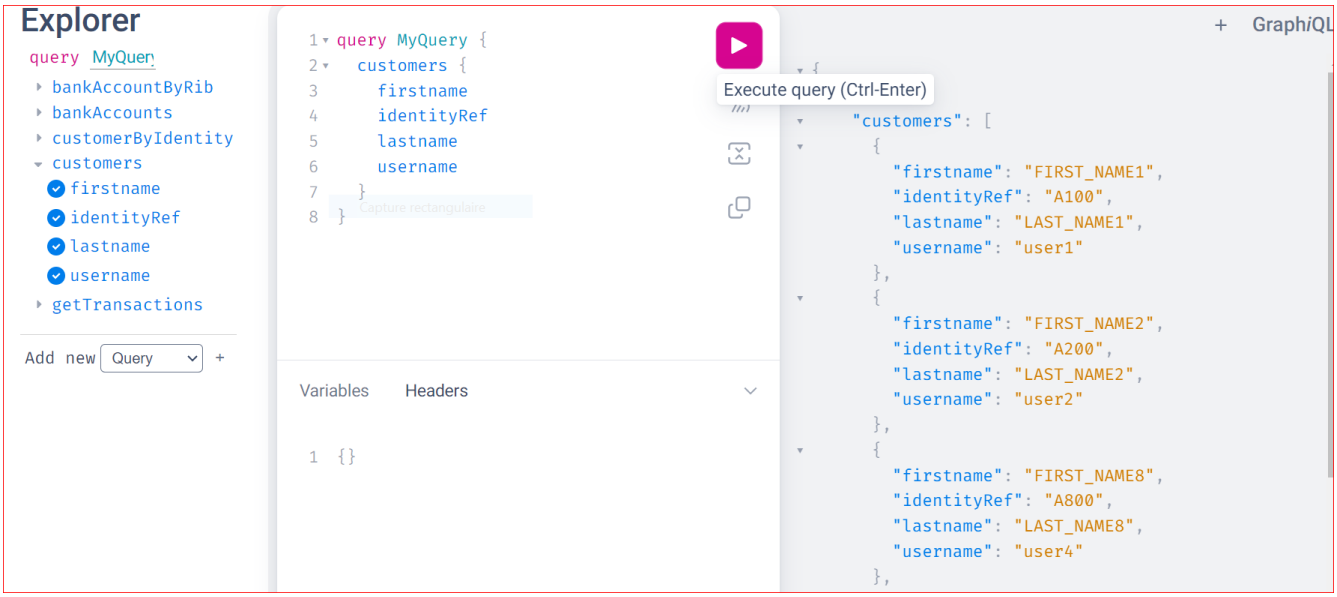


c. Tester la requête customers

- Utiliser l'explorateur de l'interface GraphiQL, cliquer sur le service *customers* et cocher les données que vous voulez récupérer et ensuite cliquer sur la flèche au centre pour exécuter la requête :

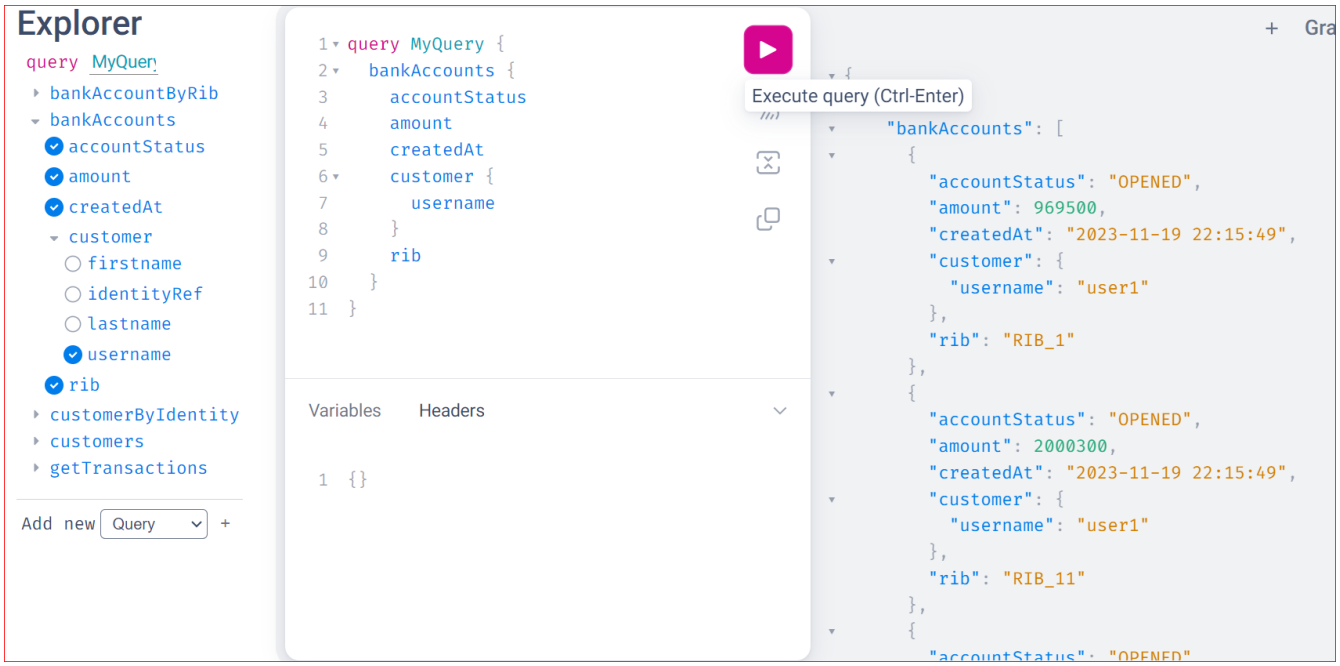
d. Tester la requête customerByIdentity

- Utiliser l'explorateur de l'interface GraphiQL, cliquer sur le service *customerByIdentity,* cocher les données que vous voulez récupérer, entrer le numéro d'identité du client que vous voulez consulter et ensuite cliquer sur la flèche au centre pour exécuter la requête :



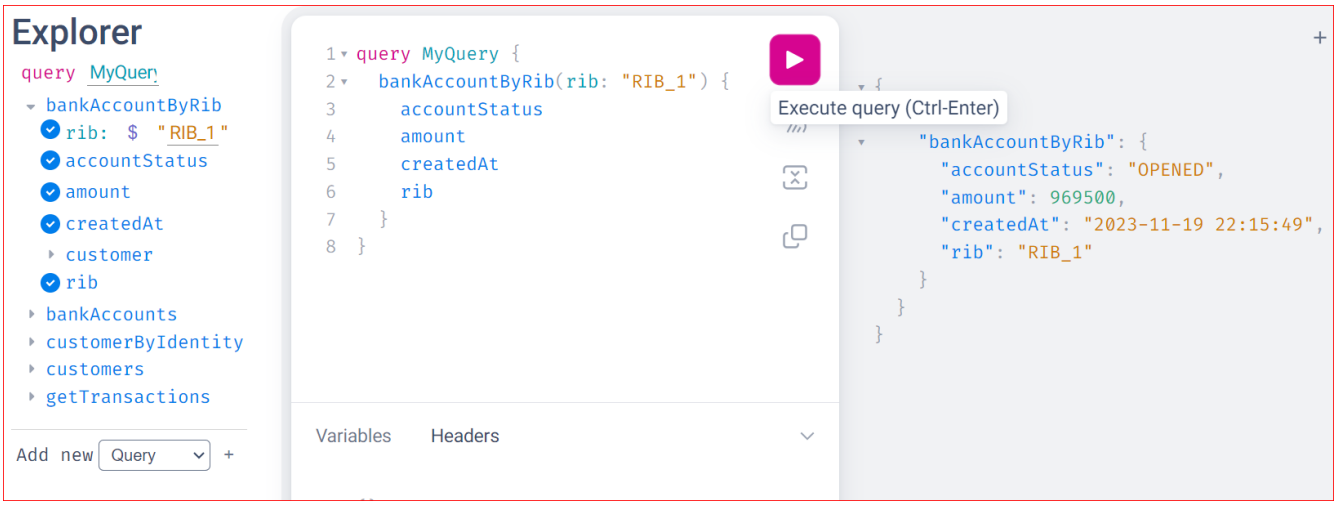
e. Tester la requête bankAccounts

- Utiliser l'explorateur de l'interface GraphiQL, cliquer sur le service **bankAccounts** et cocher les données que vous voulez récupérer et ensuite cliquer sur la flèche au centre pour exécuter la requête :



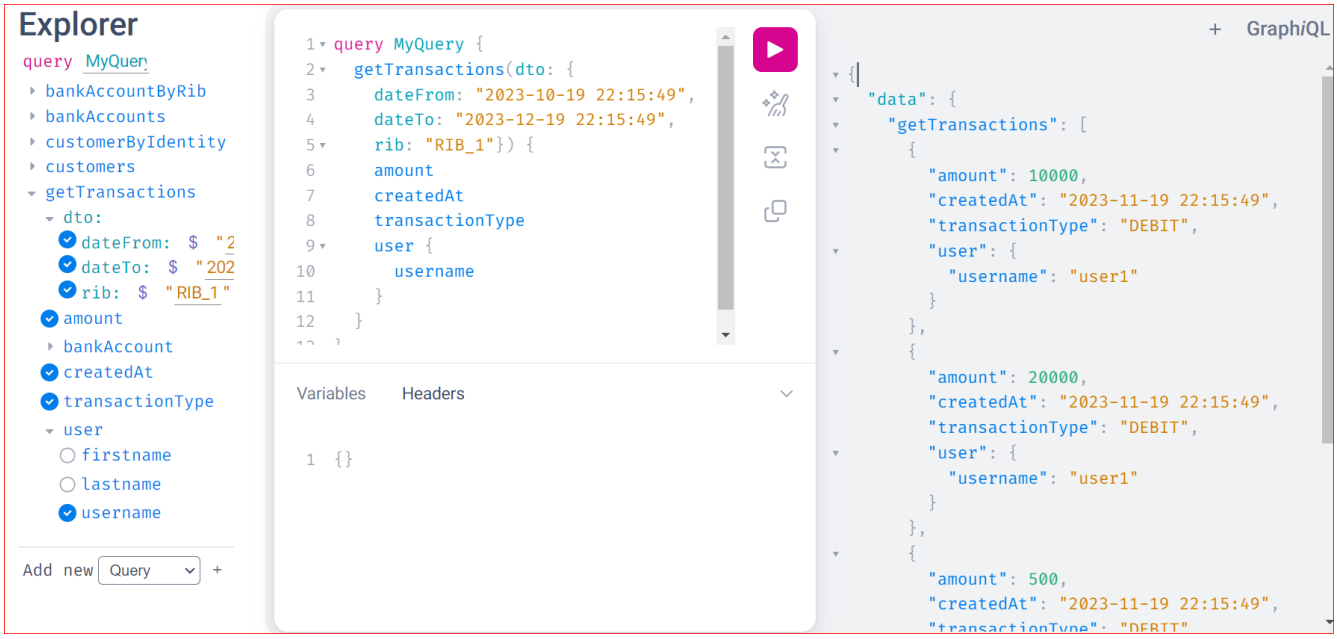
f. Tester la requête bankAccountByRib

- Utiliser l'explorateur de l'interface GraphiQL, cliquer sur le service **bankAccountByRib,** cocher les données que vous voulez récupérer, entrer le RIB du compte que vous voulez consulter et ensuite cliquer sur la flèche au centre pour exécuter la requête :



g. Tester la requête getTransactions

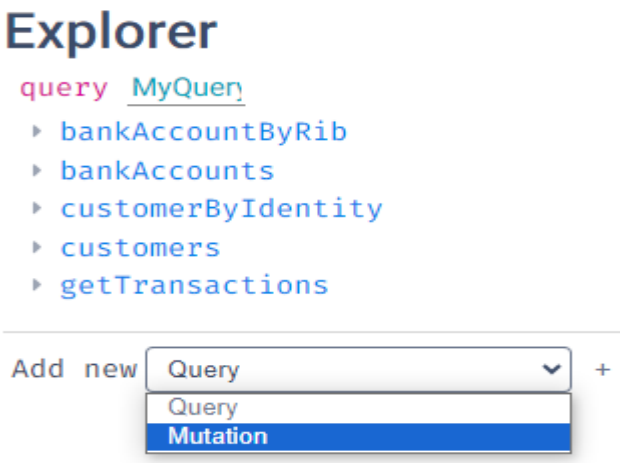
- Utiliser l'explorateur de l'interface GraphiQL, cliquer sur le service **getTransactions** et cocher les données que vous voulez récupérer, entrer la date de début, entrer la date de fin, entrer le numéro de RIB et ensuite cliquer sur la flèche au centre pour exécuter la requête :



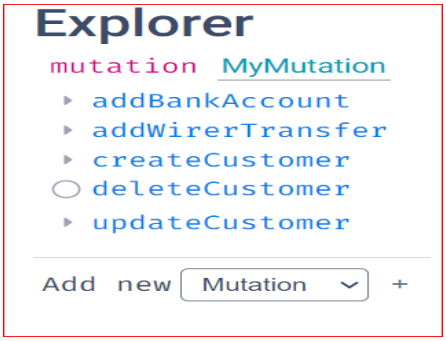
NB : La date doit respecter le format ***yyyy-MM-dd HH:mm:ss***

h. Tester la requête createCustomer

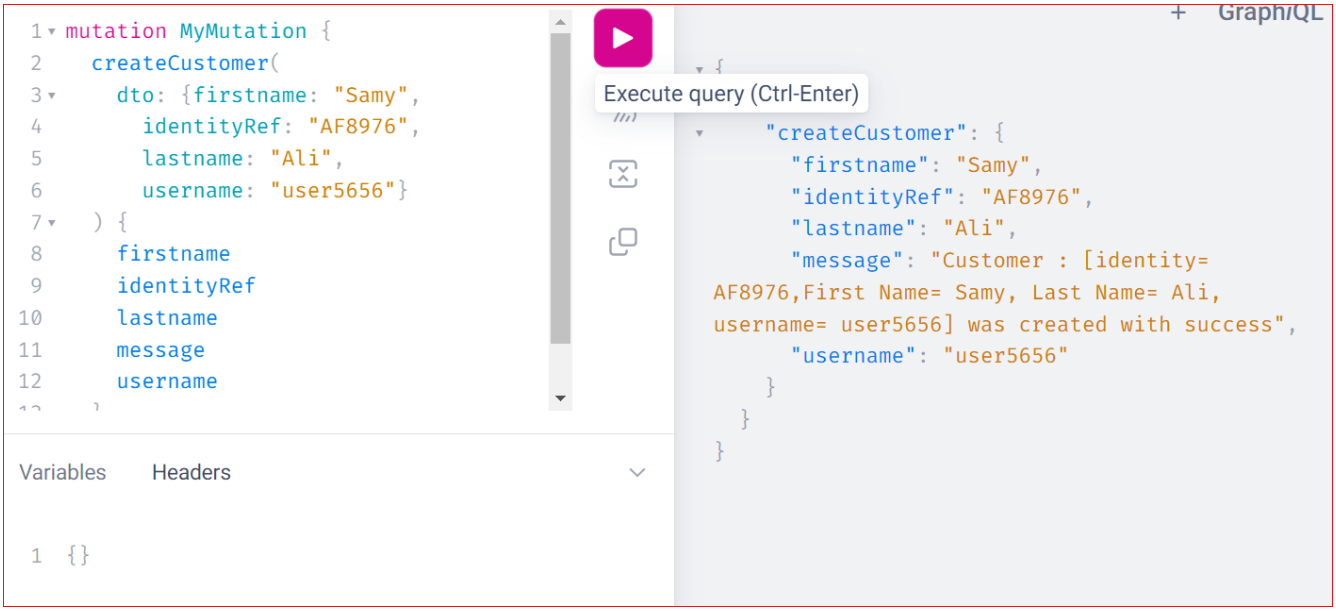
- Au niveau de la liste déroulante, choisir *Mutation* au lieu de *Query* comme le montre la figure suivante :



- Cliquer ensuite sur (+), les services suivants seront affichés :



- Cliquer sur *createCustomer*, sélectionner les données que vous voulez récupérer, entrer les données du client à créer (firstname, lastname, identityRef et username) et cliquer ensuite sur la flèche au centre pour exécuter la requête :

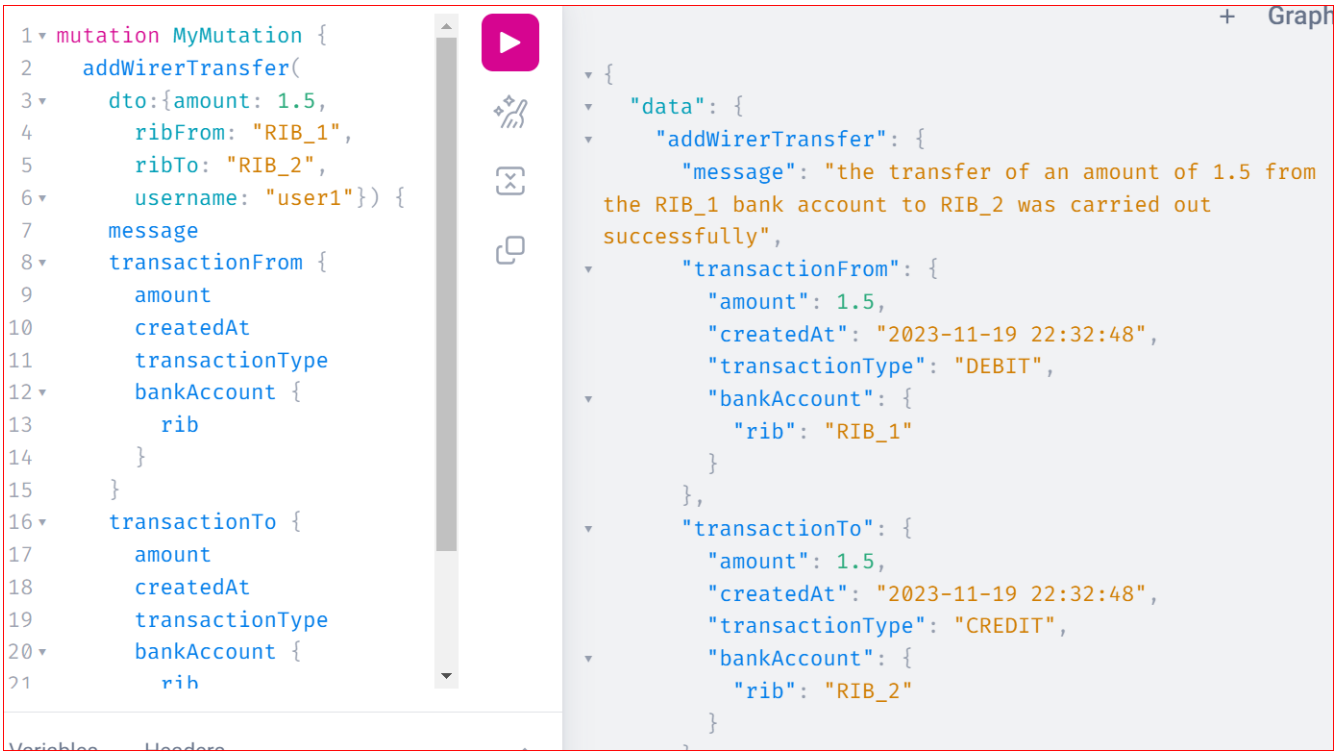


i. Tester la requête addBankAccount

- Cliquer sur *addBankAccount*, sélectionner les données que vous voulez récupérer, entrer les données du compte bancaire à créer (RIB, le montant et l'identité du client) et cliquer ensuite sur la flèche au centre pour exécuter la requête :

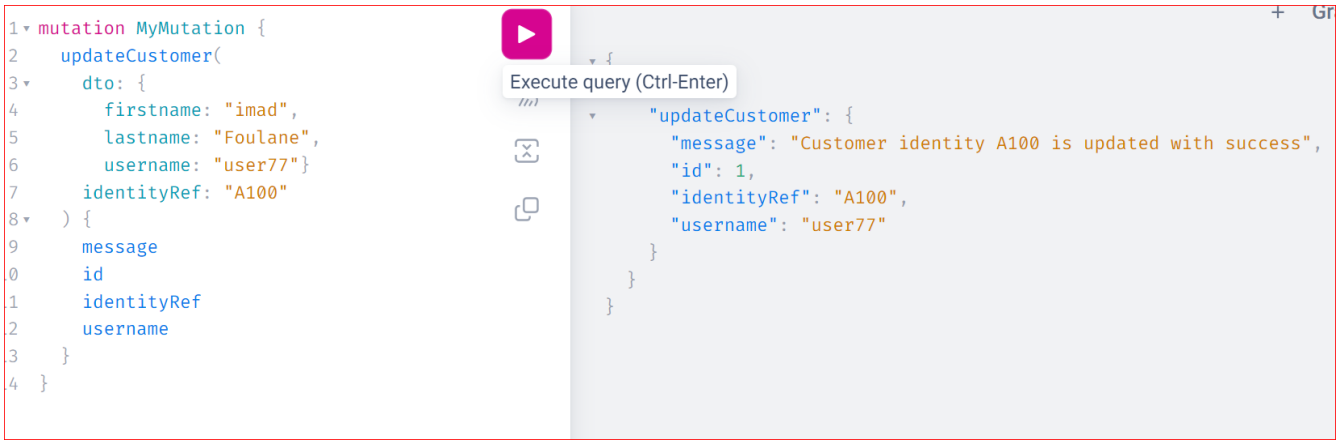
j. Tester la requête addWirerTransfer

- Cliquer sur ***addWirerTransfer***, sélectionner les données que vous voulez récupérer, entrer les données du virement à créer (le RIB émetteur, le RIB destinataire, le montant et l'utilisateur) et cliquer ensuite sur la flèche au centre pour exécuter la requête :



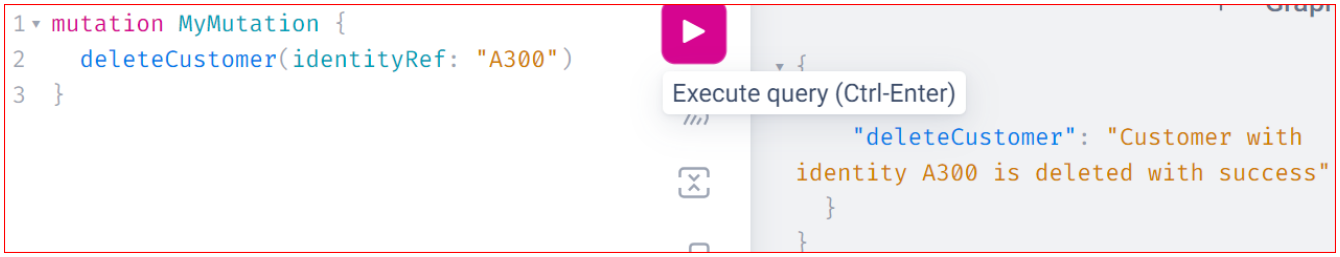
k. Tester la requête updateCustomer

- Cliquer sur ***updateCustomer***, entrer firstname, lastname, username et identityRef
- Choisir les données que vous voulez récupérer et cliquer ensuite sur la flèche au centre pour exécuter la requête :



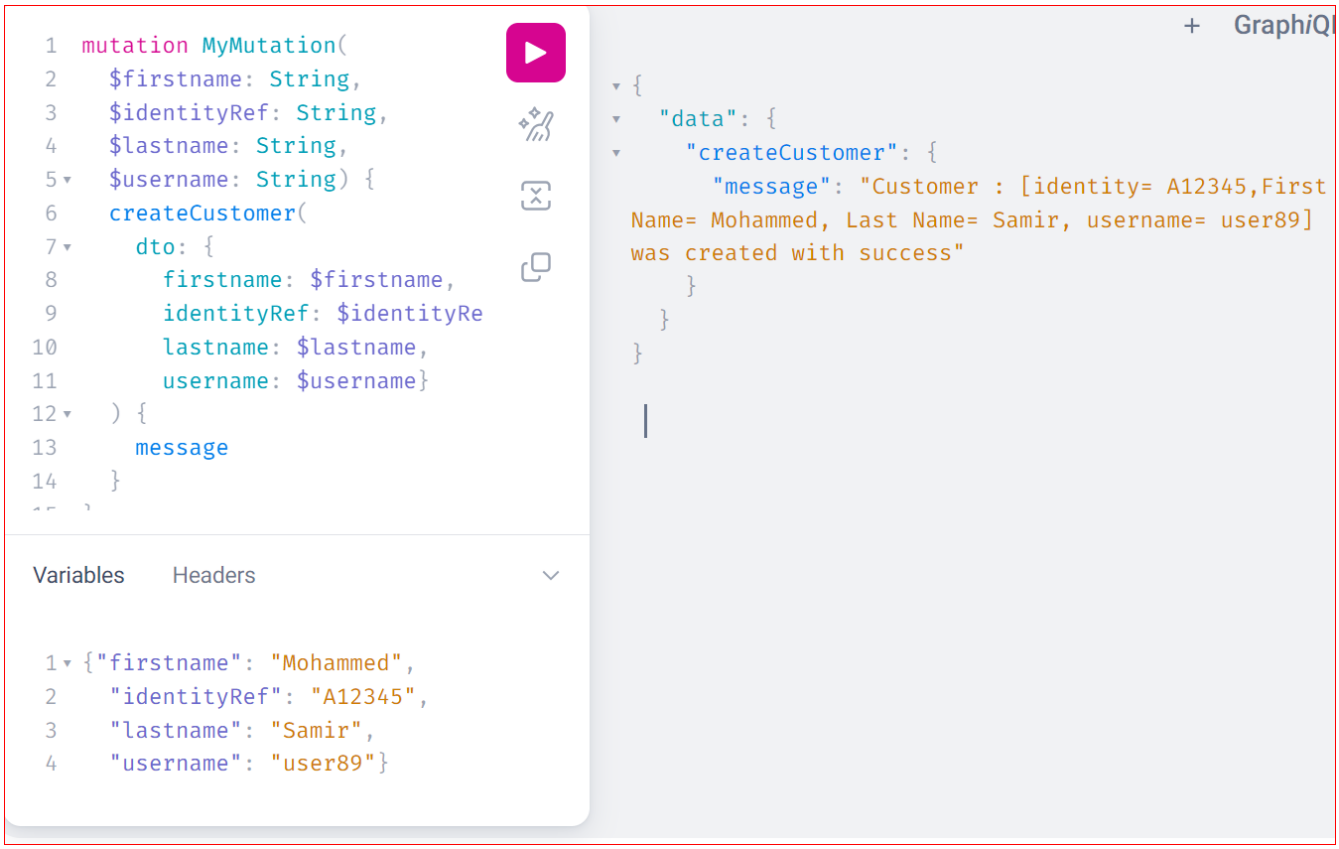
I. Tester la requête deleteCustomer

- Cliquer sur ***deleteCustomer***, entrer identityRef du client à supprimer et cliquer ensuite sur la flèche au centre pour exécuter la requête :

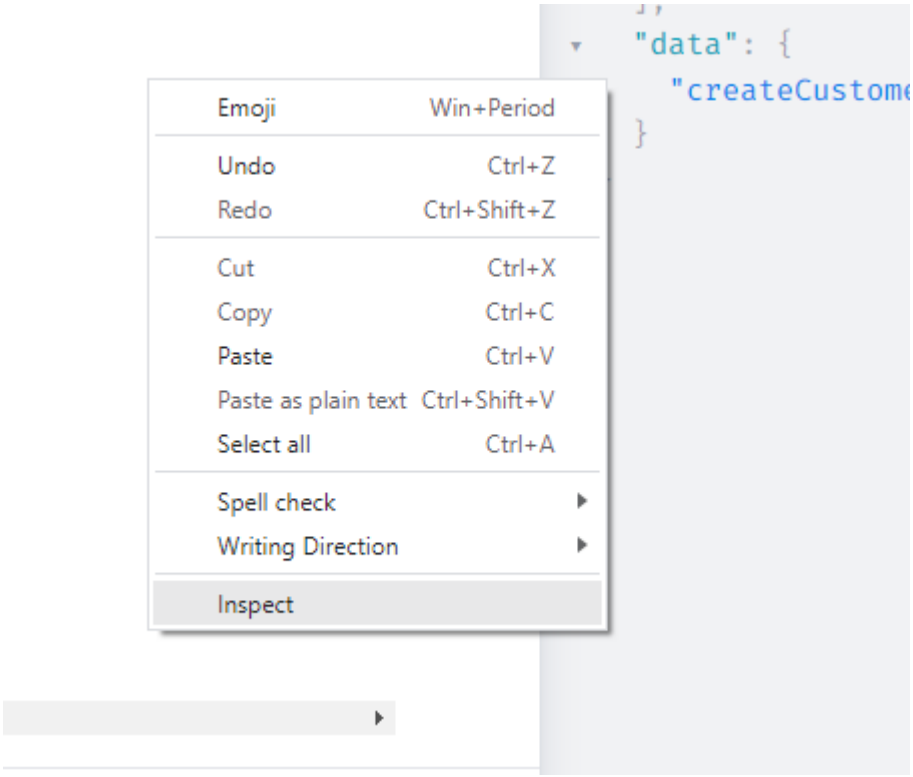


VI. Créer des requêtes paramétrées au niveau de GraphiQL

- Par exemple, pour le service createCustomer, créer la requête paramétrée ci-dessous :

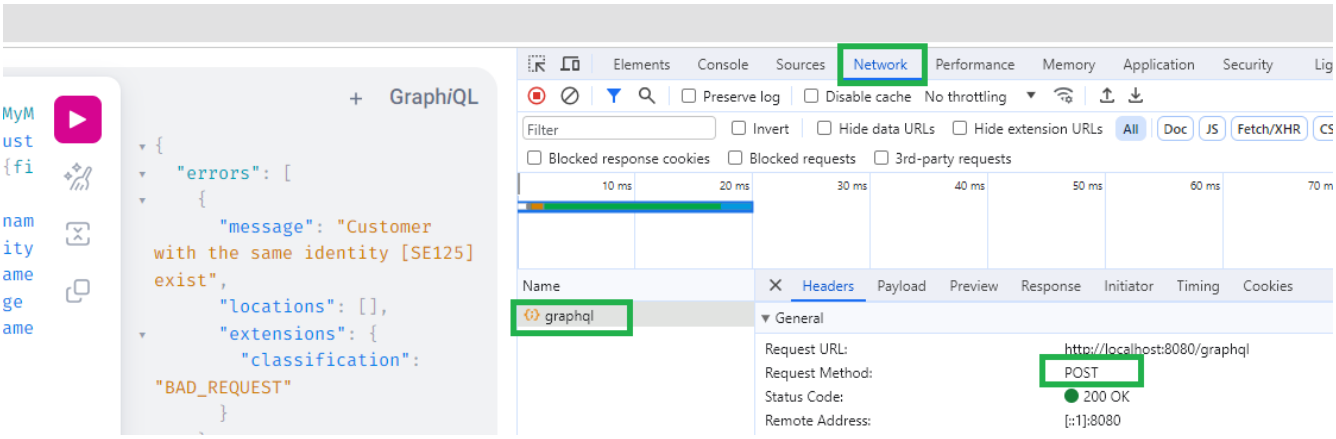


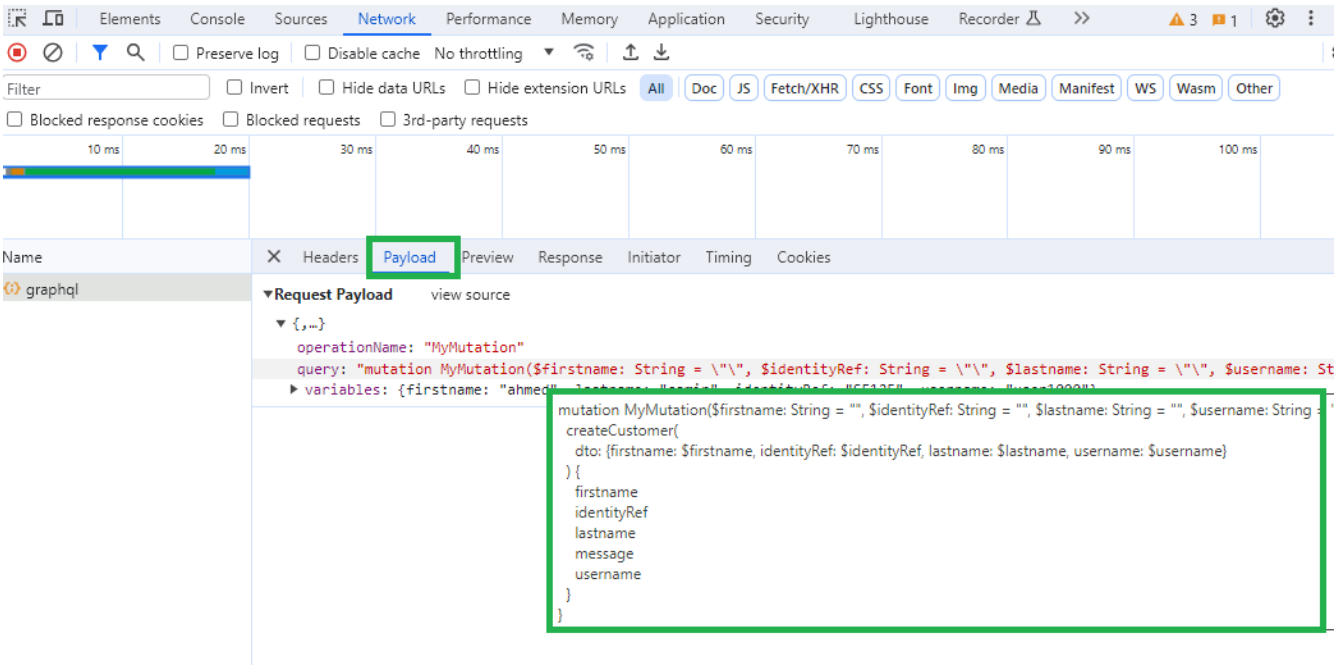
- Remarquer que les variables sont définies dans la rubrique Variables.
- Vous pouvez consulter la requête envoyée par GraphiQL en inspectant votre navigateur comme expliqué ci-dessous :
  - ✓ Dans la page de GraphiQL, cliquer à droite de votre souris :



✓ Cliquer sur Inspect.

✓ Au niveau de la page, cliquer sur *Network* et exécuter ensuite votre requête. Vous allez remarquer que la méthode http qui est envoyée est bien POST et que la requête GraphQL est mise dans le body comme montré ci-après :





VII. Gestion des exceptions avec GraphQL

- Si vous essayez d'ajouter un client dont le numéro de l'identité existe déjà ou bien le nom d'utilisateur est déjà utilisé, le serveur GraphQL enverra le message d'erreur suivant :


```
{
  "errors": [
    {
      "message": "INTERNAL_ERROR for 034ce8bb-36e6-8ad1-ea1f-2474107e5745",
      "locations": [
        {
          "line": 5,
          "column": 3
        }
      ],
      "path": [
        "createCustomer"
      ],
      "extensions": {
        "classification": "INTERNAL_ERROR"
      }
    }
  ],
  "data": {
    "createCustomer": null
  }
}
```

- Pour intercepter les exceptions GraphQL et personnaliser les messages à retourner aux clients, créer la classe ***GraphQlExceptionHandler*** suivante :

```java
package ma.formations.graphql.service.exception;

import graphql.GraphQLError;
import graphql.GraphqlErrorBuilder;
import graphql.schema.DataFetchingEnvironment;
import org.springframework.graphql.execution.DataFetcherExceptionResolverAdapter;
import org.springframework.graphql.execution.ErrorType;
import org.springframework.stereotype.Component;

@Component
public class GraphQlExceptionHandler extends DataFetcherExceptionResolverAdapter {

   @Override
   protected GraphQLError resolveToSingleError(Throwable ex, DataFetchingEnvironment env) {

      if (ex instanceof BusinessException) {
         return GraphqlErrorBuilder.newError()
               .errorType(ErrorType.BAD_REQUEST)
               .message(ex.getMessage())
               .build();
      } else {
         return GraphqlErrorBuilder.newError()
               .errorType(ErrorType.INTERNAL_ERROR)
               .message(String.format("Technical error !!. Contact your administrator : see details : [%s]", ex.getMessage()))
               .build();
      }
   }
}
```

Explications :

- ✓ Les exceptions GraphQL sont traitées par la méthode ***resolveToSingleError(Throwable ex, DataFetchingEnvironment env)*** de la classe ***DataFetcherExceptionResolverAdapter.*** C'est pourquoi, pour intercepter ces exceptions, il suffit d'étendre cette classe et redéfinir la méthode ***resolveToSingleError.***
- ✓ Notre méthode redéfinie ***resolveToSingleError*** traite deux types d'exceptions :
  - L'exception de type *BusinessException*
  - Et les autres exceptions.

- Lancer votre application et refaire le même test. Remarquer que votre message en cas d'exception de type *BusinessException* a été bien affichée :

Conclusion

Le code source de cet atelier est disponible sur GITHUB :

https://github.com/abbouformations/bank-service-graphql.git