Département d'Informatique

Filiere : IAAD

A ,U :2023-2024

Université Moulay Ismail faculté des Sciences Meknès

Département d'Informatique

TP N 6 : Architectures Micro-services

Module : Systèmes Distribués

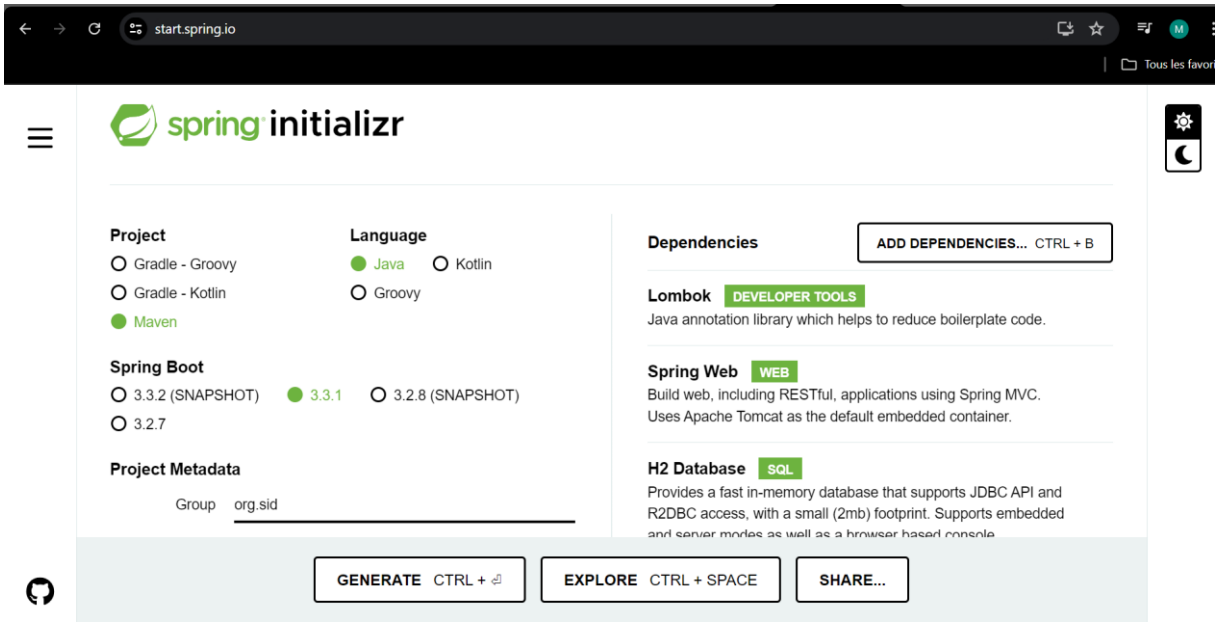
Réalisée par :

- ***Illa Meryeme***

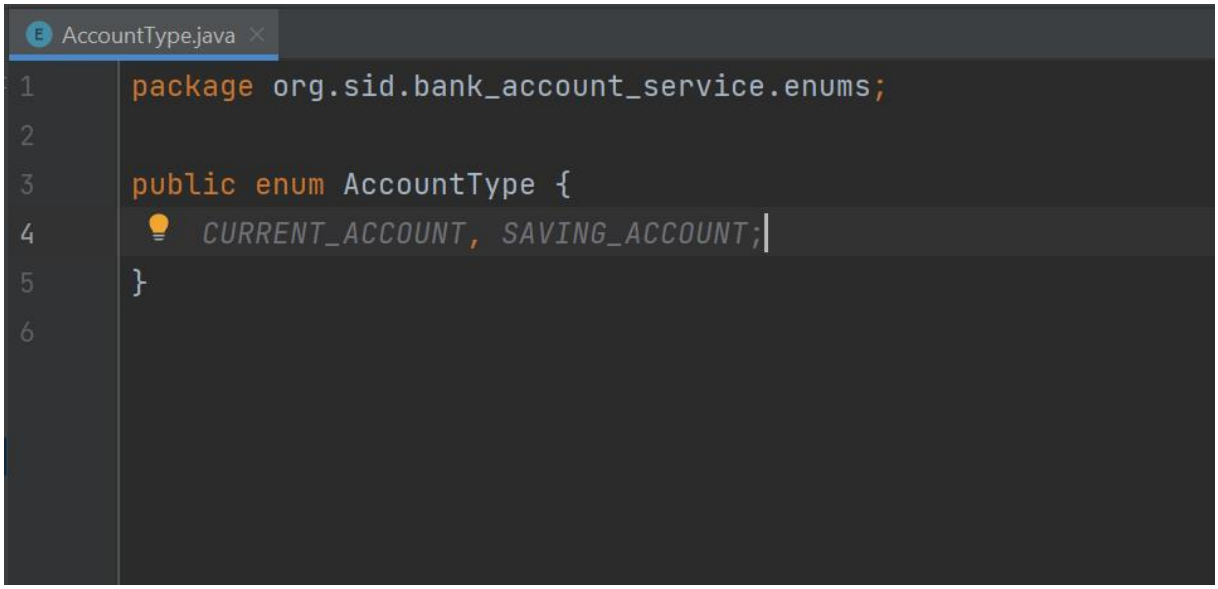
Première Partie : Développer un micro-service :

Partie 1 :

On va créer un projet spring dans Spring Initializr et ajouter les dépendances nécessaires puis générer le projet



On crée un package entities contient la classe java BankAccount et autre package enums contient l'enum AcountType



BankAccount.java

```java
package org.sid.bank_account_service.entities;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.sid.bank_account_service.enums.AccountType;

import javax.persistence.Entity;
import javax.persistence.Id;
import java.util.Date;
@Entity
@Data @NoArgsConstructor @AllArgsConstructor @Builder
public class BankAccount {
    @Id
    private String id;
    private Date createdAt;
    private double balance;
    private String currency;
    private AccountType type;
}
```

Endpoints Spring

On va créer une interface BankAccountRepository

BankAccountRepository.java

```java
package org.sid.bank_account_service.repositories;

import org.sid.bank_account_service.entities.BankAccount;
import org.springframework.data.jpa.repository.JpaRepository;

public interface BankAccountRepository extends JpaRepository<BankAccount, String> {
}
```

On va faire un premier test donc dans l'application c'est les premiers discours fait d'habitude on va les faire ici donc pour cela on va insérer quelques comptes dans la base de données donc on va utiliser voila on va créer un objet dans l'application
On va créer trois façons pour créer un compte banque

BankAccountServiceApplication.java

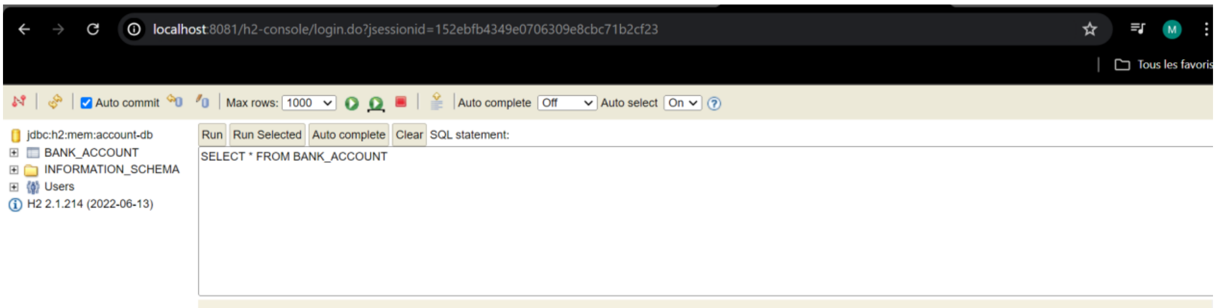
```java
public static void main(String[] args) { SpringApplication.run(BankAccountServiceApp

@Bean
CommandLineRunner start(BankAccountRepository bankAccountRepository) {
    return args -> {
        for (int i=0; i<10; i++) {
            BankAccount bankAccount = BankAccount.builder()
                    .id(UUID.randomUUID().toString())
                    .type(Math.random()>0.5? AccountType.CURRENT_ACCOUNT: AccountType
                    .balance(10000+Math.random()*90000)
                    .createdAt(new Date())
                    .currency("MAD")
                    .build();
            bankAccountRepository.save(bankAccount);
        }

    };
}

}
```

Alors avant de faire le premier test, on va paramétrer donc ma base de données avec spring.datasource

application.properties

```
spring.application.name=bank-account-service
spring.datasource.url=jdbc:h2:mem:account-db
spring.h2.console.enabled=true
server.port=8081
```

On exécute l'application
Pour un microservice qui est démarré en 13 secondes c'est énorme, le bout d'une application spring ça peut se faire avec une seconde , 2 secondes n'est pas plus.
Il y a un moyen de créer ce qu'on appelle les applications natives s'appelle GraalVM



ID	BALANCE	CREATED_AT	CURRENCY	TYPE
5f042f6f-5c17-4334-96f3-9e350ba0f2a9	87867.65217231154	2024-07-05 00:54:54.12	MAD	1
651b4a86-6350-4415-880c-ea6fa2519bc7	17402.903803056117	2024-07-05 00:54:54.212	MAD	1
c738f800-ed8b-4373-b6ee-285e81804b92	81417.09278723247	2024-07-05 00:54:54.214	MAD	1
b06ced0a-4ff5-4e1f-8a08-22b632f28ac5	68872.47856856225	2024-07-05 00:54:54.215	MAD	0
37a37147-03a9-44f7-8f41-3a37b41a0c69	83932.46570400729	2024-07-05 00:54:54.216	MAD	0
3a44a4f1-ca56-4c0a-9713-6f339d45078c	66649.9494609258	2024-07-05 00:54:54.218	MAD	0
f0a7735e-d805-45d5-b4b7-39907b7b4723	36927.41764486482	2024-07-05 00:54:54.219	MAD	1
adb11054-0310-452e-b201-2e7c210c48e9	78885.73489352765	2024-07-05 00:54:54.22	MAD	0
3b5c9cd4-bc88-4e79-9d6b-ea883af18ce9	71869.59586760412	2024-07-05 00:54:54.222	MAD	0
e49cad69-3720-4d67-aa16-49f4eccccc3e5	97452.56671839181	2024-07-05 00:54:54.223	MAD	0

(10 rows, 40 ms)

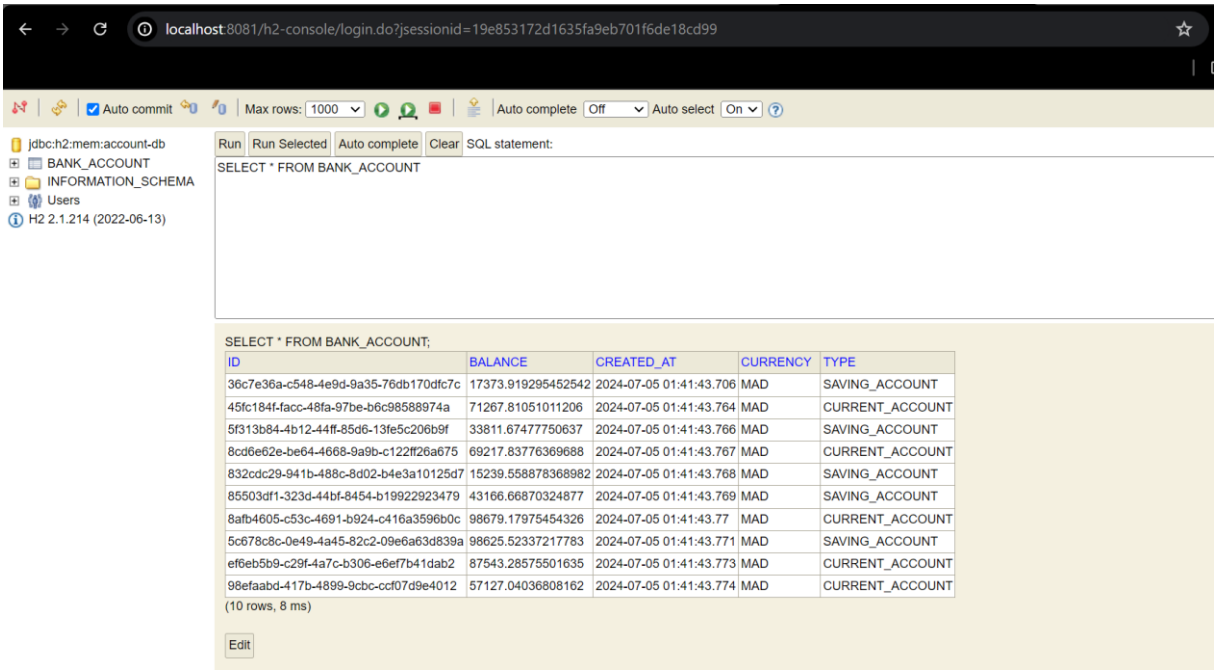
Edit

On va afficher le type de type String au lieu de ordinal (par défaut)

BankAccount.java

```java
import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.Id;
import java.util.Date;
@Entity
@Data @NoArgsConstructor @AllArgsConstructor @Builder
public class BankAccount {
    @Id
    private String id;
    private Date createdAt;
    private double balance;
    private String currency;
    @Enumerated(EnumType.STRING)
    private AccountType type;
}
```

On s’exécute

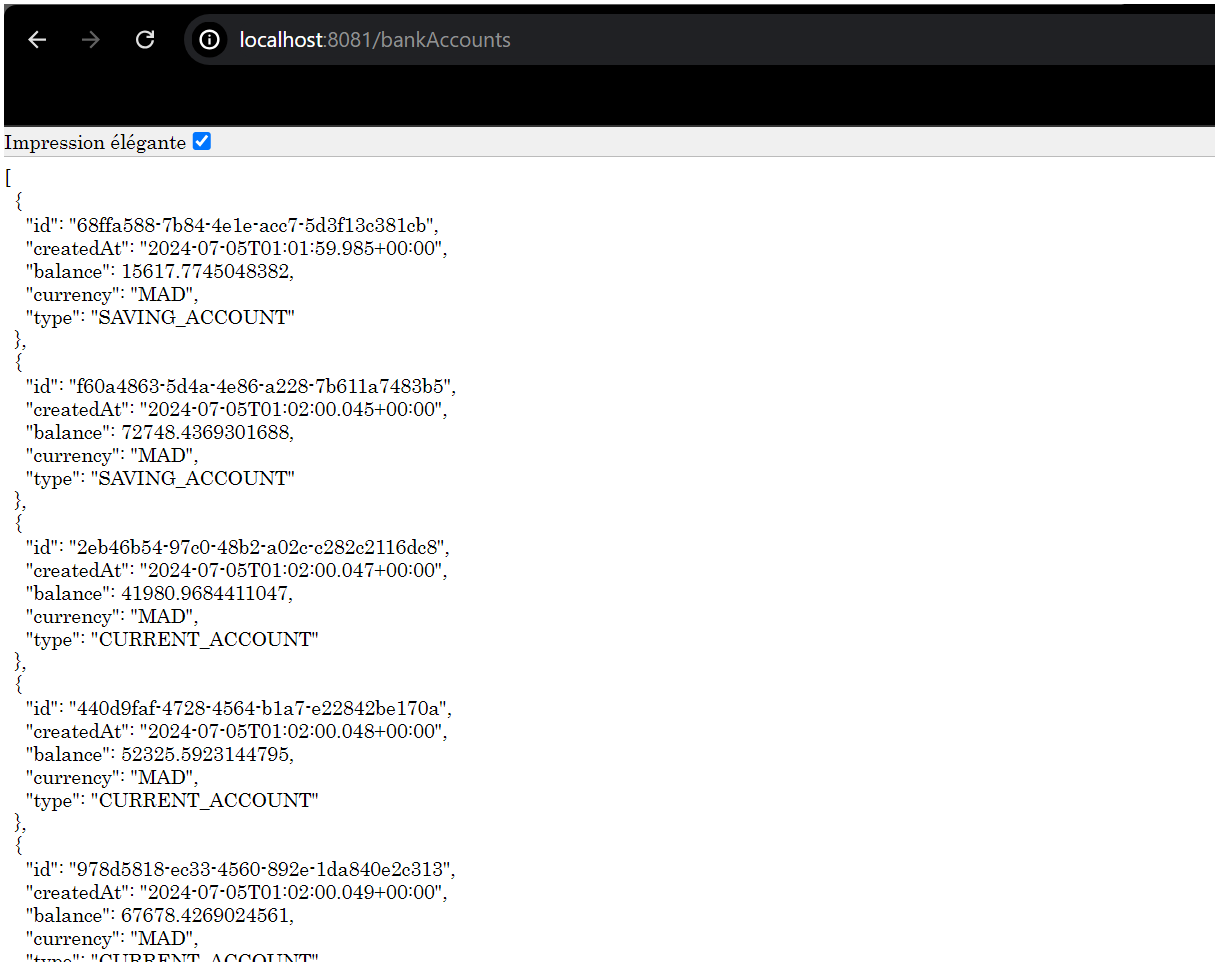


On va créer un package web dans lequel on va créer donc la classe AccountRestController alors les connecteurs c'est des contrôleurs et on va utiliser directement RestController donc on aura besoin d'injecter pour le moment, la couche repositories parce qu'on a besoin d'accéder à la base de données et pour faire l'injection des dépendances on utilise @Autowired ce qui est déprécie et pour faire une injection meilleure il faut faire l'injection via le constructor.
On va consulter la liste des comptes par la méthode bankAccounts() et on va accéder à cette méthode par @getMapping et aussi une méthode pour consulter un compte
On utilise la notation @PathVariable pour indiquer que c'est un paramétre qui va etre récupéré à partir de path

AccountRestController.java

```java
@RestController
public class AccountRestController {
    @Autowired
    private BankAccountRepository bankAccountRepository;

    public AccountRestController(BankAccountRepository bankAccountRepository){
        this.bankAccountRepository= bankAccountRepository;

    }
    @GetMapping("/bankAccounts")
    public List<BankAccount> bankAccounts() {
        return bankAccountRepository.findAll();
    }

    @GetMapping("/bankAccount/{id}")
    public BankAccount bankAccount(@PathVariable String id) {
        return bankAccountRepository.findById(id)
                .orElseThrow(() ->new RuntimeException(String.format("Account %s not fou
    }
```

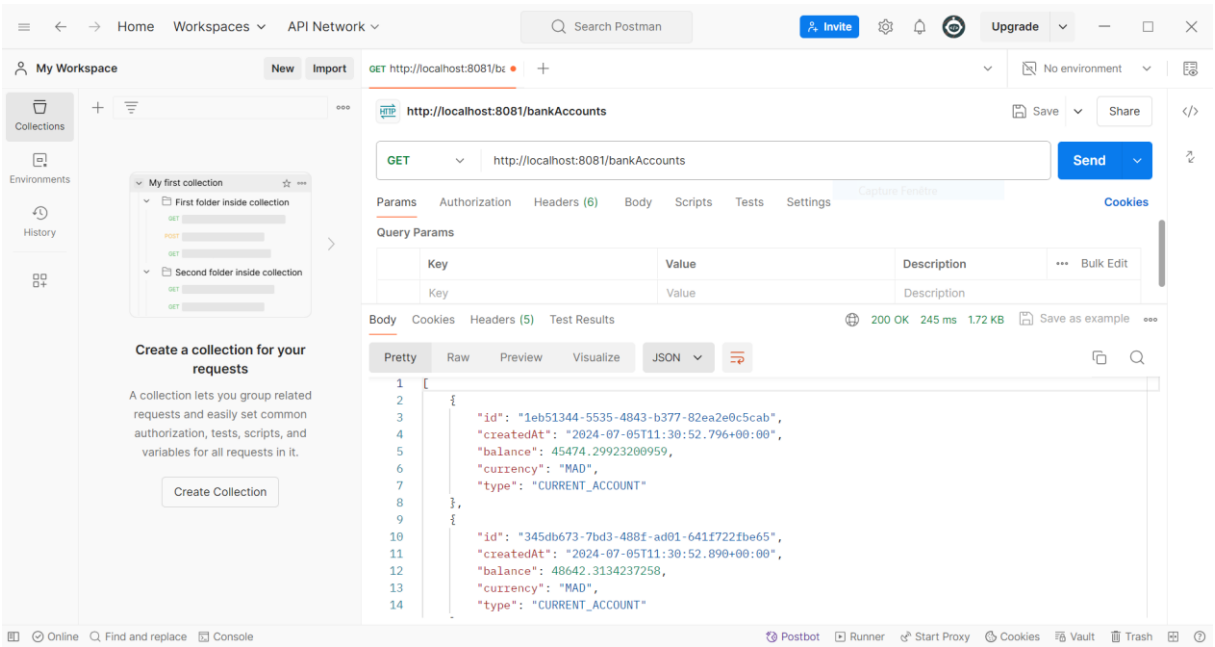
On teste l'application

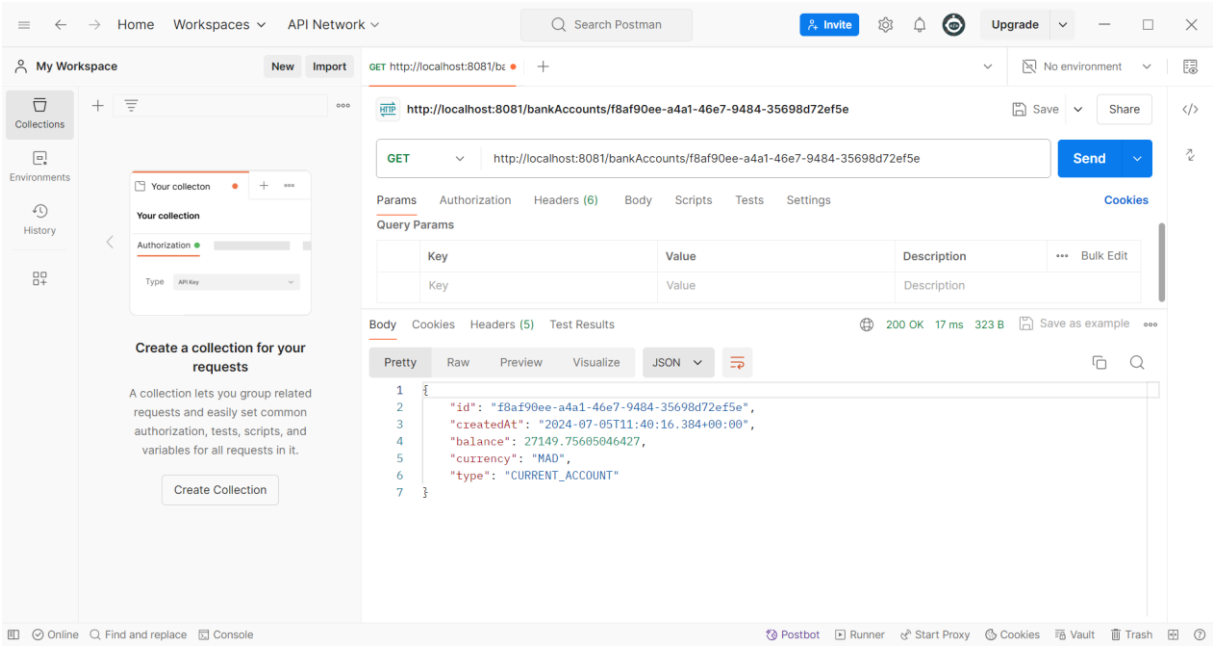


On va créer une méthode qui permet de retourner un objet et rappeler save et on utilise @RequestBody pour indiquer que les données de la requête viennent de les données de backend à quand il va les récupérer dans le corps de la requête et pour faire Update on utilise une requête avec put (@PutMapping) puis delete

```java
@PostMapping("/bankAccounts")
public BankAccount save(@RequestBody BankAccount bankAccount) {
    if(bankAccount.getId()==null) bankAccount.setId(UUID.randomUUID().toString());
    return bankAccountRepository.save(bankAccount);
}
@PutMapping("/bankAccounts/{id}")
public BankAccount update(@PathVariable String id, @RequestBody BankAccount bankAccou
    BankAccount account=bankAccountRepository.findById(id).orElseThrow();
    if(bankAccount.getBalance() != null) account.setBalance(bankAccount.getBalance())
    if(bankAccount.getCreatedAt() != null) account.setCreatedAt(new Date());
    if(bankAccount.getType() != null) account.setType(bankAccount.getType());
    if(bankAccount.getCurrency() != null) account.setCurrency(bankAccount.getCurrency
    return bankAccountRepository.save(account);
}
@DeleteMapping("/bankAccounts/{id}")
public void deleteAccount(@PathVariable String id) {
    bankAccountRepository.deleteById(id);
}
```

On va tester, et pour tester les méthodes post on va utiliser l'outil de postman

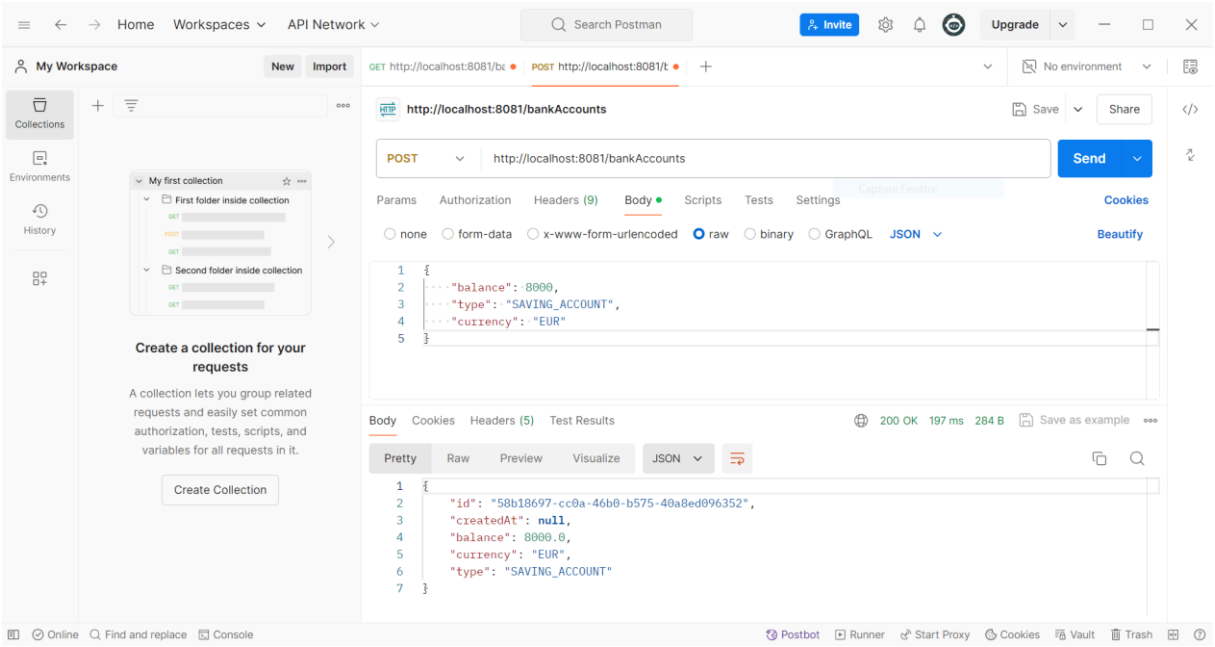




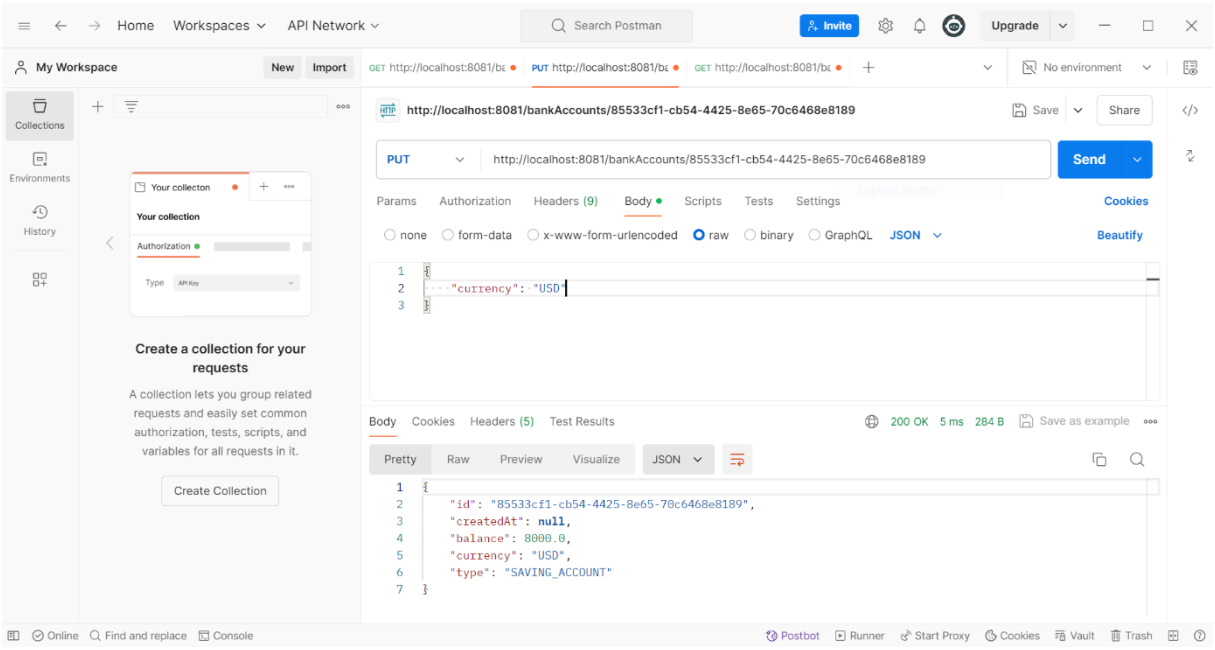
Dans la méthode save, on suppose qu'on reçoit l'id , si on envoie un compte sans id on le génère automatique

```java
@PostMapping("/bankAccounts")
public BankAccount save(@RequestBody BankAccount bankAccount) {
    if(bankAccount.getId()==null) bankAccount.setId(UUID.randomUUID().toString());
    return bankAccountRepository.save(bankAccount);
}
```

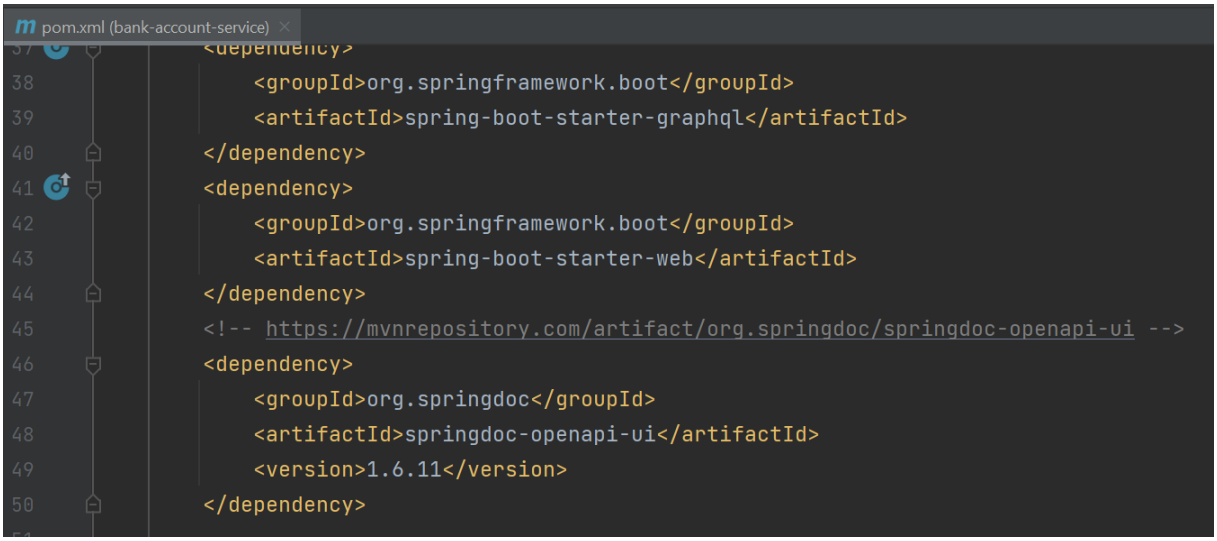
On teste



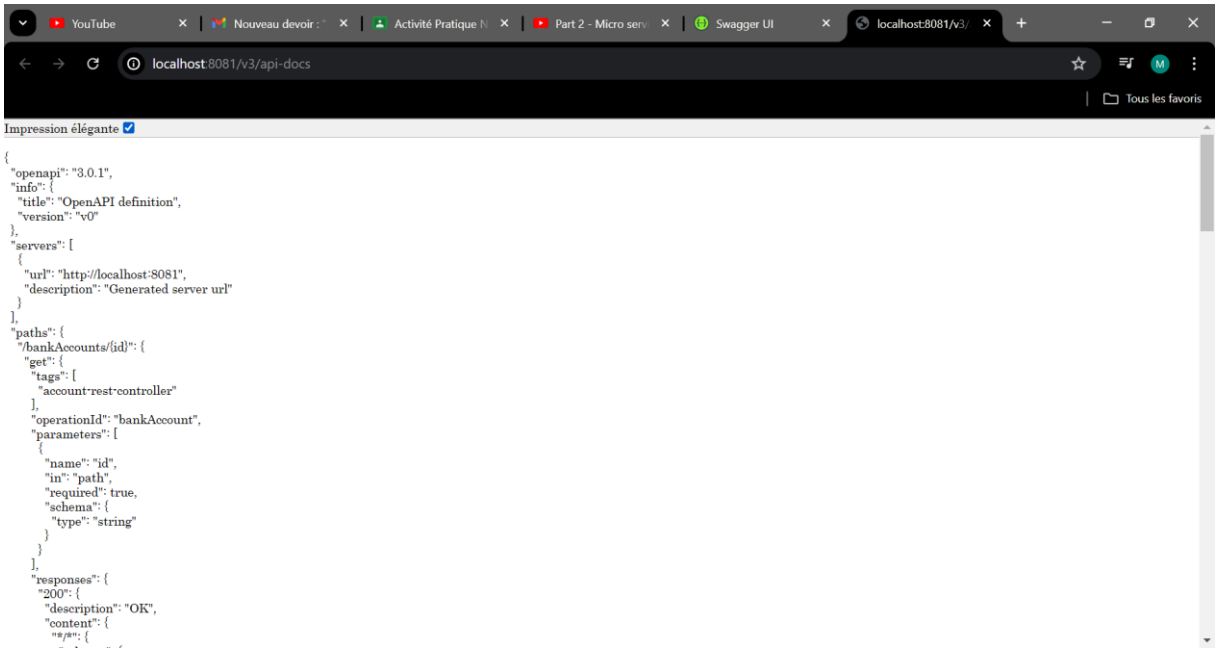
Si on fait put par exemple et on veut mettre à jour un compte et on veut mettre à jour currency



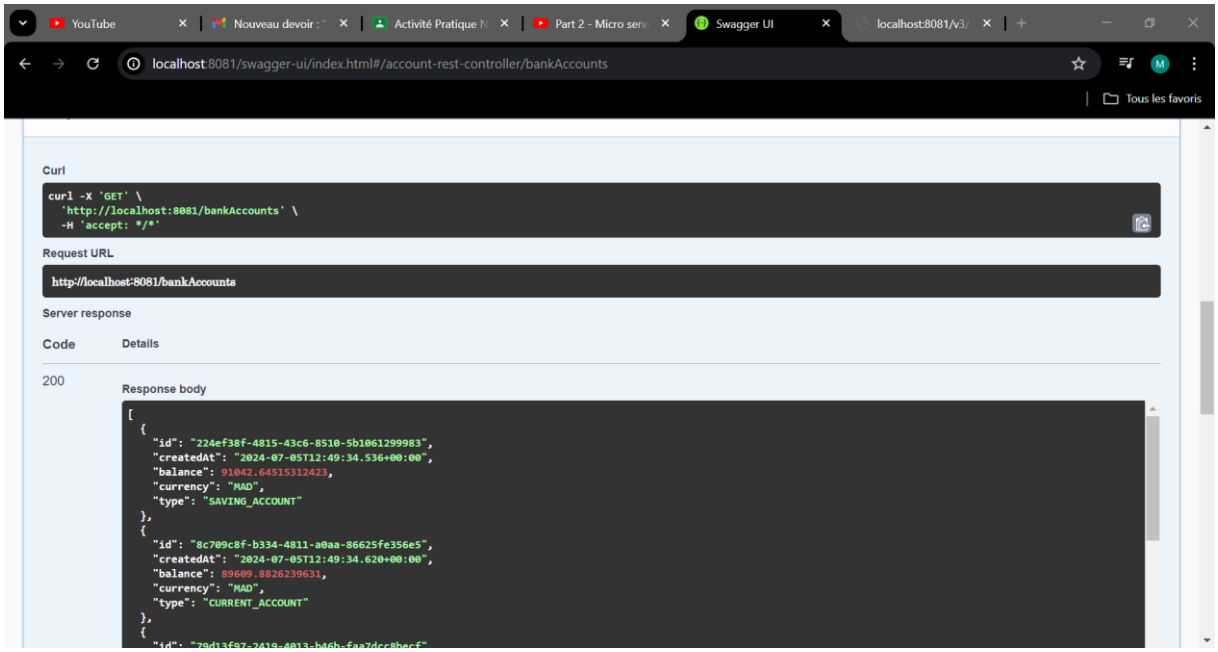
Après ça le service peut communiquer avec le monde extérieur via RestAPI et on va utiliser la documentation Swagger et on ajoute la dépendance



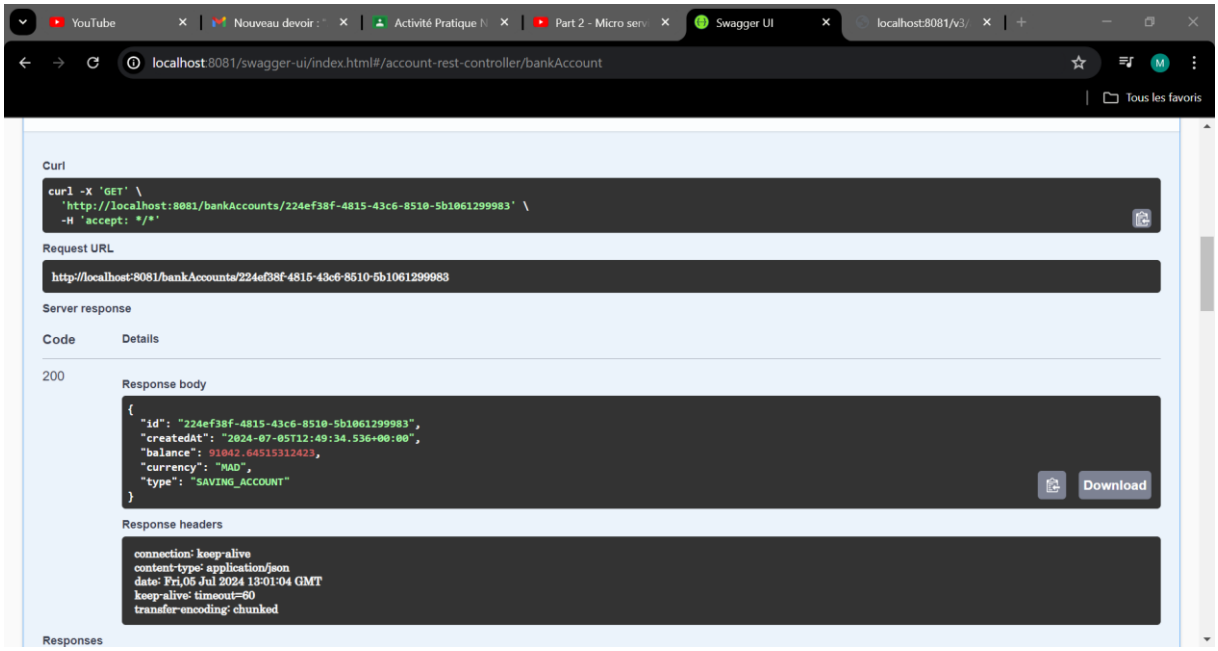
Et la documentation open ai



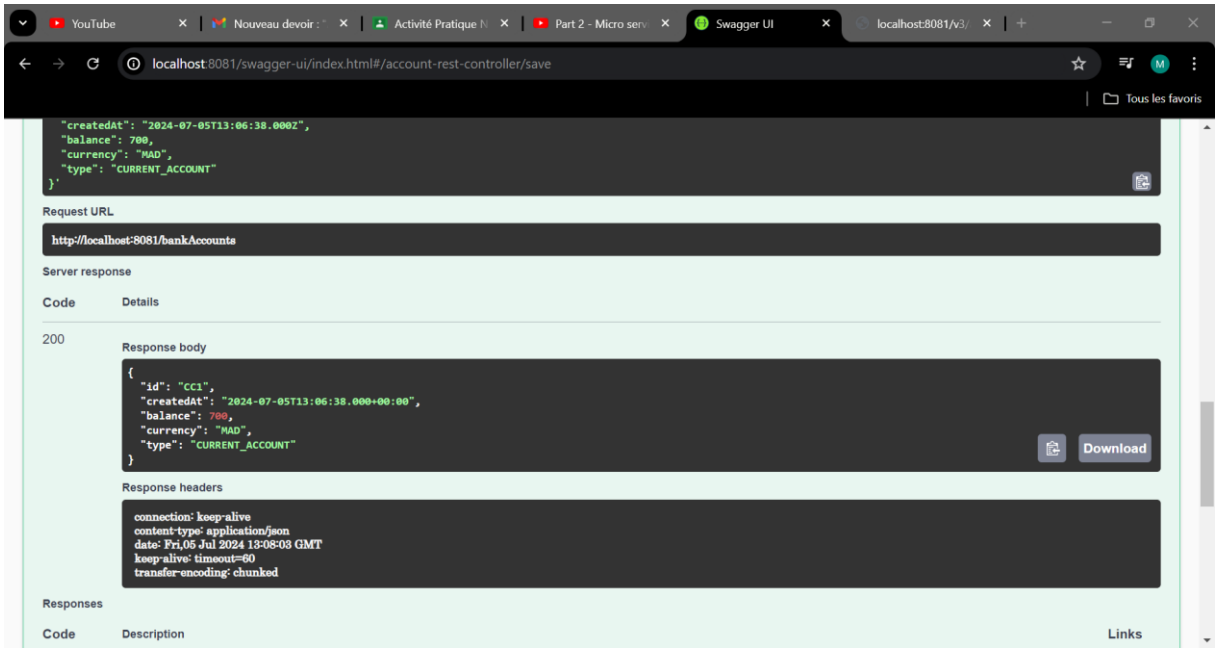
On fait un test dans swagger (l'utilisateur n'a pas besoin de postman), pour GET bankAccounts



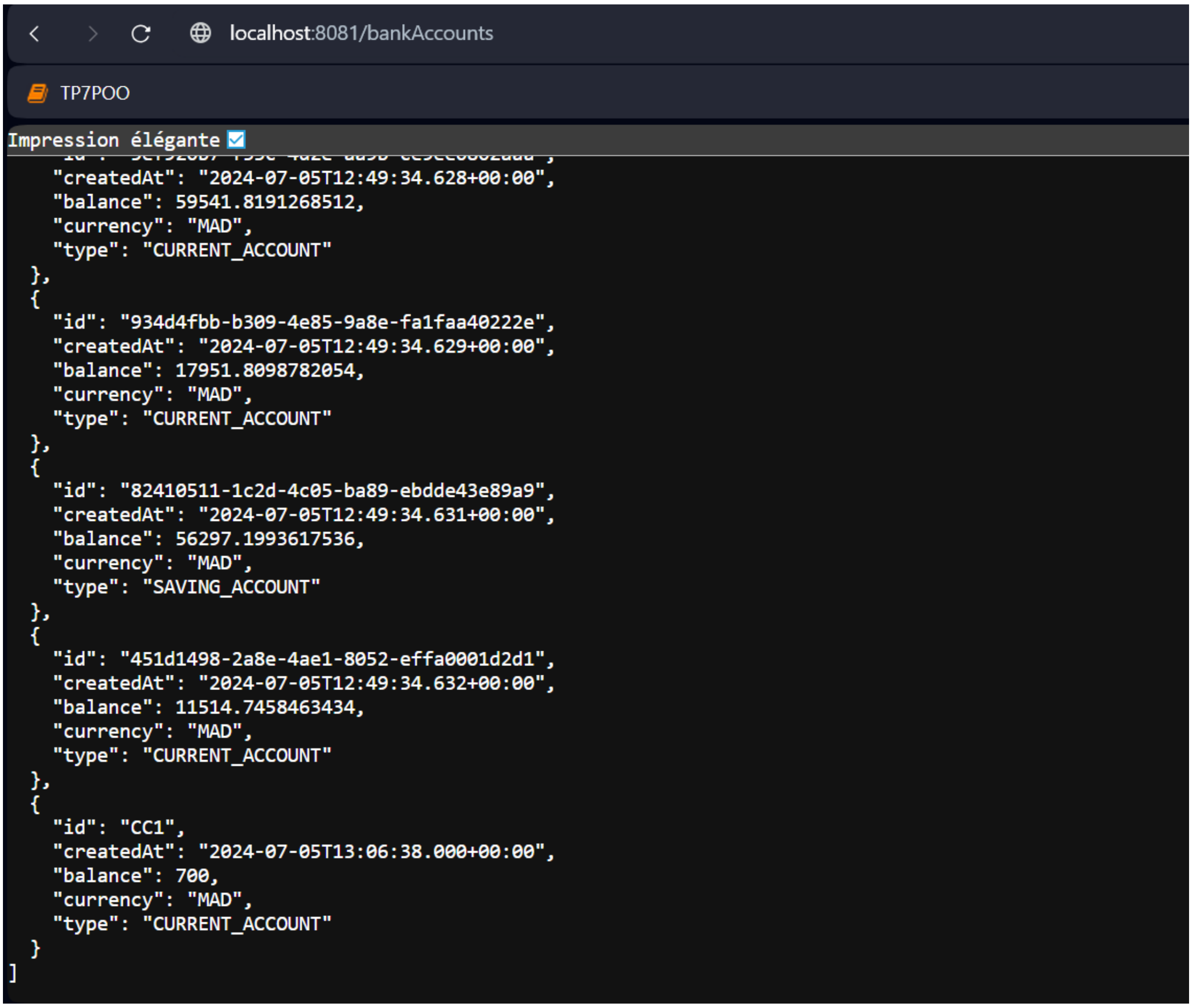
Pour Consulter un compte (par exemple 224ef38f-4815-43c6-8510-5b1061299983)



Si on veut ajouter un compte on fait la méthode post

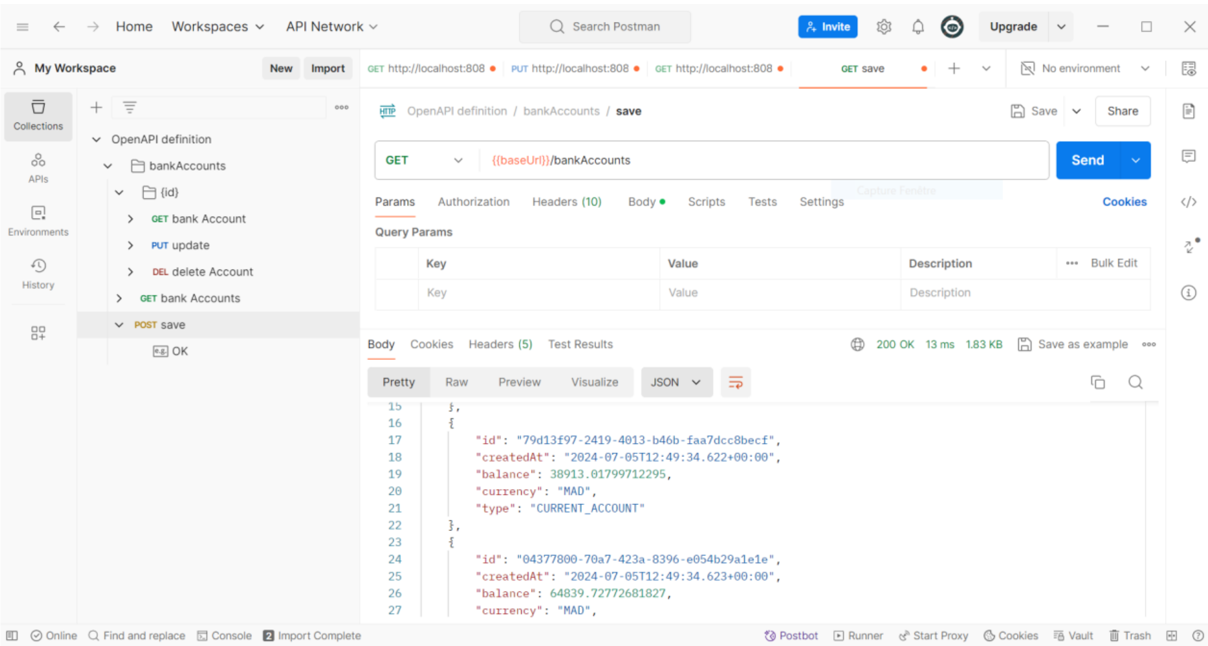


On vérifie




```
    "createdAt": "2024-07-05T12:49:34.628+00:00",
    "balance": 59541.8191268512,
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT"
  },
  {
    "id": "934d4fbb-b309-4e85-9a8e-fa1faa40222e",
    "createdAt": "2024-07-05T12:49:34.629+00:00",
    "balance": 17951.8098782054,
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT"
  },
  {
    "id": "82410511-1c2d-4c05-ba89-ebdde43e89a9",
    "createdAt": "2024-07-05T12:49:34.631+00:00",
    "balance": 56297.1993617536,
    "currency": "MAD",
    "type": "SAVING_ACCOUNT"
  },
  {
    "id": "451d1498-2a8e-4ae1-8052-effa0001d2d1",
    "createdAt": "2024-07-05T12:49:34.632+00:00",
    "balance": 11514.7458463434,
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT"
  },
  {
    "id": "CC1",
    "createdAt": "2024-07-05T13:06:38.000+00:00",
    "balance": 700,
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT"
  }
]
```

La documentation donne l'interface de l'API c'est pratique de l'utiliser pour tester les API ou on peut utiliser la documentation sous postman



On ajouter la dépendance rest, spring va créer un web service généré

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

On ajoute a l'interface BankAccountRepository l'annotation @RepositoryRestResource pour demander à spring au démarrage de démarrer un web service Rest Full qui permet de gérer les entités de BankAccount

BankAccountRepository.java

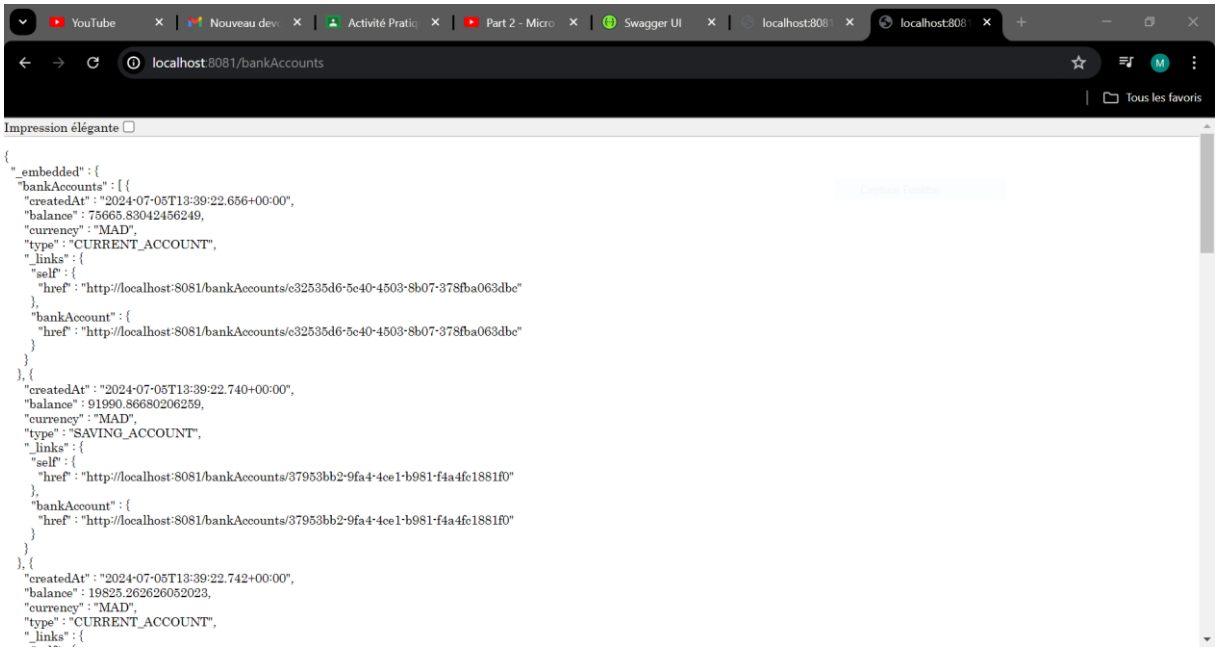
```java
package org.sid.bank_account_service.repositories;

import org.sid.bank_account_service.entities.BankAccount;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource
public interface BankAccountRepository extends JpaRepository<BankAccount, String> {
}
```

Et on ajoute @RequetMapping(« /api ») c'est-à-dire pour accéder à ce web service il faut écrire /api/bankAccounts

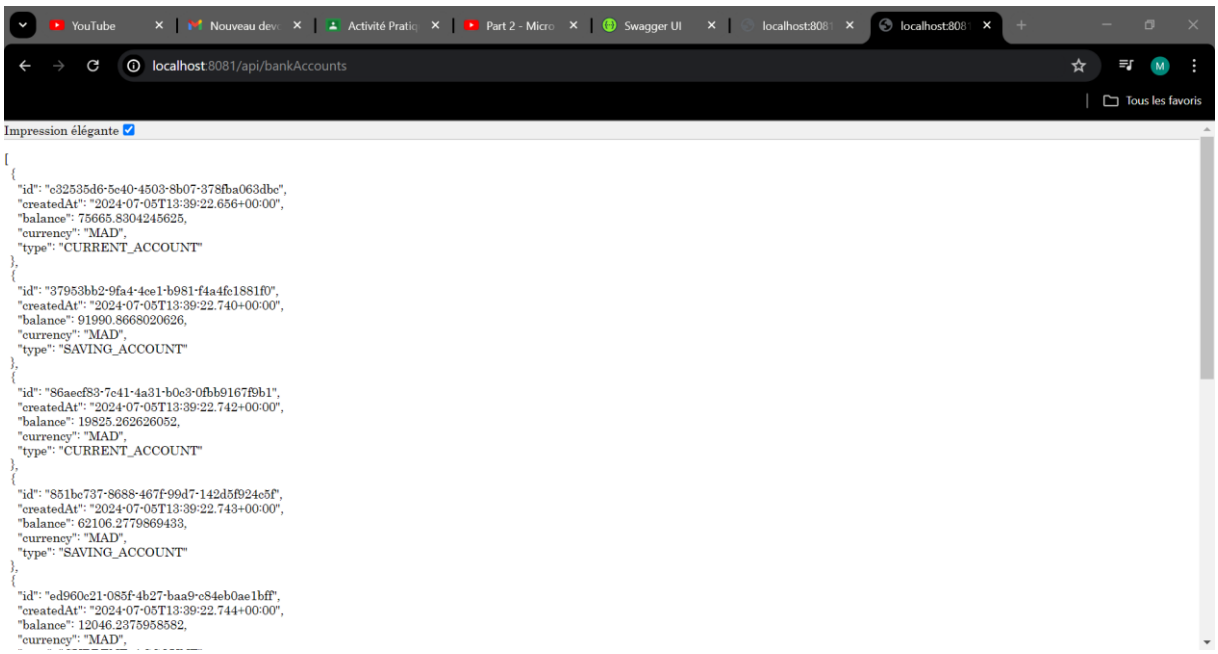
AccountRestController.java

```java
import java.util.List;
import java.util.UUID;

@RestController
@RequestMapping("/api")
public class AccountRestController {
    @Autowired
    private BankAccountRepository bankAccountRepository;

    public AccountRestController(BankAccountRepository bankAccountRepository){
        this.bankAccountRepository= bankAccountRepository;

    }
    @GetMapping("/bankAccounts")
    public List<BankAccount> bankAccounts() { return bankAccountRepository.findAll();

    @GetMapping("/bankAccounts/{id}")
    public BankAccount bankAccount(@PathVariable String id) {
        return bankAccountRepository.findById(id)
```

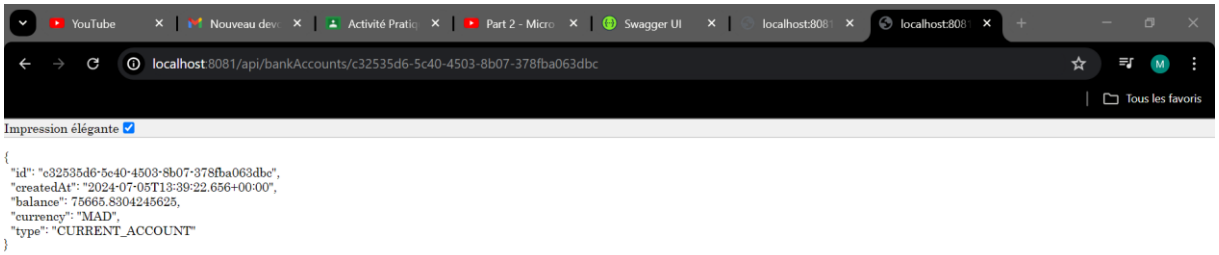
On exécute




```
{
  "_embedded" : {
    "bankAccounts" : [ {
      "createdAt" : "2024-07-05T13:39:22.656+00:00",
      "balance" : 75665.8304256249,
      "currency" : "MAD",
      "type" : "CURRENT_ACCOUNT",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8081/bankAccounts/c32535d6-5c40-4503-8b07-378fba063dbc"
        },
        "bankAccount" : {
          "href" : "http://localhost:8081/bankAccounts/c32535d6-5c40-4503-8b07-378fba063dbc"
        }
      }
    }, {
      "createdAt" : "2024-07-05T13:39:22.740+00:00",
      "balance" : 91990.8668206259,
      "currency" : "MAD",
      "type" : "SAVING_ACCOUNT",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8081/bankAccounts/37953bb2-9fa4-4ce1-b981-f4a4fc1881f0"
        },
        "bankAccount" : {
          "href" : "http://localhost:8081/bankAccounts/37953bb2-9fa4-4ce1-b981-f4a4fc1881f0"
        }
      }
    }, {
      "createdAt" : "2024-07-05T13:39:22.742+00:00",
      "balance" : 19825.262626052023,
      "currency" : "MAD",
      "type" : "CURRENT_ACCOUNT",
      "_links" : {
```

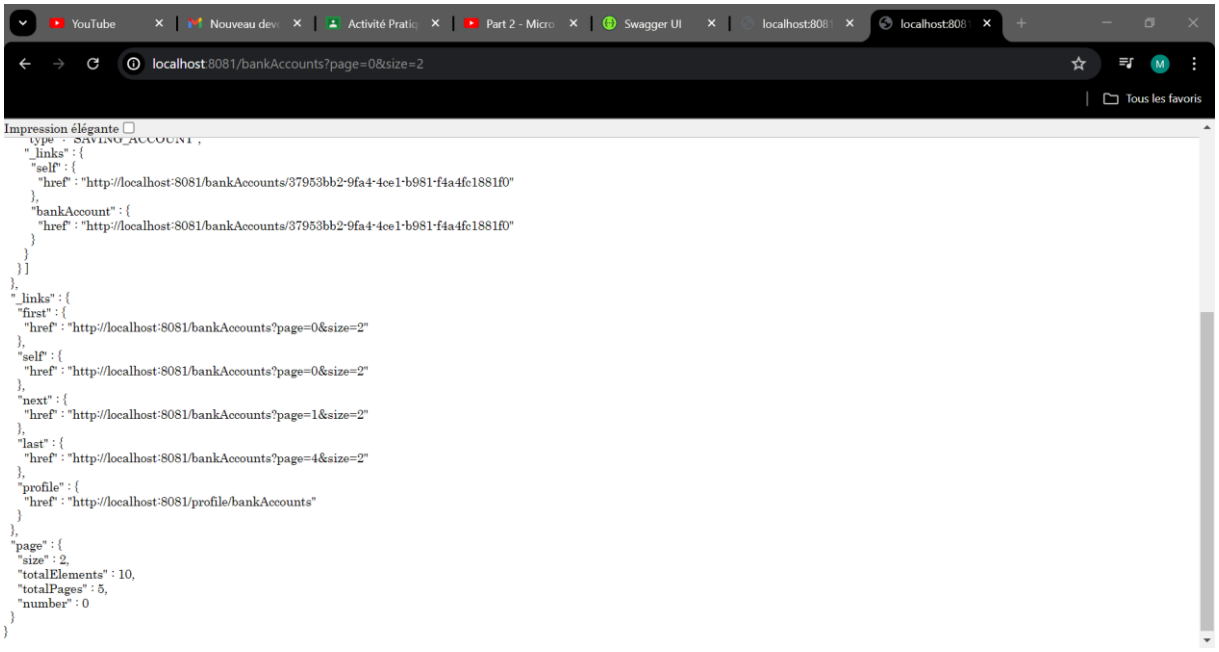




```
[
  {
    "id": "c32535d6-5c40-4503-8b07-378fba063dbc",
    "createdAt": "2024-07-05T13:39:22.656+00:00",
    "balance": 75665.8304245625,
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT"
  },
  {
    "id": "37953bb2-9fa4-4ce1-b981-f4a4fc1881f0",
    "createdAt": "2024-07-05T13:39:22.740+00:00",
    "balance": 91990.8668020626,
    "currency": "MAD",
    "type": "SAVING_ACCOUNT"
  },
  {
    "id": "86aecf83-7c41-4a31-b0c3-0fbb9167f9b1",
    "createdAt": "2024-07-05T13:39:22.742+00:00",
    "balance": 19825.262626052,
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT"
  },
  {
    "id": "851bc737-8688-467f-99d7-142d5f924c5f",
    "createdAt": "2024-07-05T13:39:22.743+00:00",
    "balance": 62106.2779869433,
    "currency": "MAD",
    "type": "SAVING_ACCOUNT"
  },
  {
    "id": "ed960c21-085f-4b27-baa9-c84eb0ae1bff",
    "createdAt": "2024-07-05T13:39:22.744+00:00",
    "balance": 12046.2375958582,
    "currency": "MAD",
```




```
{
  "id": "c32535d6-5c40-4503-8b07-378fba063dbc",
  "createdAt": "2024-07-05T13:39:22.656+00:00",
  "balance": 75665.8304245625,
  "currency": "MAD",
  "type": "CURRENT_ACCOUNT"
}
```


Il fait par défaut la pagination



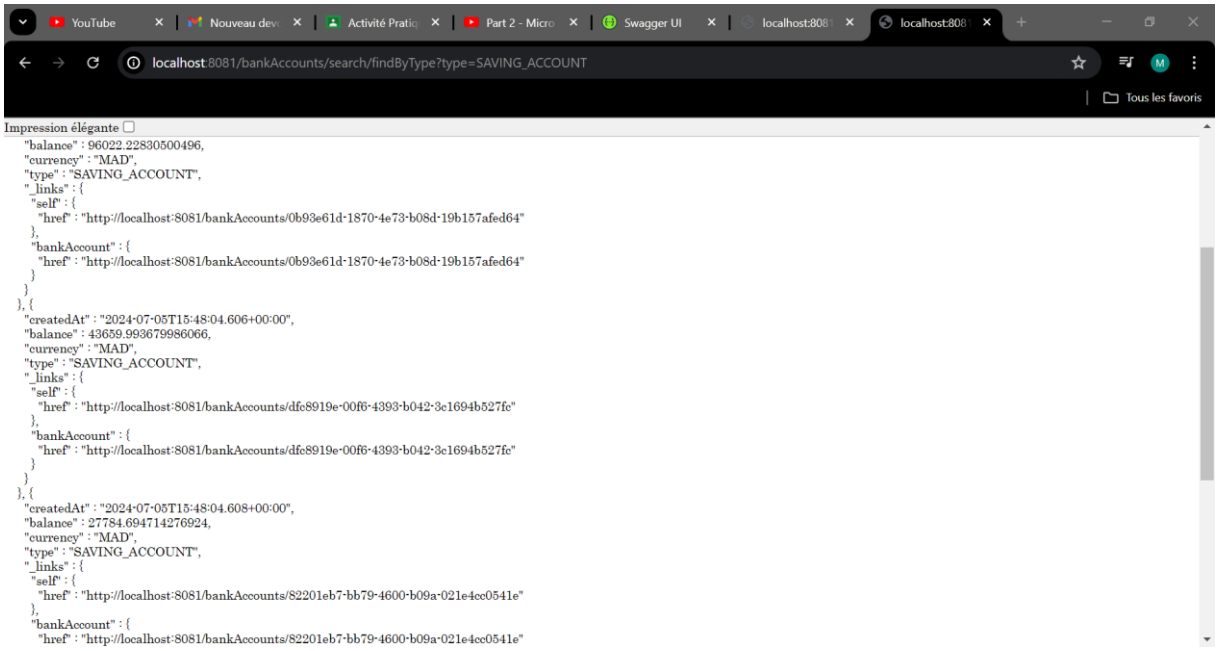

```
    "type" : "SAVING_ACCOUNT",
    "_links" : {
     "self" : {
      "href" : "http://localhost:8081/bankAccounts/37953bb2-9fa4-4ce1-b981-f4a4fc1881f0"
     },
     "bankAccount" : {
      "href" : "http://localhost:8081/bankAccounts/37953bb2-9fa4-4ce1-b981-f4a4fc1881f0"
     }
    }
   } ]
 },
 "_links" : {
  "first" : {
   "href" : "http://localhost:8081/bankAccounts?page=0&size=2"
  },
  "self" : {
   "href" : "http://localhost:8081/bankAccounts?page=0&size=2"
  },
  "next" : {
   "href" : "http://localhost:8081/bankAccounts?page=1&size=2"
  },
  "last" : {
   "href" : "http://localhost:8081/bankAccounts?page=4&size=2"
  },
  "profile" : {
   "href" : "http://localhost:8081/profile/bankAccounts"
  }
 },
 "page" : {
  "size" : 2,
  "totalElements" : 10,
  "totalPages" : 5,
  "number" : 0
 }
}
```

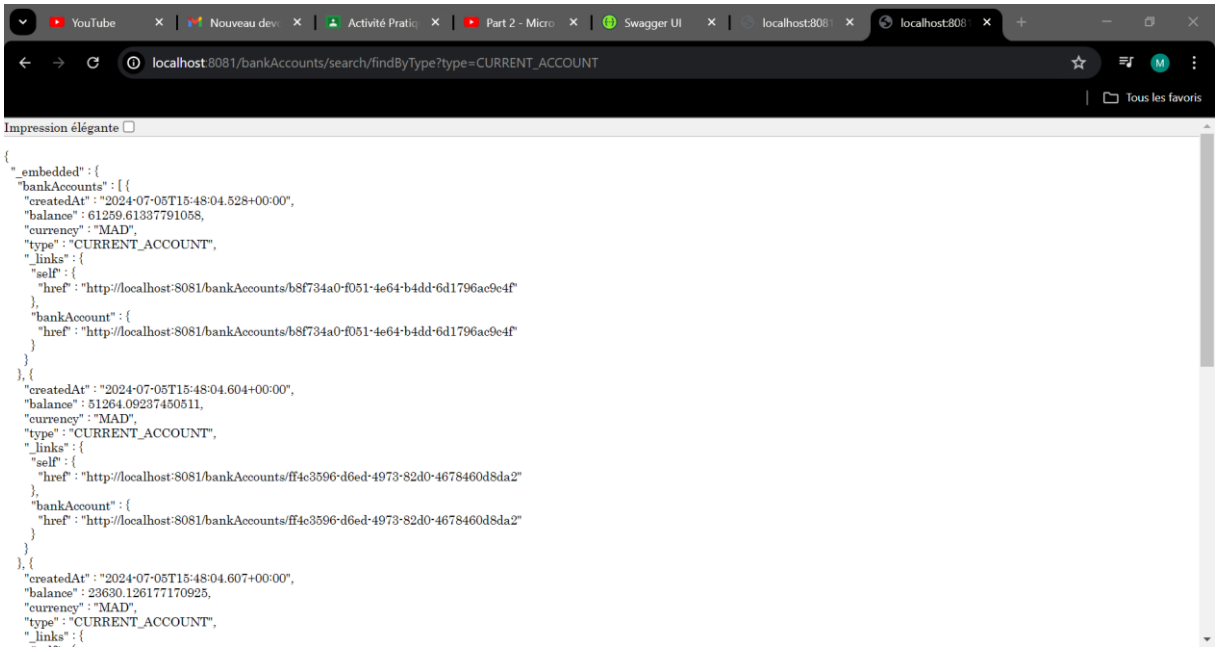

Si on fait appel à une méthode dans l'interface repository

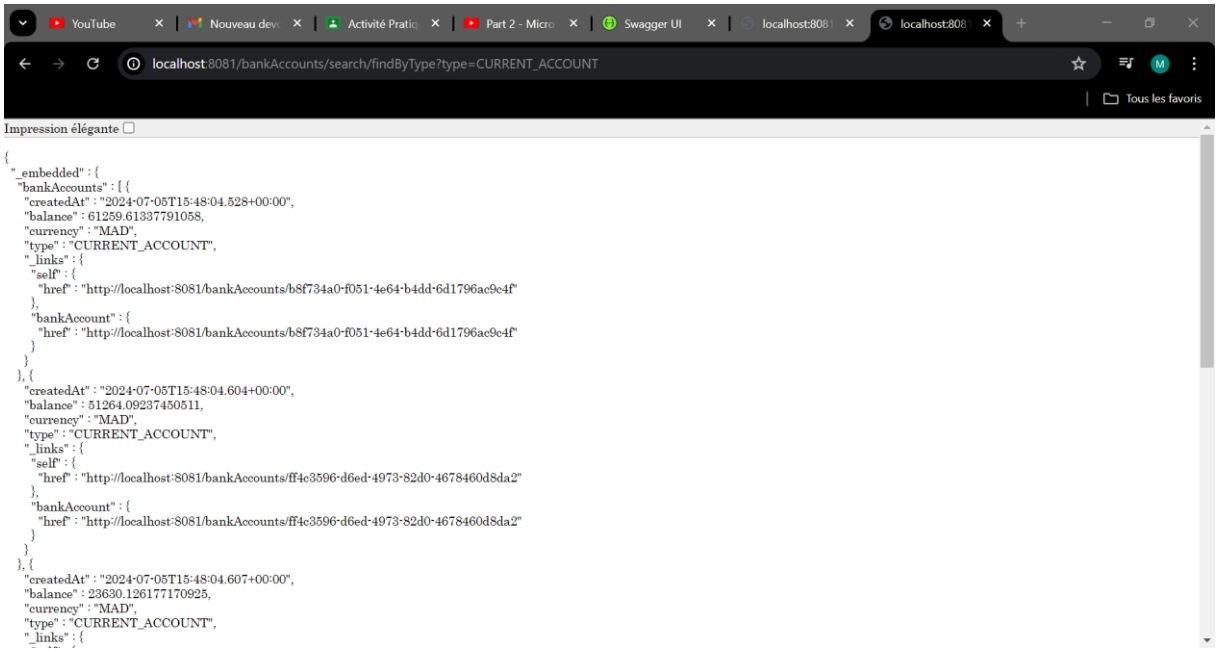
BankAccountRepository.java

```java
package org.sid.bank_account_service.repositories;

import org.sid.bank_account_service.entities.BankAccount;
import org.sid.bank_account_service.enums.AccountType;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

import java.util.List;

@RepositoryRestResource
public interface BankAccountRepository extends JpaRepository<BankAccount, String> {

    List<BankAccount> findByType(AccountType type) ;
}
```

On accède directement à la méthode





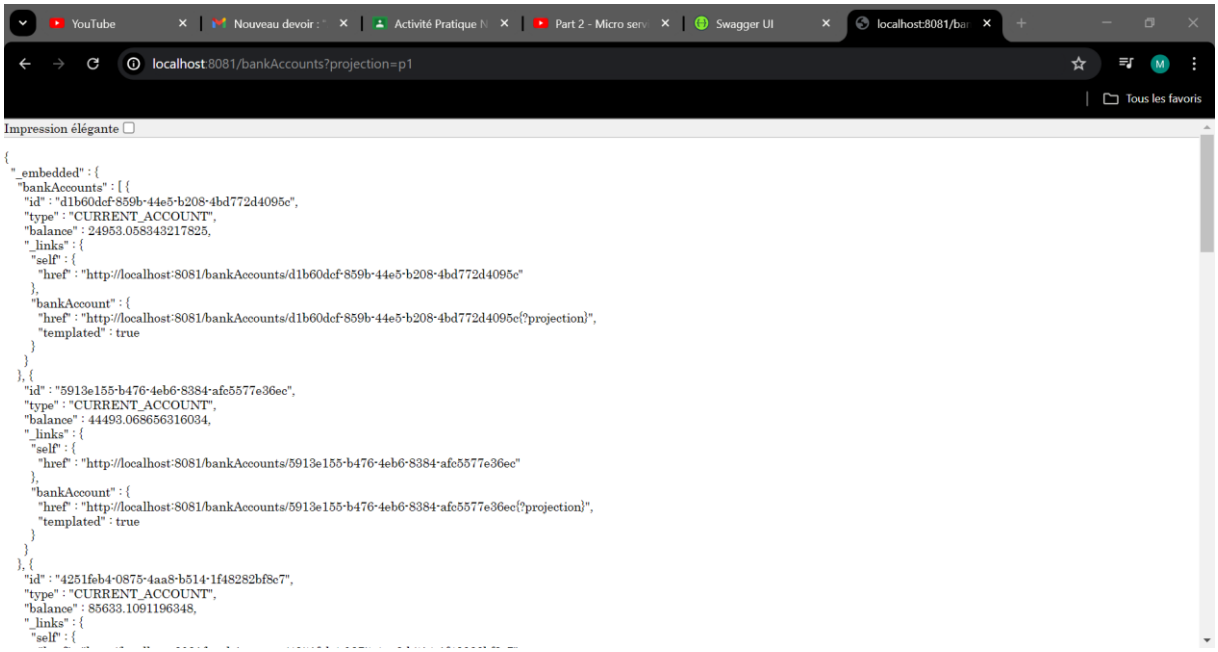


Spring database donne une possibilité d'utiliser ce qu'on appelle les projections et avec spring database on va créer une interface dans entities et on utilise l'annotation projection

AccountProjection.java

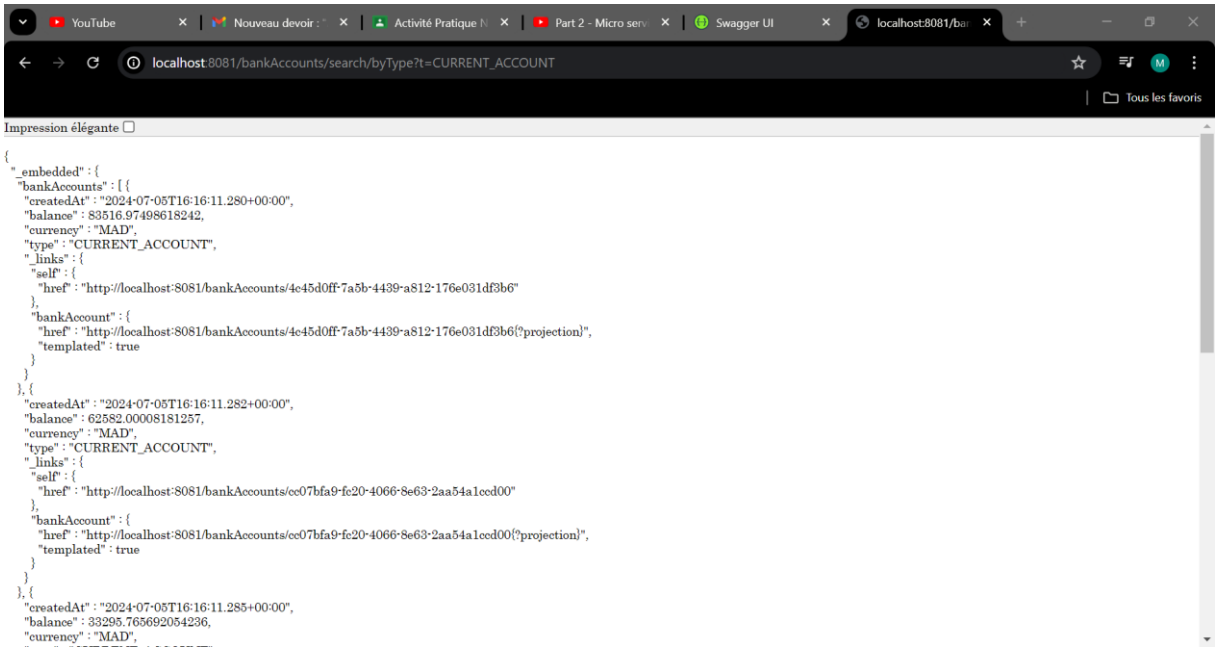
```java
package org.sid.bank_account_service.entities;

import org.sid.bank_account_service.enums.AccountType;
import org.springframework.data.rest.core.config.Projection;

@Projection(types = BankAccount.class, name = "p1")
public interface AccountProjection {
    public String getId();
    public AccountType getType();
    public Double getBalance();
}
```

La projection se fait seulement sur les attributs qu'on a cité

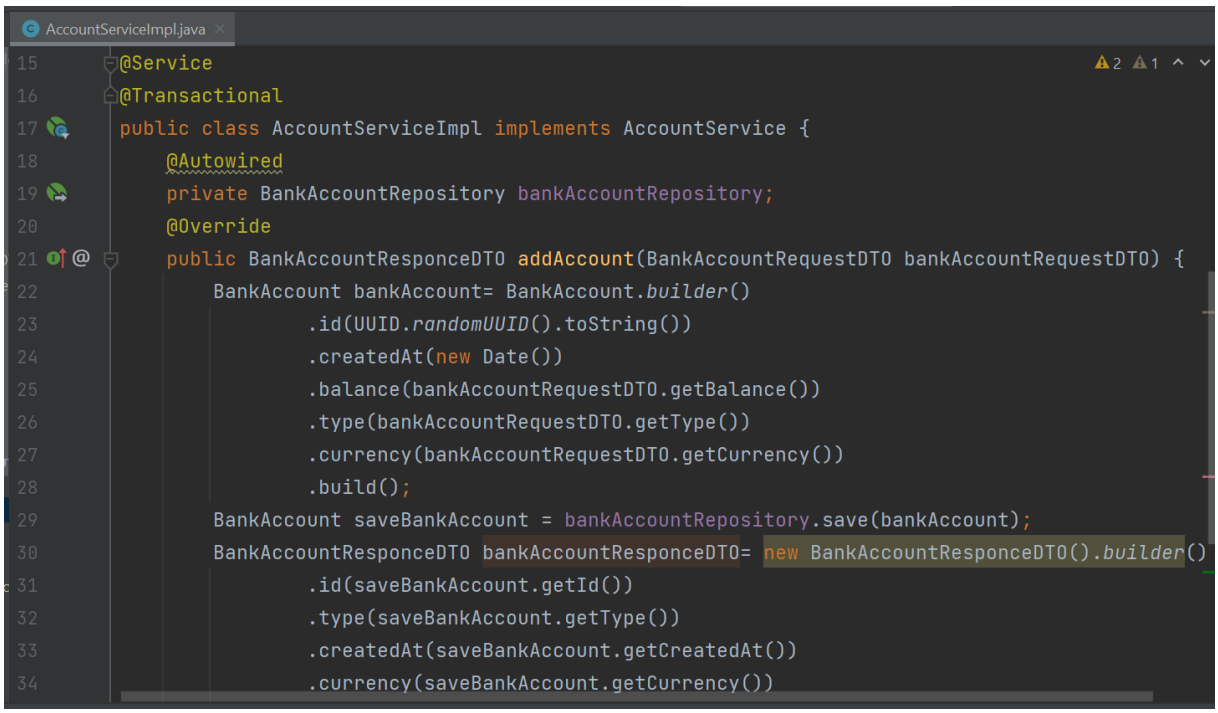



```
{
  "_embedded" : {
    "bankAccounts" : [ {
      "id" : "d1b60dcf-859b-44e5-b208-4bd772d4095c",
      "type" : "CURRENT_ACCOUNT",
      "balance" : 24953.058343217825,
      "_links" : {
        "self" : {
          "href" : "http://localhost:8081/bankAccounts/d1b60dcf-859b-44e5-b208-4bd772d4095c"
        },
        "bankAccount" : {
          "href" : "http://localhost:8081/bankAccounts/d1b60dcf-859b-44e5-b208-4bd772d4095c{?projection}",
          "templated" : true
        }
      }
    }, {
      "id" : "5913e155-b476-4eb6-8384-afc5577e36ec",
      "type" : "CURRENT_ACCOUNT",
      "balance" : 44493.068656316034,
      "_links" : {
        "self" : {
          "href" : "http://localhost:8081/bankAccounts/5913e155-b476-4eb6-8384-afc5577e36ec"
        },
        "bankAccount" : {
          "href" : "http://localhost:8081/bankAccounts/5913e155-b476-4eb6-8384-afc5577e36ec{?projection}",
          "templated" : true
        }
      }
    }, {
      "id" : "4251feb4-0875-4aa8-b514-1f48282bf8c7",
      "type" : "CURRENT_ACCOUNT",
      "balance" : 85633.1091196348,
      "_links" : {
        "self" : {
```


On ajoute les annotations @RestRessource et @Param pour les alias de méthode et le paramètre respectivement



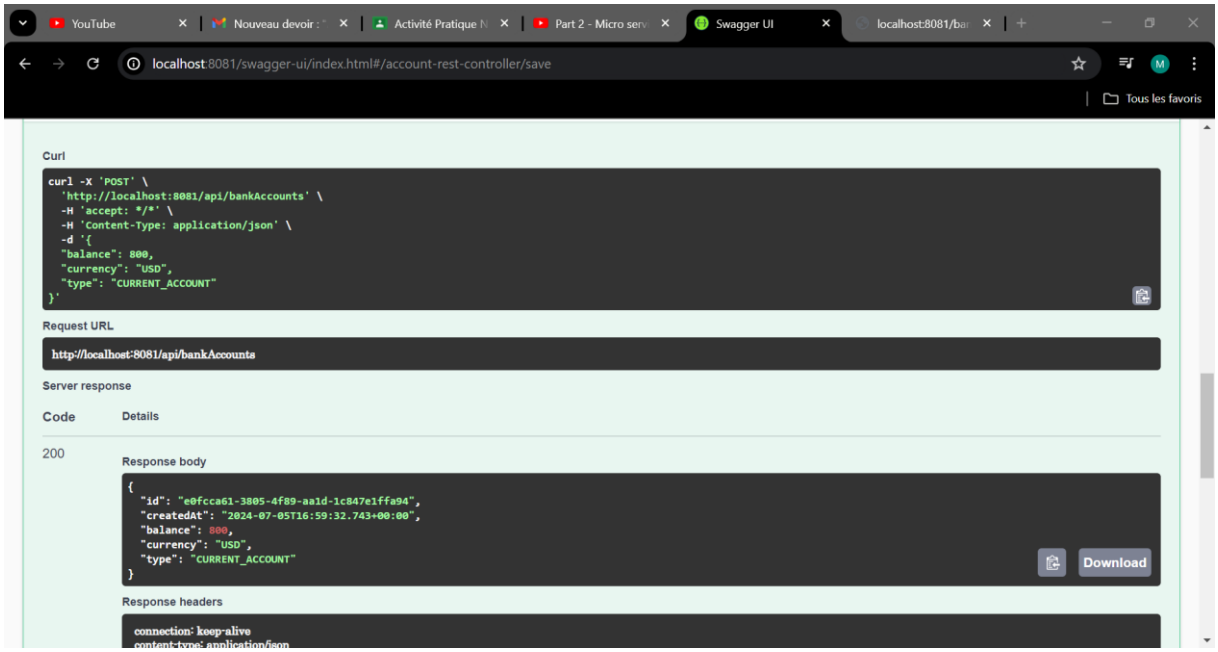
Maintenant, il faut créer la couche service et DTO pour respecter les normes
On l'interface AccountService, on va créer une implémentation et on ajoute les annotations @Service et @Transactionnal pour la couche service et on va créer un objet BankAccount et on enregistre ce compte et après on va transférer les DTO vers les entités (le mapping)



Et on va changer dans le contrôleur la méthode save

```java
public AccountRestController(BankAccountRepository bankAccountRepository, AccountServic
    this.bankAccountRepository= bankAccountRepository;

    this.accountService = accountService;
}
@GetMapping("/bankAccounts")
public List<BankAccount> bankAccounts() { return bankAccountRepository.findAll(); }

@GetMapping("/bankAccounts/{id}")
public BankAccount bankAccount(@PathVariable String id) {
    return bankAccountRepository.findById(id)
            .orElseThrow(() ->new RuntimeException(String.format("Account %s not found"
}
@PostMapping("/bankAccounts")
public BankAccountResponceDTO save(@RequestBody BankAccountRequestDTO requestDTO) {
    return accountService.addAccount(requestDTO);
}
@PutMapping("/bankAccounts/{id}")
public BankAccount update(@PathVariable String id, @RequestBody BankAccount bankAccount
```

On teste



Pour respecter les normes, on ajoute un package qui s'appelle mappers dans lequel on va créer la classe AccountMapper

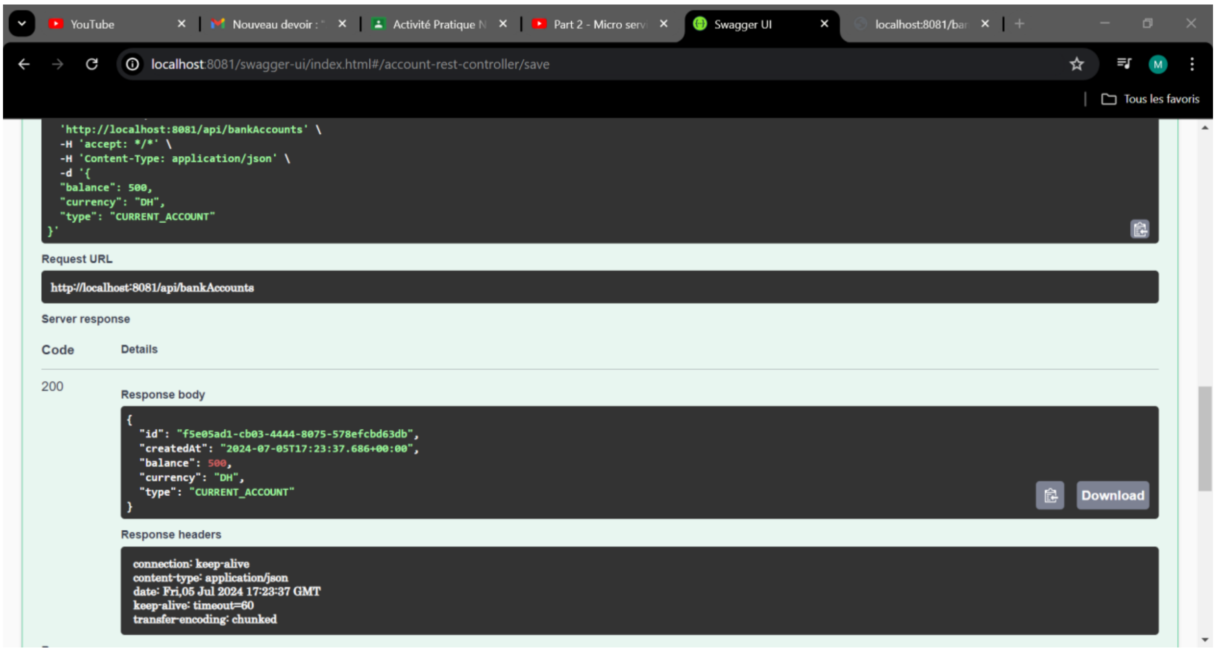
```java
import org.springframework.beans.BeanUtils;
import org.springframework.stereotype.Component;

@Component
public class AccountMapper {
    public BankAccountResponceDTO fromBankAccount(BankAccount bankAccount) {
        BankAccountResponceDTO bankAccountResponceDTO= new BankAccountResponceDTO();
        BeanUtils.copyProperties(bankAccount, bankAccountResponceDTO);
        return bankAccountResponceDTO;

    }
}
```

```java
                .balance(bankAccountRequestDTO.getBalance())
                .type(bankAccountRequestDTO.getType())
                .currency(bankAccountRequestDTO.getCurrency())
                .build();
        BankAccount saveBankAccount = bankAccountRepository.save(bankAccount);
        BankAccountResponceDTO bankAccountResponceDTO=accountMapper.fromBankAccount(saveBankAc

        return bankAccountResponceDTO;
    }
}
```

```java
    @GetMapping("/bankAccounts/{id}")
    public BankAccount bankAccount(@PathVariable String id) {
        return bankAccountRepository.findById(id)
                .orElseThrow(() ->new RuntimeException(String.format("Account %s not found",
    }
    @PostMapping("/bankAccounts")
    public BankAccountResponceDTO save(@RequestBody BankAccountRequestDTO requestDTO) {
        return accountService.addAccount(requestDTO);
    }
    @PutMapping("/bankAccounts/{id}")
    public BankAccount update(@PathVariable String id, @RequestBody BankAccount bankAccount)
        BankAccount account=bankAccountRepository.findById(id).orElseThrow();
```

```java
import org.springframework.beans.BeanUtils;
import org.springframework.stereotype.Component;

@Component
public class AccountMapper {
    public BankAccountResponceDTO fromBankAccount(BankAccount bankAccount) {
        BankAccountResponceDTO bankAccountResponceDTO= new BankAccountResponceDTO();
        BeanUtils.copyProperties(bankAccount, bankAccountResponceDTO);
        return bankAccountResponceDTO;

    }
}
```

On teste



Partie 2 :

Un micro service peut avoir plusieurs connecteurs, il est fait pour communiquer avec le monde extérieur par la communication rest et on peut utiliser aussi soap et on peut également utiliser Graph QL ou bien GRPC

Dans le même micro service, on va créer un contrôleur qui permet de communiquer avec le web service en utilisant Graph QL

Et pour utiliser Spring Graph QL, on aura besoin de créer un schéma

On crée un dossier graphql contient le file schema.graphqls, dans lequel on décle les types comme le type Query dont on spécifie les requetes à implémenter dans le service par exemple on veut un service qui permet de consulter la liste des comptes

```
type Query{
    accountsList : [BankAccount]

}

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

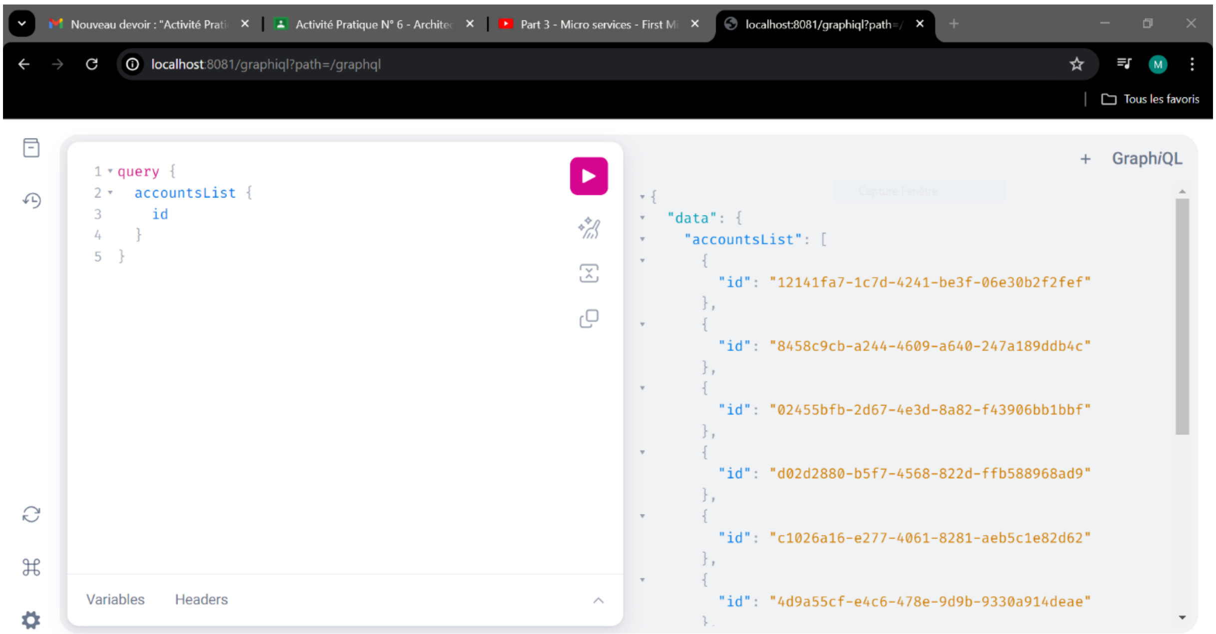
Et maintenant, on va implémenter le service, on crée la classe BankAccountGraphQLController c'est un controller et après on déclare BankAccountRepository pour accéder à la base de données puis injecter les données en utilisant @Autowired et on va créer une méthode qui permet de retourner la liste et on utilise l'annotation @QueryMapping pour comme dire automatiquement quand le client demande accountsList il va exécuter une méthode qui s'appelle accountsList (on a les mêmes noms)

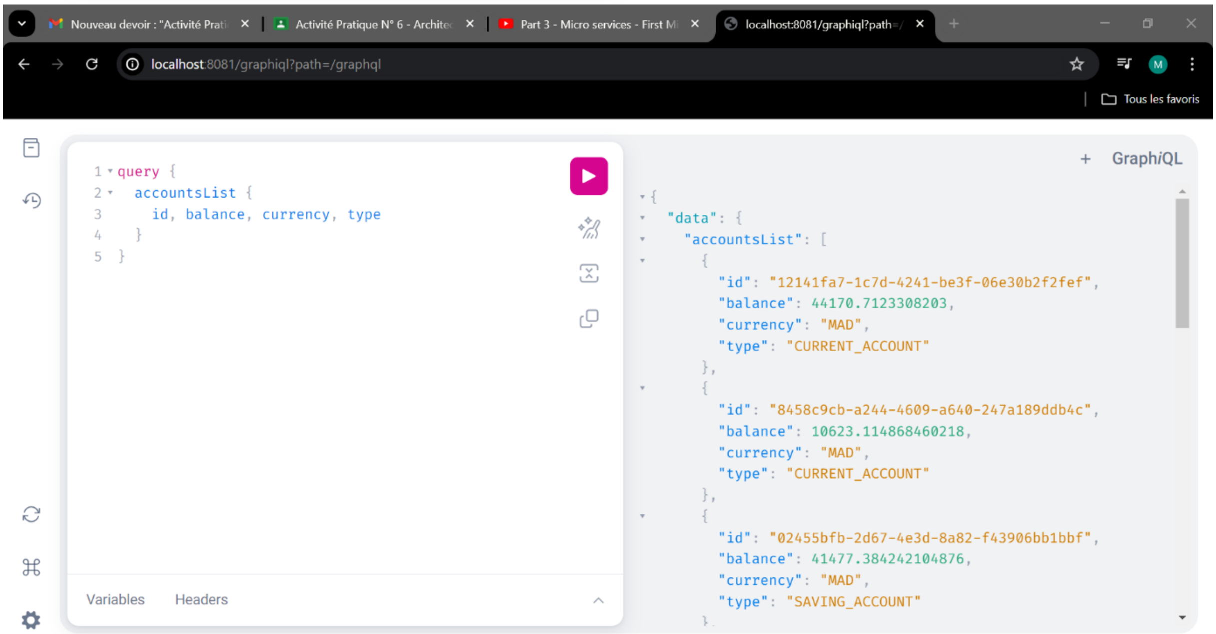
```java
import org.sid.bank_account_service.repositories.BankAccountRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.graphql.data.method.annotation.QueryMapping;
import org.springframework.stereotype.Controller;

import java.util.List;

@Controller
public class BankAccountGraphQLController {

    @Autowired
    private BankAccountRepository bankAccountRepository;
    @QueryMapping
    public List<BankAccount> accountsList() {
        return bankAccountRepository.findAll();
    }
}
```

Pour que GraphQL puisse fonctionner nous aurons besoin d'ajouteer une ligne dans fichier properties

```
spring.application.name=bank-account-service
spring.datasource.url=jdbc:h2:mem:account-db
spring.h2.console.enabled=true
server.port=8081
spring.graphql.graphiql.enabled=true
```

On teste





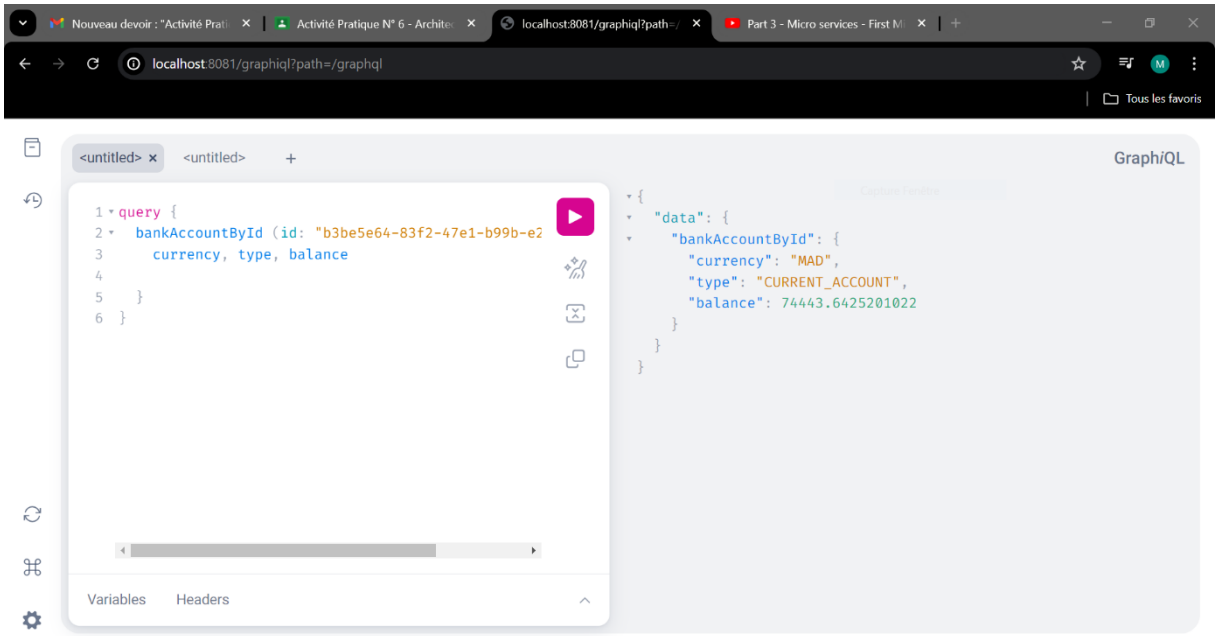
Supposant on veut récupérer un compte on va créer une autre méthode

```java
public class BankAccountGraphQLController {

    @Autowired
    private BankAccountRepository bankAccountRepository;
    @QueryMapping
    public List<BankAccount> accountsList() {
        return bankAccountRepository.findAll();
    }
    @QueryMapping
    public BankAccount bankAccountById(@Argument String id) {
        return bankAccountRepository.findById(id)
                .orElseThrow(()-> new RuntimeException(String.format("Account %s not found",id)));
    }
}
```

Pour utiliser cette méthode il va falloir faire ce travail dans le schèma

```
type Query{
    accountsList : [BankAccount],
    accountById (id: String) : BankAccount

}

type BankAccount {
    id: String,
    createdAt: Float,
    balance: Float,
    currency: String,
    type: String

}
```

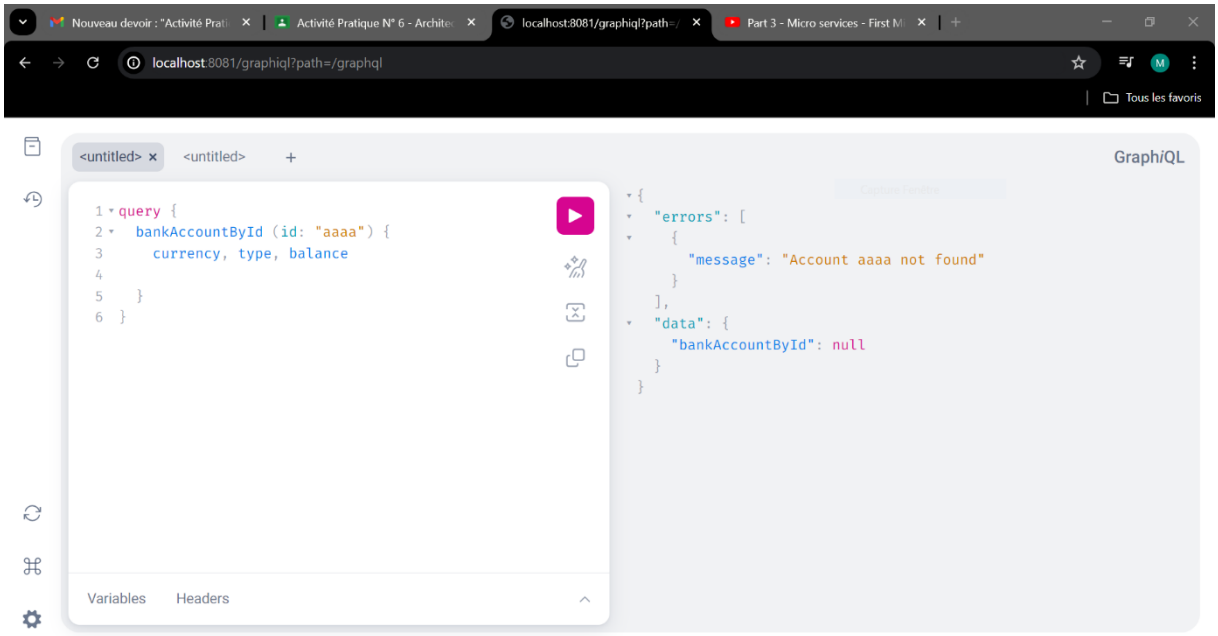
On teste



Maintenant comment on va récupérer le message d'exception et pour ce faire, on ajoute un handler d'exception

```java
import graphql.ErrorClassification;
import graphql.GraphQLError;
import graphql.language.SourceLocation;
import graphql.schema.DataFetchingEnvironment;
import org.springframework.graphql.execution.DataFetcherExceptionResolverAdapter;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
public class CustomDataFetcherExceptionResolver extends DataFetcherExceptionResolverAdapter {
    @Override
    protected GraphQLError resolveToSingleError(Throwable ex, DataFetchingEnvironment env) {
        return new GraphQLError() {
            @Override
            public String getMessage() {
                return ex.getMessage();
            }
```

On teste



Supposons maintenant qu'on va ajouter une méthode qui permet d'ajouter un compte (Dans le contrôleur) et pour mapper la méthode on utilise @MutationMapping et après on déclare la méthode dans le schéma

```java
public class BankAccountGraphQLController {

    @Autowired
    private BankAccountRepository bankAccountRepository;
    @QueryMapping
    public List<BankAccount> accountsList() {
        return bankAccountRepository.findAll();
    }
    @QueryMapping
    public BankAccount bankAccountById(@Argument String id) {
        return bankAccountRepository.findById(id)
                .orElseThrow(()-> new RuntimeException(String.format("Account %s not fou
    }
    @MutationMapping
    public BankAccount addAccount(@Argument BankAccount bankAccount) {
        return bankAccountRepository.save(bankAccount);
    }
}
```

```
type Mutation {
    addAccount(bankAccount: BankAccountDTO) : BankAccount
}

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

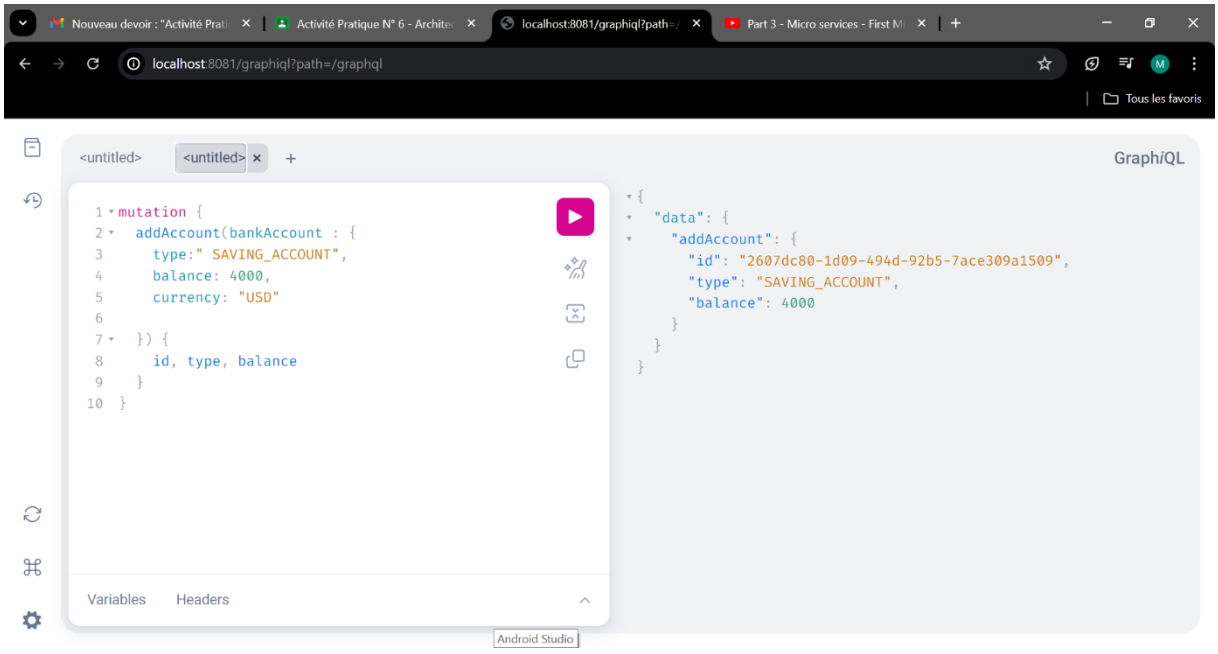
On utilise le type record qui est un objet dans lequel on spécifie les paramètres de constructeur dont on a besoin

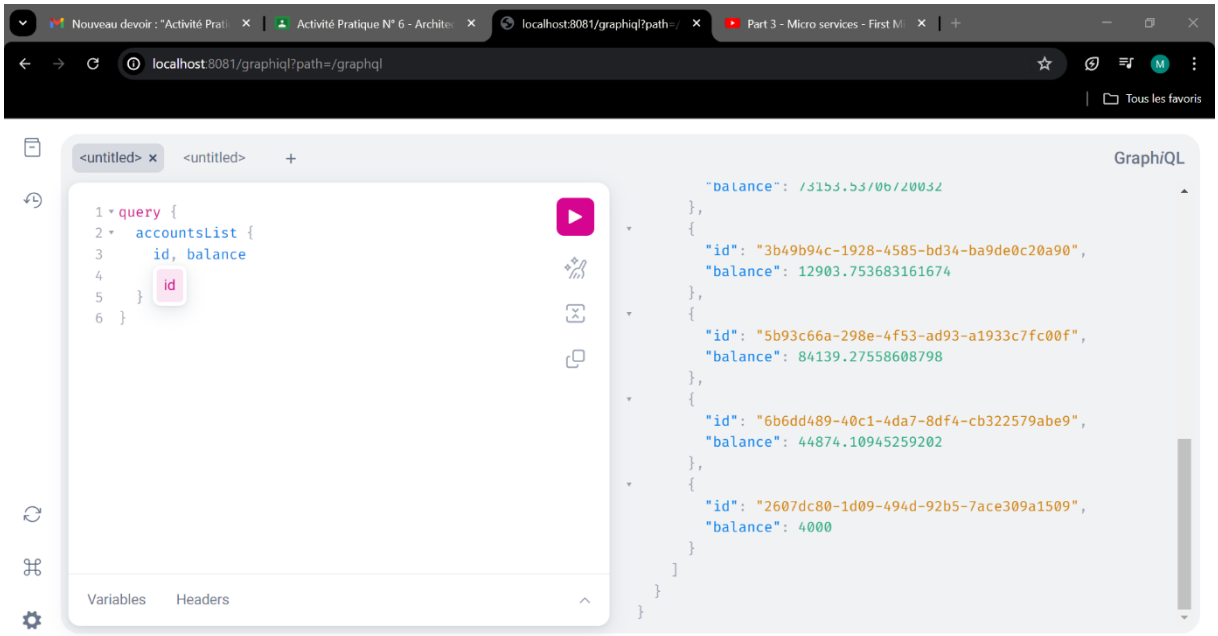
```
record BankAccountDTO(Double balance, String type, String currency ) {

}
```

Pour créer une mutation

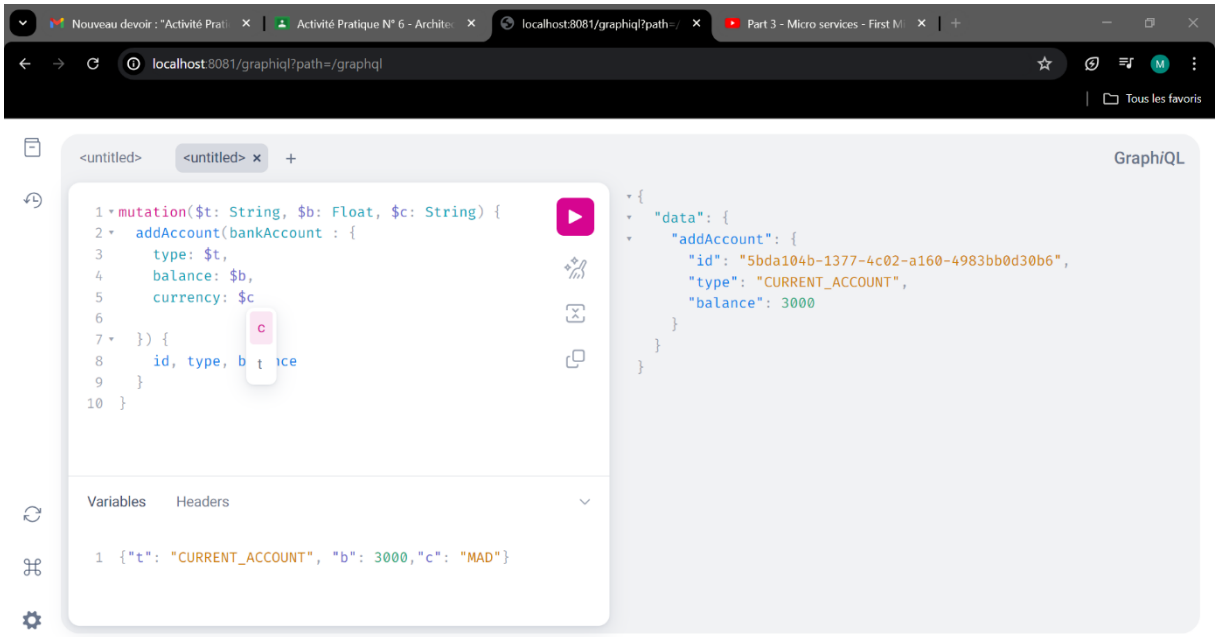
```java
    @Autowired
    private AccountService accountService;
    @QueryMapping
    public List<BankAccount> accountsList() {
        return bankAccountRepository.findAll();
    }
    @QueryMapping
    public BankAccount bankAccountById(@Argument String id) {
        return bankAccountRepository.findById(id)
                .orElseThrow(()-> new RuntimeException(String.format("Account %s not found",id
    }
    @MutationMapping
    public BankAccountResponceDTO addAccount(@Argument BankAccountRequestDTO bankAccount) {
        return accountService.addAccount(bankAccount);
    }

/*record BankAccountDTO(Double balance, String type, String currency ) {

} */
}
```

Si on veut envoyer une requête, on va faire une méthode avec post dans le corps contient mutation





On peut utiliser une requête paramétrée



On va ajouter une méthode update

```java
    }
    @Override
    public BankAccountResponceDTO updateAccount(String id, BankAccountRequestDTO bankAccountRe
        BankAccount bankAccount= BankAccount.builder()
                .id(id)
                .createdAt(new Date())
                .balance(bankAccountRequestDTO.getBalance())
                .type(bankAccountRequestDTO.getType())
                .currency(bankAccountRequestDTO.getCurrency())
                .build();
        BankAccount saveBankAccount = bankAccountRepository.save(bankAccount);
        BankAccountResponceDTO bankAccountResponceDTO=accountMapper.fromBankAccount(saveBankAc

        return bankAccountResponceDTO;
    }
}
```

```java
    }
    @MutationMapping
    public BankAccountResponceDTO updateAccount(@Argument String id, @Argument BankAccountReques
        return accountService.updateAccount(id, bankAccount);
    }
```

On va déclarer la méthode dans le fichier schema.graphqls

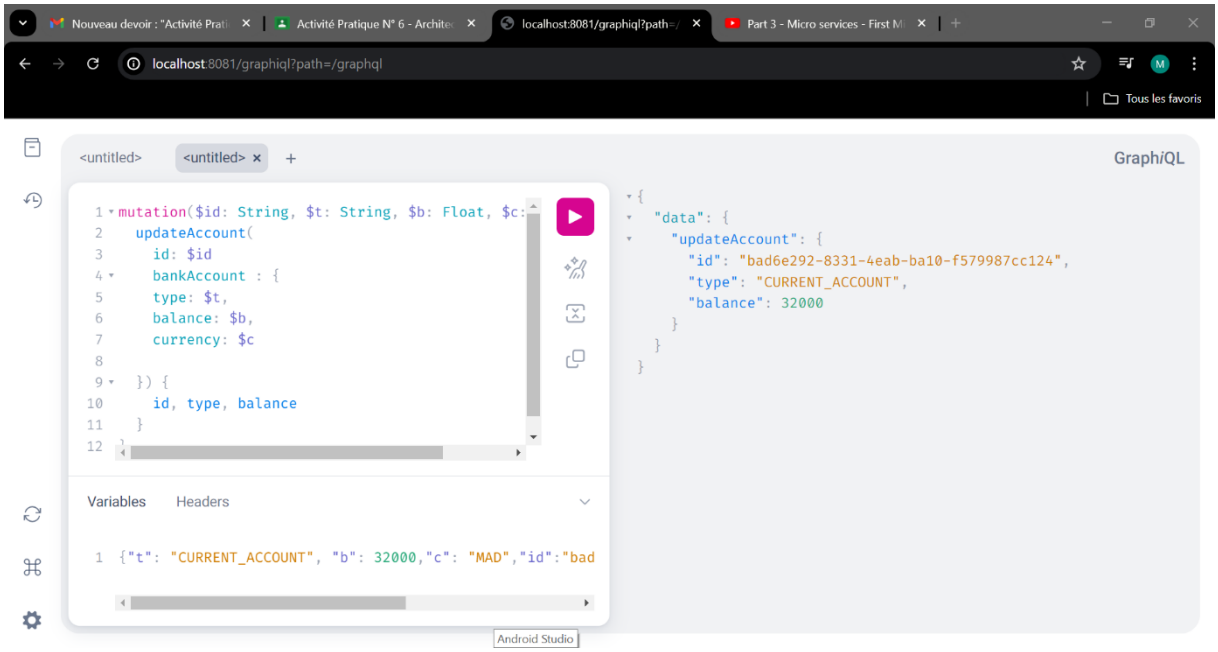
```graphql
}
type Mutation {
    addAccount(bankAccount: BankAccountDTO) : BankAccount,
    updateAccount(id: String, bankAccount: BankAccountDTO) : BankAccount
}
```

On ajoute la méthode deleteAccount

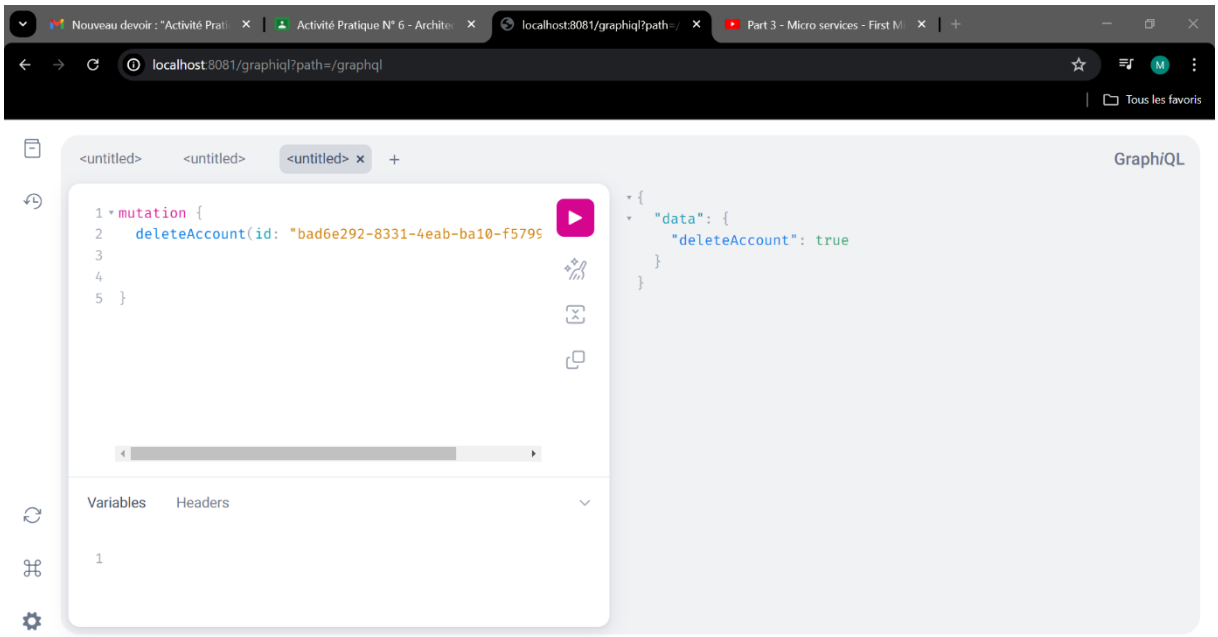
```java
    @MutationMapping
    public Boolean deleteAccount(@Argument String id) {
        bankAccountRepository.deleteById(id);
        return true;
    }
```

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

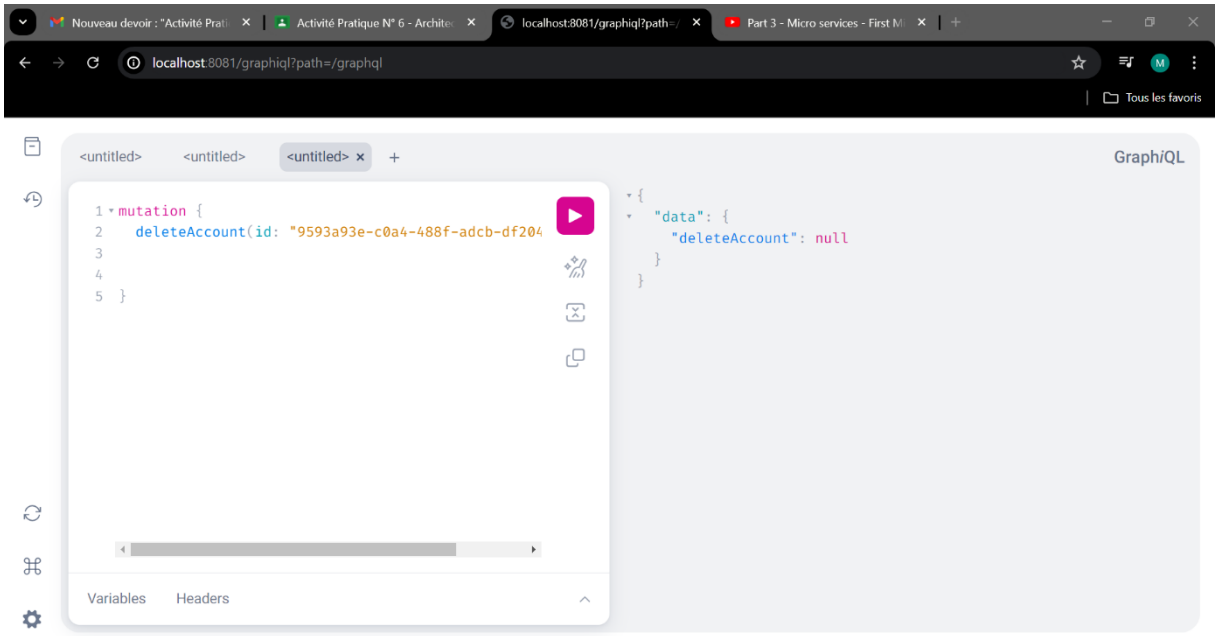
On teste pour la méthode update



Pour tester la méthode deleteAccount



Au lieu d'utiliser Boolean pour la méthode delete on utilise String, on teste :



Le dernier point c'est les relations, Supposons qu'un compte appartient à un client on crér une classe customer dans entities

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

On remplit l'application par 10 comptes par client (il y a 4 clients)

```java
@Bean
CommandLineRunner start(BankAccountRepository bankAccountRepository, CustomerRepository cu
    return args -> {
        Stream.of("Mohamed", "Yassine", "Hanae", "Imane").forEach(c -> {
            Customer customer = Customer.builder()
                    .name(c)
                    .build();
            customerRepository.save(customer);
        });
        customerRepository.findAll().forEach(customer -> {
            for (int i = 0; i < 10; i++) {
                BankAccount bankAccount = BankAccount.builder()
                        .id(UUID.randomUUID().toString())
                        .type(Math.random() > 0.5 ? AccountType.CURRENT_ACCOUNT : AccountT
```

On a comme résultat :

```
SELECT * FROM CUSTOMER
```

```
SELECT * FROM CUSTOMER;
```

ID	NAME
1	Mohamed
2	Yassine
3	Hanae
4	Imane

(4 rows, 12 ms)

Edit

```
SELECT * FROM BANK_ACCOUNT
```

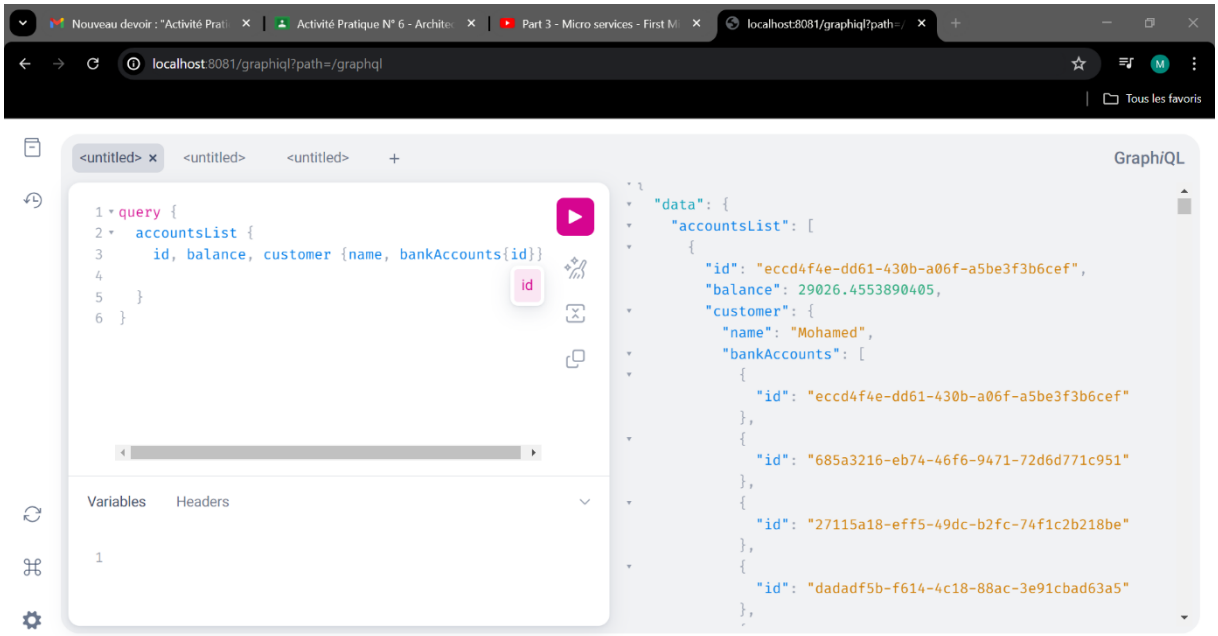
SELECT * FROM BANK_ACCOUNT;

ID	BALANCE	CREATED_AT	CURRENCY	TYPE	CUSTOMER_ID
0a8d8685-8d3e-4911-90e5-9ea3e76984e7	95724.13058576173	2024-07-06 05:29:42.286	MAD	SAVING_ACCOUNT	1
8116e4be-f9cc-4879-a468-782b9219ac9f	57689.86712407745	2024-07-06 05:29:42.305	MAD	SAVING_ACCOUNT	1
719e59e4-674a-43ac-b943-c14a309e43c9	72888.2868586081	2024-07-06 05:29:42.306	MAD	SAVING_ACCOUNT	1
4a826c6f-fc98-4b2d-81b5-214d064bec1f	21958.35334700341	2024-07-06 05:29:42.308	MAD	CURRENT_ACCOUNT	1
db05d820-6072-4a83-a351-b47388504ad4	29479.803592736767	2024-07-06 05:29:42.309	MAD	CURRENT_ACCOUNT	1
cd9e2575-be60-413d-9d9d-b003dc725113	17343.217931493637	2024-07-06 05:29:42.311	MAD	SAVING_ACCOUNT	1
05071cec-3bcf-45b4-8856-867e51762b05	45492.09195410184	2024-07-06 05:29:42.312	MAD	CURRENT_ACCOUNT	1
396cc748-fd7e-4b63-a751-ed1ff3d8c025	80976.44686937286	2024-07-06 05:29:42.314	MAD	SAVING_ACCOUNT	1
d64ecd55-3c6d-4340-9b02-194519e123e8	18321.233623271706	2024-07-06 05:29:42.315	MAD	SAVING_ACCOUNT	1
7daba7ba-cb61-4e2b-803b-add752b9a415	76940.96280764975	2024-07-06 05:29:42.317	MAD	CURRENT_ACCOUNT	1
c01d3279-5942-4472-b0df-63c0f50c9e5d	80203.63311574711	2024-07-06 05:29:42.318	MAD	SAVING_ACCOUNT	2
1a15c7b5-5def-4f4f-8b20-30ad8390ed36	66396.3829598157	2024-07-06 05:29:42.319	MAD	SAVING_ACCOUNT	2

Pour notre API Graphql, on ajoute le customer pour le demander dans le schema

```
type Customer {
    id: ID,
    name: String,
    bankAccounts: [BankAccount]

}

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

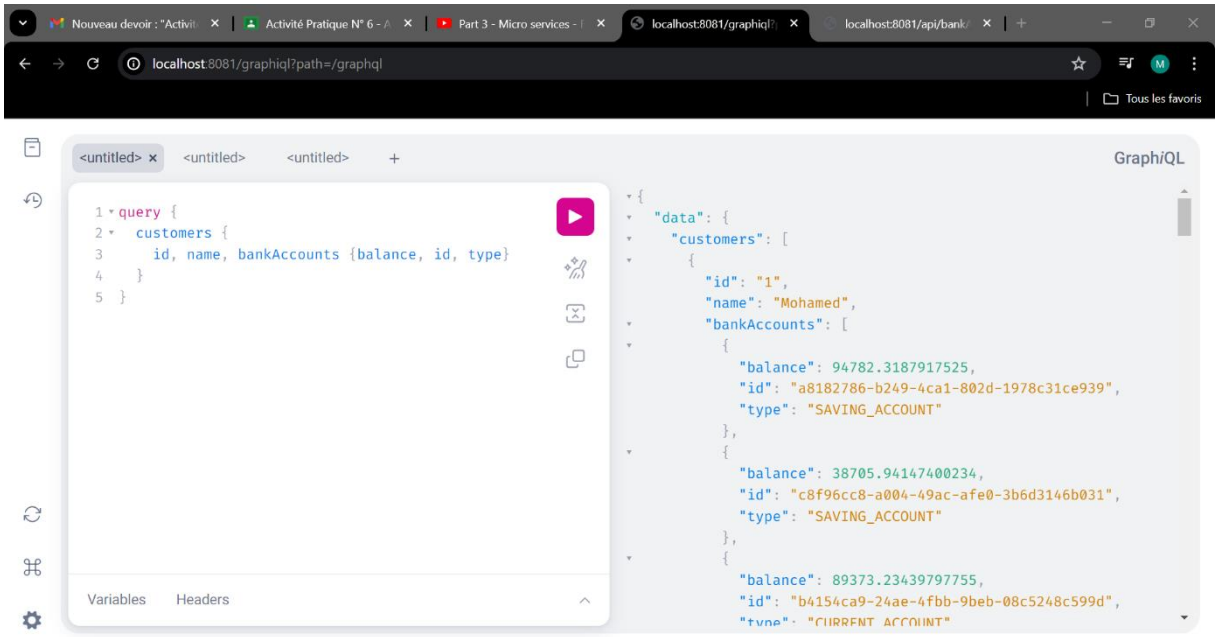
On teste



Maintenant, on ajoute une méthode qui permet de consulter customer et on implémente cette méthode dans graphql

```
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) : String
}

type Customer {
    id: ID,
    name: String,
    bankAccounts: [BankAccount]
```

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

}
```



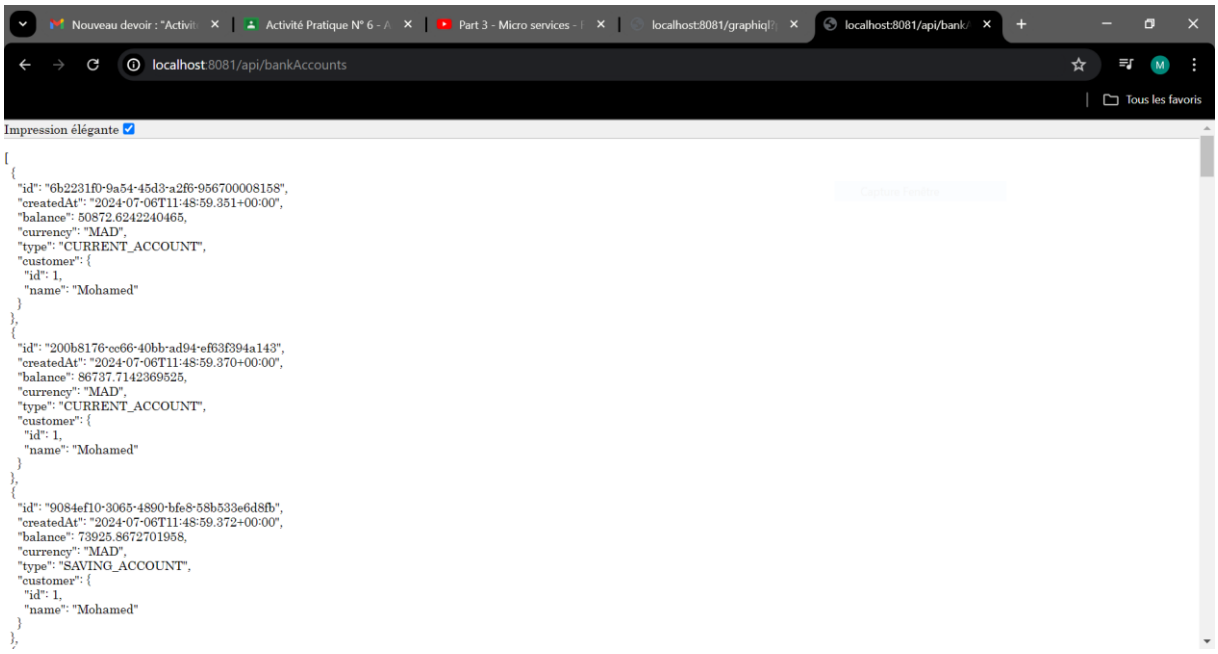
Maintenant si on demande api/bankAccounts, une exception sera générée

```
at com.fasterxml.jackson.databind.ser.BeanSerializer.serialize(BeanSerializer.java:178) ~[jackson-databind-2.13.3.jar:2
at com.fasterxml.jackson.databind.ser.BeanPropertyWriter.serializeAsField(BeanPropertyWriter.java:728) ~[jackson-databi
at com.fasterxml.jackson.databind.ser.std.BeanSerializerBase.serializeFields(BeanSerializerBase.java:774) ~[jackson-dat
at com.fasterxml.jackson.databind.ser.BeanSerializer.serialize(BeanSerializer.java:178) ~[jackson-databind-2.13.3.jar:2
at com.fasterxml.jackson.databind.ser.std.CollectionSerializer.serializeContents(CollectionSerializer.java:145) ~[jacks
at com.fasterxml.jackson.databind.ser.std.CollectionSerializer.serialize(CollectionSerializer.java:107) ~[jackson-datab
at com.fasterxml.jackson.databind.ser.std.CollectionSerializer.serialize(CollectionSerializer.java:25) ~[jackson-databi
at com.fasterxml.jackson.databind.ser.BeanPropertyWriter.serializeAsField(BeanPropertyWriter.java:728) ~[jackson-databi
at com.fasterxml.jackson.databind.ser.std.BeanSerializerBase.serializeFields(BeanSerializerBase.java:774) ~[jackson-dat
```

Pour éviter ce problème, il faut passer vers les DTO ou bien aller vers Customer

```java
import javax.persistence.*;
import java.util.List;

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

Et si on fait ça, ça va marcher pour rest et non pas graphql (il ne change rien)



Deuxième partie : Développer une architecture micro-service

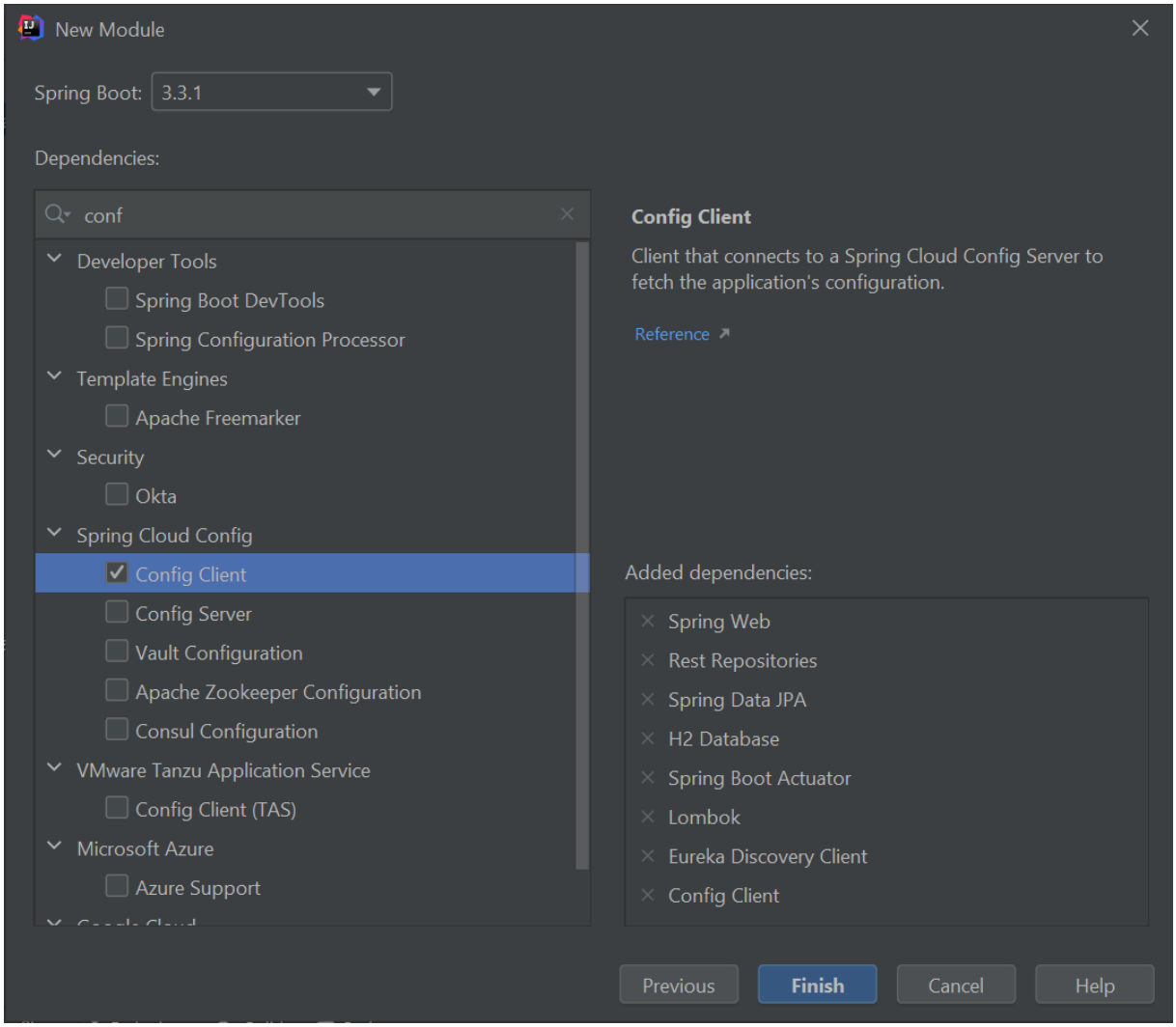
Partie 1 :

L'approche micro service est un style d'architecture qui consiste à découper un grand domaine metier en un ensemble sous domaines pour chacun on crée une solution indépendante

On utilise des services techniques pour que ces micro services puisse fonctionner dans le cadre d'une même application

Dans cette partie nous allons voir comment mettre en place un exemple d'architecture micro service

On va créer un nouveau projet bank-account-service et on ajouter des modules (un projet de plusieurs modules dont chaque module représente un micro service) avec l'ajout des dépendances suivantes pour les modules customer-service et account-service



Et on va créer le module gateway-service (Spring cloud gateway est une gateway qui est basée sur les entrées/sorties non bloquantes), puis discovery-service et config-service et à chacun ses propres dépendances

On va commencer par customer-service et on va le tester indépendamment et après on va voir comment faire pour qu'il puisse s'enregistrer au niveau Discovery une fois qu'on va créer les micro services.

Dans customer-service, on va créer les packages web, entities, repository, service et mapper et on crée l'entité Customer et on ajoute les annotations lombok

Customer.java

```java
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.*;

@Entity
@Getter @Setter @ToString @NoArgsConstructor @AllArgsConstructor @Builder
public class Customer {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
}
```

On crée l'interface CustomerRepository et on peut ajouter l'annotation @RepositoryRestResource de spring data Rest il permet de démarrer automatiquement un web service restfull qui permet de gérer les customers mais on ne le fait pas

On crée CustomerRestController dans package web c'est un contrôleur et on injecte les dépendances par un constructeur avec paramètres et après on crée une méthode pour consulter la liste des costumers et une méthode pour consulter un customer

CustomerRestController.java

```java
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
public class CustomerRestController {

    private CustomerRepository customerRepository;

    public CustomerRestController(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }
    @GetMapping("/customers")
    public List<Customer> customerList() {
        return customerRepository.findAll();
    }
    @GetMapping("/customers/{id}")
    public Customer customerById(@PathVariable Long id) {
        return customerRepository.findById(id).get();
    }
```

On va insérer quelques customers dans la base de données et on ajour l'annotation @Bean et on rappelle que chaque méthode utilise l'annotation Bean avec spring c'est-à-dire c'est une méthode qui va s'exécuter au démarrage spring par définition

On a trois possibilités pour créer un customer la première c'est un constructeur sans paramètre, la deuxième c'est utiliser un constructeur avec tous les paramètres et la troisième c'est d'utiliser Builder

CustomerServiceApplication.java

```
14
15  blic static void main(String[] args) { SpringApplication.run(CustomerServiceApplication
18  ean
19  mmandLineRunner commandLineRunner(CustomerRepository customerRepository) {
20      return args ->{
21          List<Customer> customerList= List.of(
22          Customer.builder()
23                  .firstName("Hassan")
24                  .lastName("Elhoumi")
25                  .email("hassan@gmail.com")
26                  .build(),
27          Customer.builder()
28                  .firstName("Mohamed")
29                  .lastName("Elhannaoui")
30                  .email("hassan@gmail.com")
31                  .build()
32          );
33          customerRepository.saveAll(customerList);
34
35      };
```

On fait le test et une exception est générée

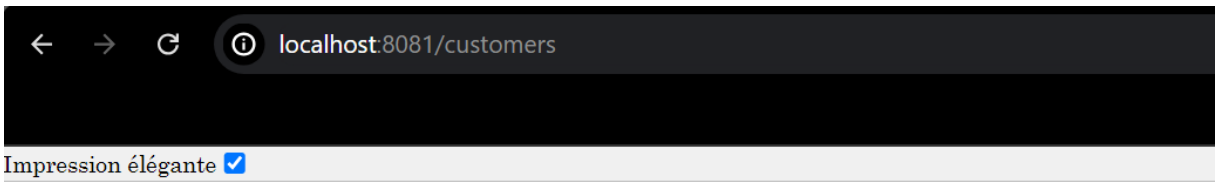
```
Action:

Add a spring.config.import=configserver: property to your configuration.
    If configuration is not required add spring.config.import=optional:configserver: instead.
    To disable this check, set spring.cloud.config.enabled=false or
    spring.cloud.config.import-check.enabled=false.


Process finished with exit code 1
```

Pour résoudre ce problème on va désactiver les services de configuration de discovery dans fichier propreties et on fait la configuration qu'on a besoin

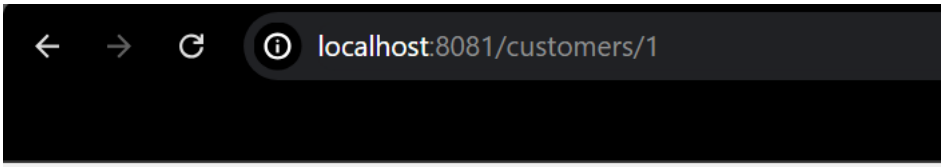
application.properties

```
1  spring.application.name=customer-service
2  server.port=8081
3  spring.datasource.url=jdbc:h2:mem:customer-db
4  spring.h2.console.enabled=true
5  spring.cloud.discovery.enabled=false
6  spring.cloud.config.enabled=false
7
```

On teste

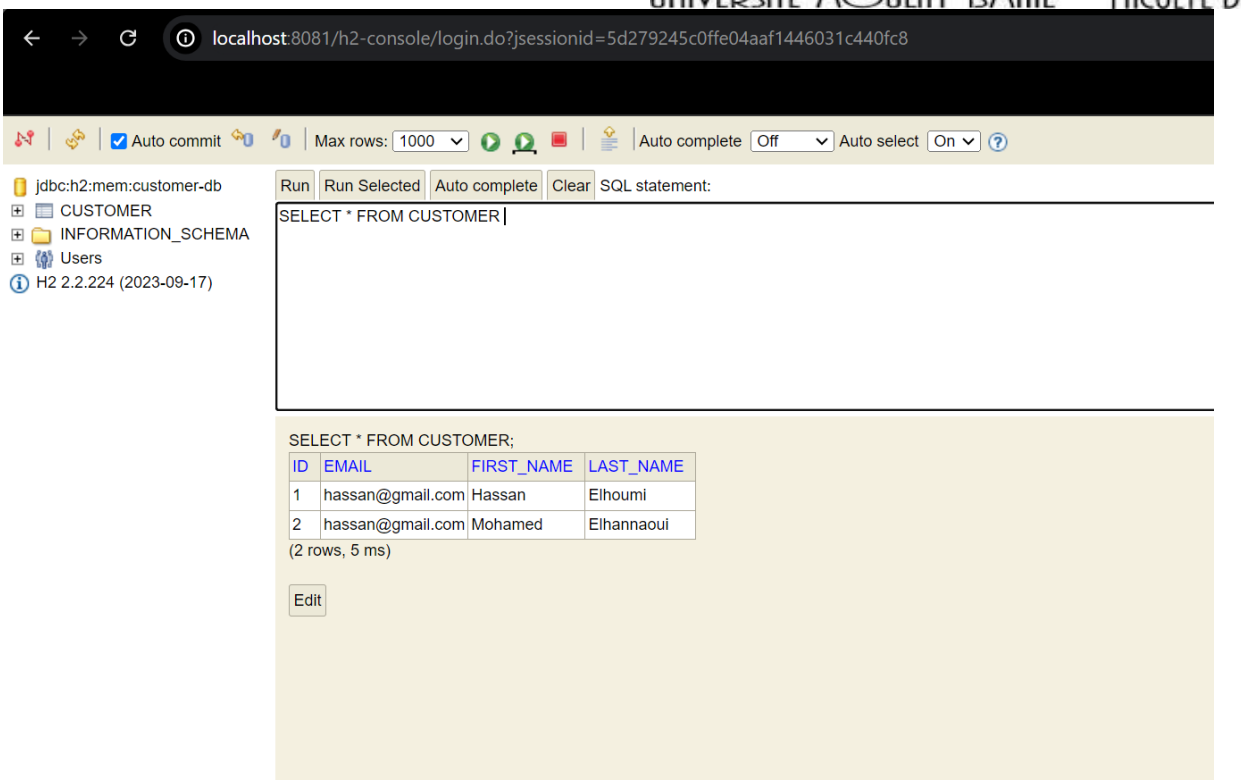



```
[
  {
    "id": 1,
    "firstName": "Hassan",
    "lastName": "Elhoumi",
    "email": "hassan@gmail.com"
  },
  {
    "id": 2,
    "firstName": "Mohamed",
    "lastName": "Elhannaoui",
    "email": "hassan@gmail.com"
  }
]
```


```
{
  "id": 1,
  "firstName": "Hassan",
  "lastName": "Elhoumi",
  "email": "hassan@gmail.com"
}
```


Si on veut consulter la base de données H2



Passant maintenant à account-service dont on va créer les packages habituels et on va créer une entité BankAccount qui contient comme attribut une simple classe Customer qui n'est pas une entité jpa et on va utiliser l'annotation @Tresient pour ignorer cet attribut

BankAccount.java

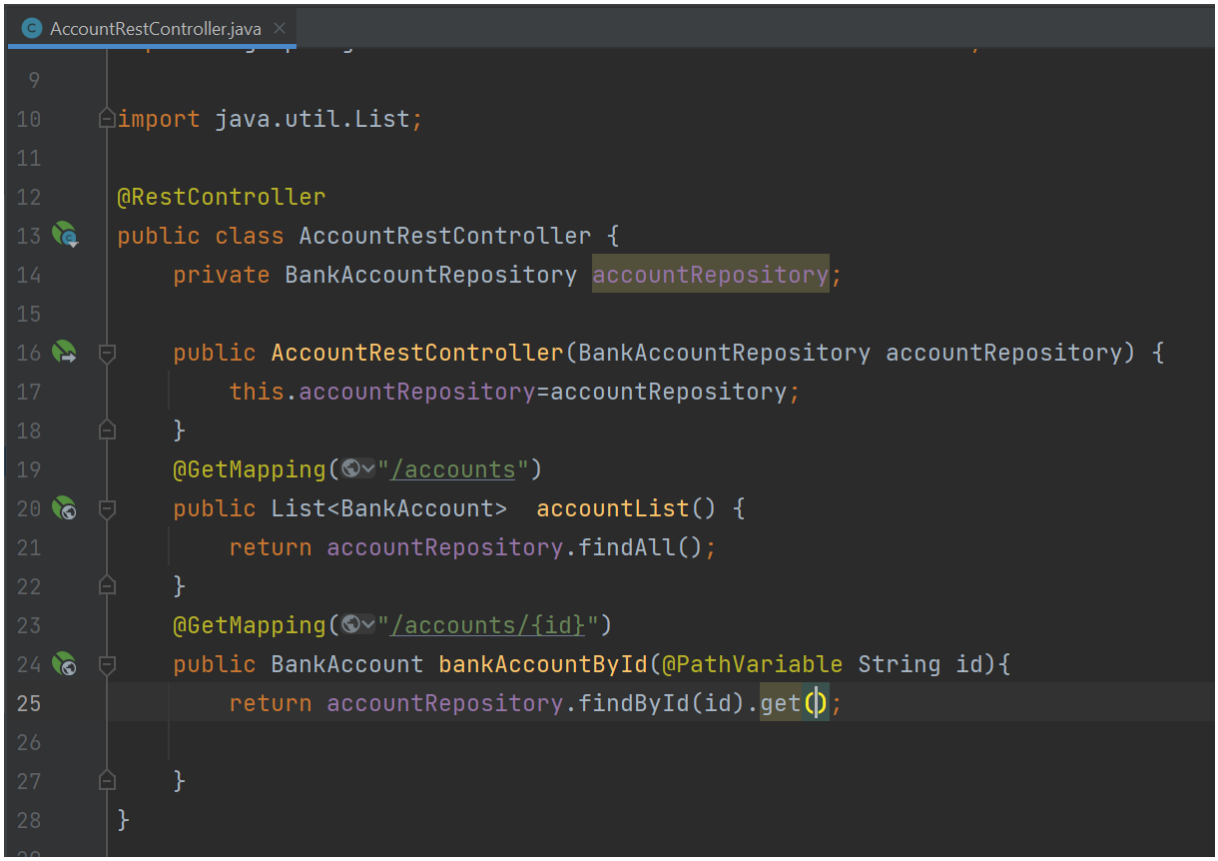
```java
package net.illa.accountservice.entities;

import jakarta.persistence.*;
import net.illa.accountservice.enums.AccountType;
import net.illa.accountservice.model.Customer;

import java.time.LocalDate;
@Entity
public class BankAccount {
    @Id
    private String accountId;
    private double balance;
    private LocalDate createdAt;
    private String currency;
    @Enumerated(EnumType.STRING)
    private AccountType type;
    @Transient
    private Customer customer;
    private Long customerId;
}
```

Voilà la classe simple Customer

Customer.java

```java
package net.illa.accountservice.model;

import lombok.Getter;
import lombok.Setter;
import lombok.ToString;

@Getter @Setter @ToString
public class Customer {
    private Long id;
    private String firstname;
    private String lastName;
    private String email;
}
```

Après on crée l'interface AccountRepository et le web service AccountRestContoller dans lequel on crée 2 méthodes la première pour consulter la liste comptes et la deuxième pour consulter un compte par l'id



Alors on va faire le test

AccountServiceApplication.java

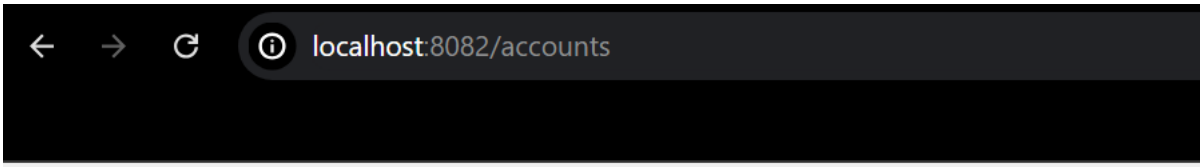
```java
    CommandLineRunner commandLineRunner(BankAccountRepository bankAccountReposit
        return args ->{
            BankAccount bankAccount1=new BankAccount().builder()
                            .accountId(UUID.randomUUID().toString())
                            .currency("MAD")
                            .balance(98000)
                            .createdAt(LocalDate.now())
                            .type(AccountType.CURRENT_ACCOUNT)
                            .customerId(Long.valueOf(1))
                            .build();
            BankAccount bankAccount2=new BankAccount().builder()
                        .accountId(UUID.randomUUID().toString())
                        .currency("MAD")
                        .balance(12000)
                        .createdAt(LocalDate.now())
                        .type(AccountType.SAVING_ACCOUNT)
                        .customerId(Long.valueOf(2))
                        .build();
            bankAccountRepository.save(bankAccount1);
            bankAccountRepository.save(bankAccount2);
```

Avant de démarrer ce micro service, on a besoin de le configurer

application.properties

```properties
spring.application.name=account-service
spring.datasource.url=jdbc:h2:mem:account-db
spring.h2.console.enabled=true
server.port=8082
spring.cloud.discovery.enabled=false
spring.cloud.config.enabled=false
```

On a le résultat



Impression élégante ☑


```
[
  {
    "accountId": "e459ccc5-f535-4648-bd5a-889e79a1c444",
    "balance": 98000,
    "createdAt": "2024-07-07",
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT",
    "customer": null,
    "customerId": 1
  },
  {
    "accountId": "2e3afdbd-4c2c-45c0-8cee-1a9db673eb91",
    "balance": 12000,
    "createdAt": "2024-07-07",
    "currency": "MAD",
    "type": "SAVING_ACCOUNT",
    "customer": null,
    "customerId": 2
  }
```

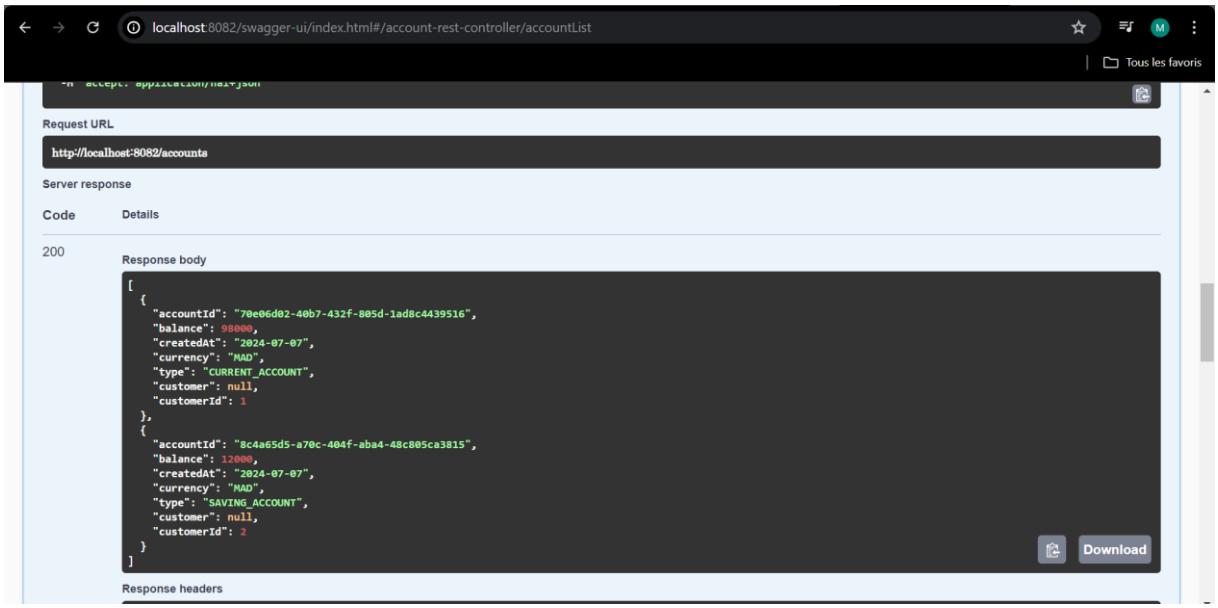

Impression élégante ☑

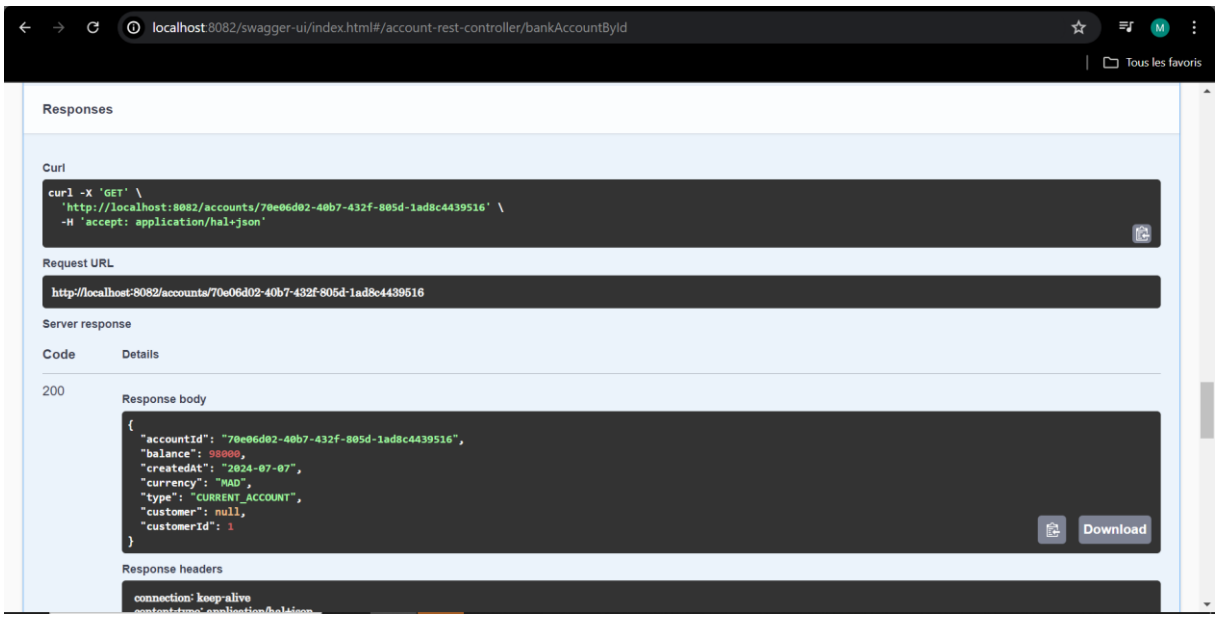

```
{
  "accountId": "e459ccc5-f535-4648-bd5a-889e79a1c444",
  "balance": 98000,
  "createdAt": "2024-07-07",
  "currency": "MAD",
  "type": "CURRENT_ACCOUNT",
  "customer": null,
  "customerId": 1
}
```

Il faut générer la documentation Swagger pour les web services et on ajoute la dépendance openai-api

```xml
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.6.0</version>
</dependency>
```

On teste



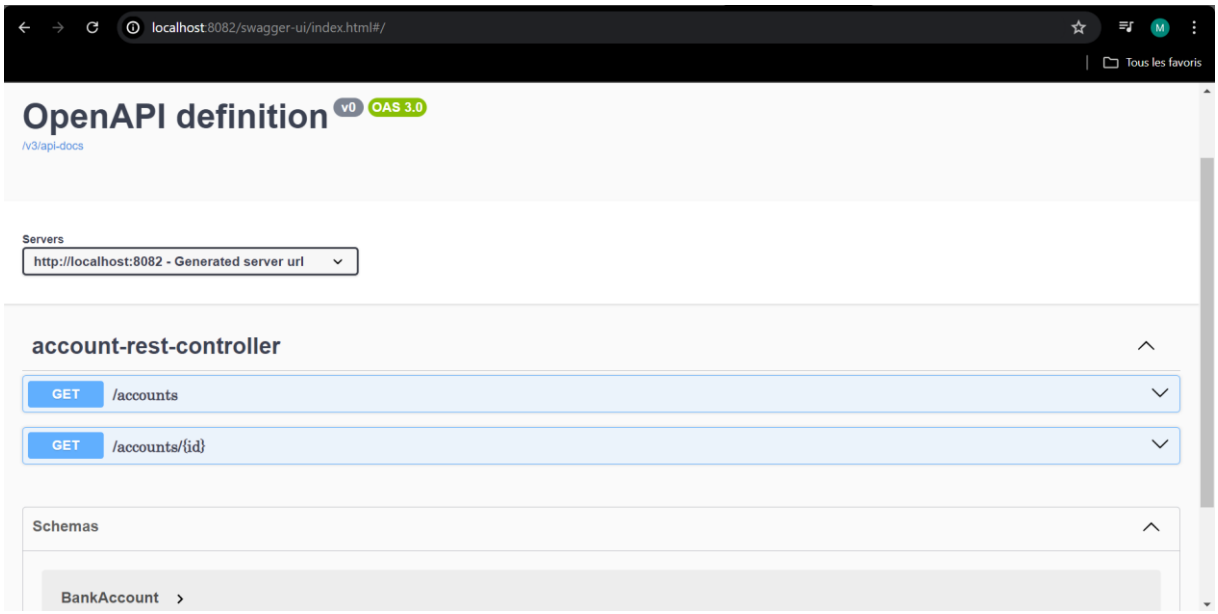


Si on veut voir la documentation swagger dans /v3/api-docs c'est l'interface du web service c'est ce document qui montre au client quelles sont les différentes méthodes qui existe dans le

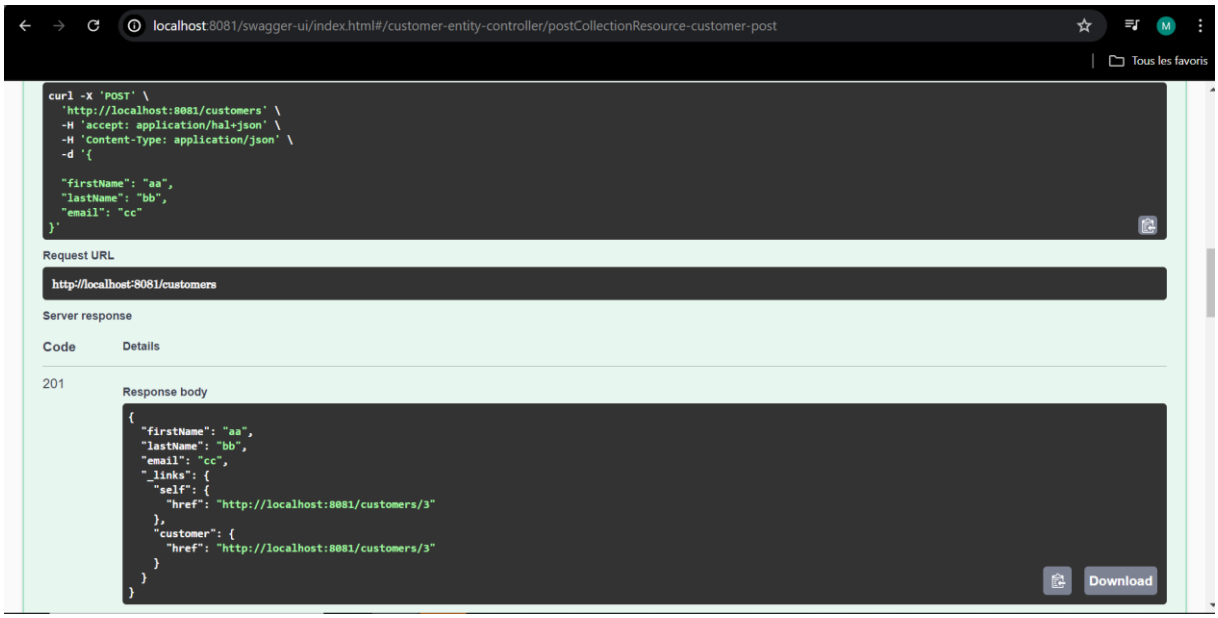
web services (dans les web services basées sur soap on utilise le WSDL mais pour d'autre qui ont basés sur rest on utilise openapi doc)

Les méthodes BankAccounts qui existe dans swagger ça va marcher car la dépendance rest n'est pas désactivé même si on n'a pas utilisé l'annotatioon @RestController

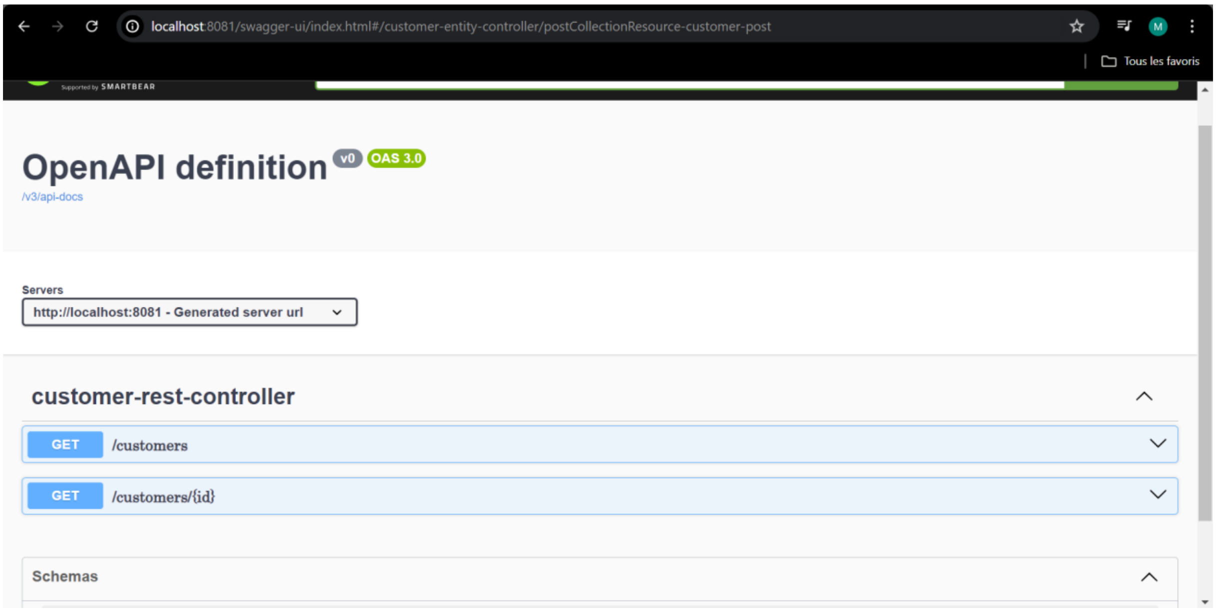
On désactive la dépendance rest



On ajoute la dépendance webmvc-ui au customer-service et on va ajouter l'annotation @RequestMapping pour éviter la confusion et on teste la méthode post et ça se passe bien



On désactive la dépendance rest pour customer-service et on lève l'annotation @RequestMapping



Maintenant, on va avoir comment mettre en place la gateway

Pour le gateway, il y a deux solutions la première c'est Zuul qui est basée sur les entrées/sotries bloquantes et spring cloud gateway basé sur les entrées/sorties non bloquantes, alors on va voir comment configurer ce gateway et pour configurer les routes de gateway, on peut le faire d'une manière statique ou dynamique et pour cette configuration on utilise le fichier application.yml pour celles qui sont complexes et application.properties pour les configurations simples.

Pour configurer les routes dans le fichier application.yml

```yaml
# application.yml
spring:
  cloud:
    gateway:
      routes:
        - id: r1
          uri: http://localhost:8081/
          predicates:
            - Path=/customers/**
        - id: r2
          uri: http://localhost:8082/
          predicates:
            - Path=/accounts/**
```

On peut aussi faire cette configuration dans application.properties mais c'est compliqué

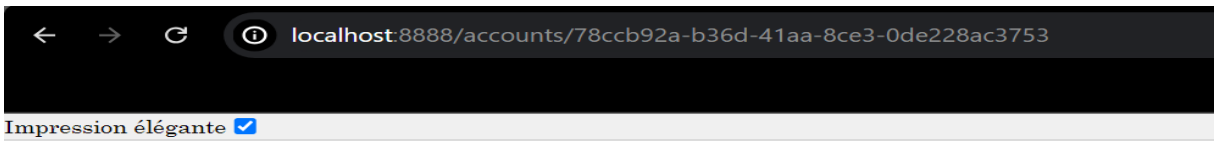
Et pour spécifier le nom de l'application et le port

```
        - id: r2
          uri: http://localhost:8082/
          predicates:
            - Path=/accounts/**
  application:
    name: gateway-service
server:
  port: 8888
```

On va démarrer ce gateway

Impression élégante


```
[
  {
    "accountId": "78ccb92a-b36d-41aa-8ce3-0de228ac3753",
    "balance": 98000,
    "createdAt": "2024-07-07",
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT",
    "customer": null,
    "customerId": 1
  },
  {
    "accountId": "baa7b014-f205-4b43-be6f-7e761e4a1e16",
    "balance": 12000,
    "createdAt": "2024-07-07",
    "currency": "MAD",
    "type": "SAVING_ACCOUNT",
    "customer": null,
    "customerId": 2
  }
]
```

Impression élégante

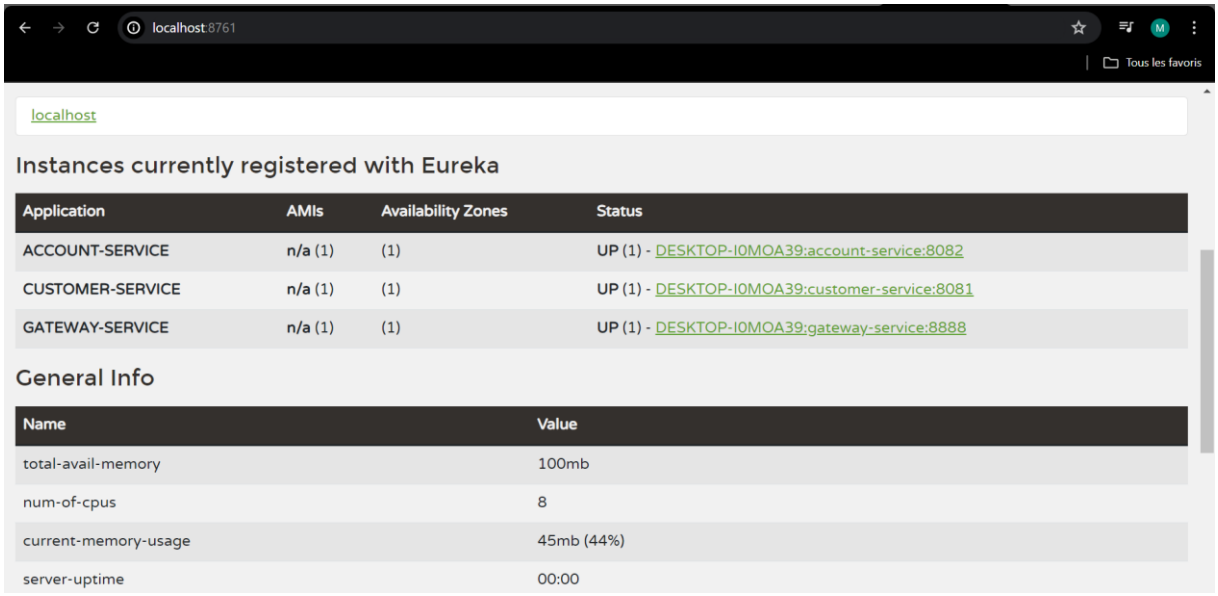

```
{
  "accountId": "78ccb92a-b36d-41aa-8ce3-0de228ac3753",
  "balance": 98000,
  "createdAt": "2024-07-07",
  "currency": "MAD",
  "type": "CURRENT_ACCOUNT",
  "customer": null,
  "customerId": 1
}
```

Ça c'est une configuration statique ou on suppose qu'on connait les adresses des micros services mais dans la pratique, on le connait pas donc la solution c'est de faire la configuration dynamique (utiliser Discovery service)

On va activer le discovery service en utilisant l'annotation EnableEurekaServer

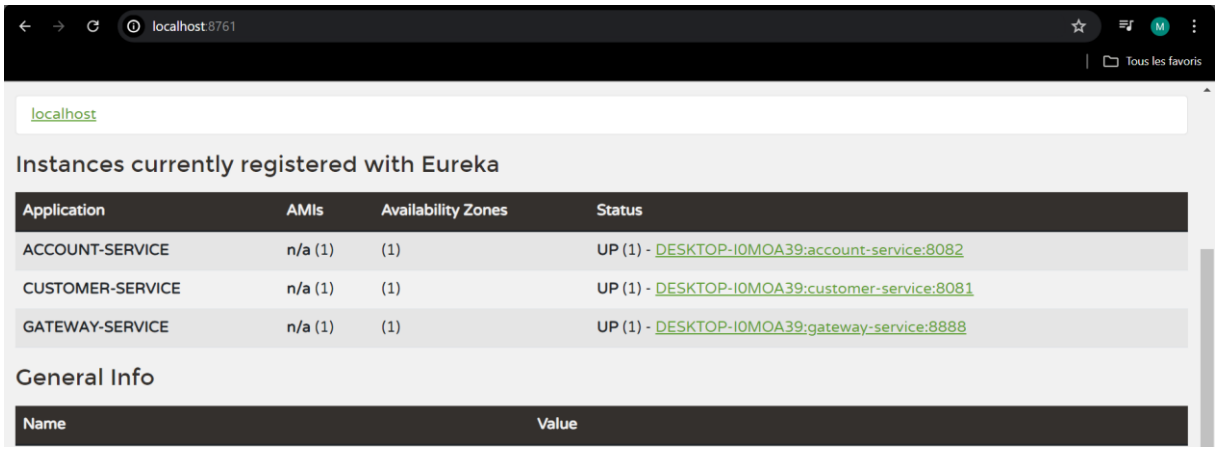
On configure application.properties

On démarre et on voie une interface eureka et pour afficher les services on va activer la propriété discovery et on redémarre chaque service



Mais les services s'enregistrent avec le nom de l'ordinateur et nous avons besoin de l'enregistrer avec leur adresses IPs et pour ce faire on ajoute 2 propriétés pour les 3 services et on redémarre

```
spring.application.name=account-service
spring.datasource.url=jdbc:h2:mem:account-db
spring.h2.console.enabled=true
server.port=8082
spring.cloud.discovery.enabled=true
spring.cloud.config.enabled=false
eureka.instance.prefer-ip-address=true
eureka.client.service-url.defaultZone=${DISCOVERY_SERVICE_URL:http://localhost:8761/eureka}
```



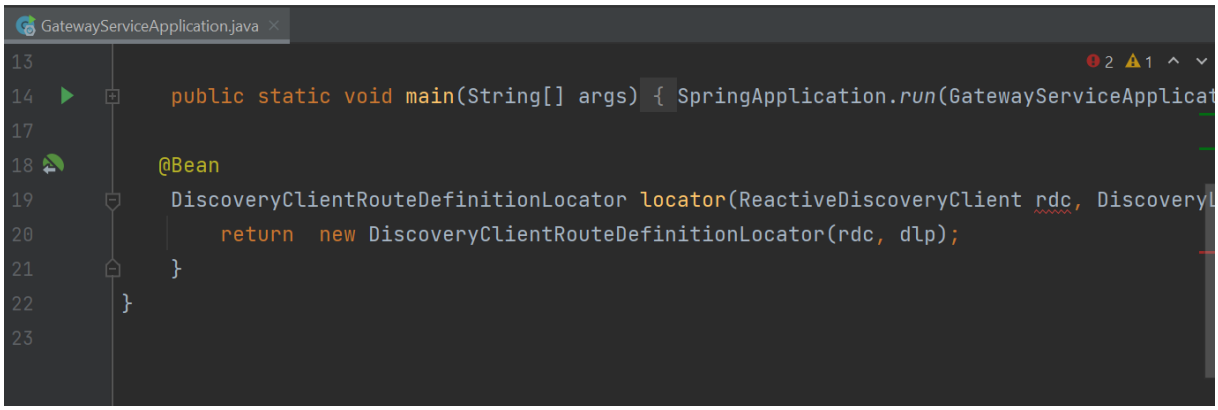
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

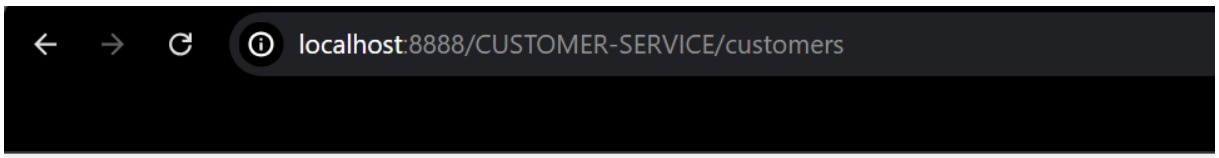
Sun Jul 07 17:46:02 CEST 2024
There was an unexpected error (type=Not Found, status=404).

Maintenant notre gateway, on va le configurer pour qu'il puisse chercher les adresses dans discovery

On fait une configuration dynamique



Après on démarre



Impression élégante ☑


```
[
  {
    "id": 1,
    "firstName": "Hassan",
    "lastName": "Elhoumi",
    "email": "hassan@gmail.com"
  },
  {
    "id": 2,
    "firstName": "Mohamed",
    "lastName": "Elhannaoui",
    "email": "hassan@gmail.com"
  }
]
```


Si on veut activer les services de actuator, on utilise une propriété

On démarre

localhost:8888/ACCOUNT-SERVICE/actuator/beans

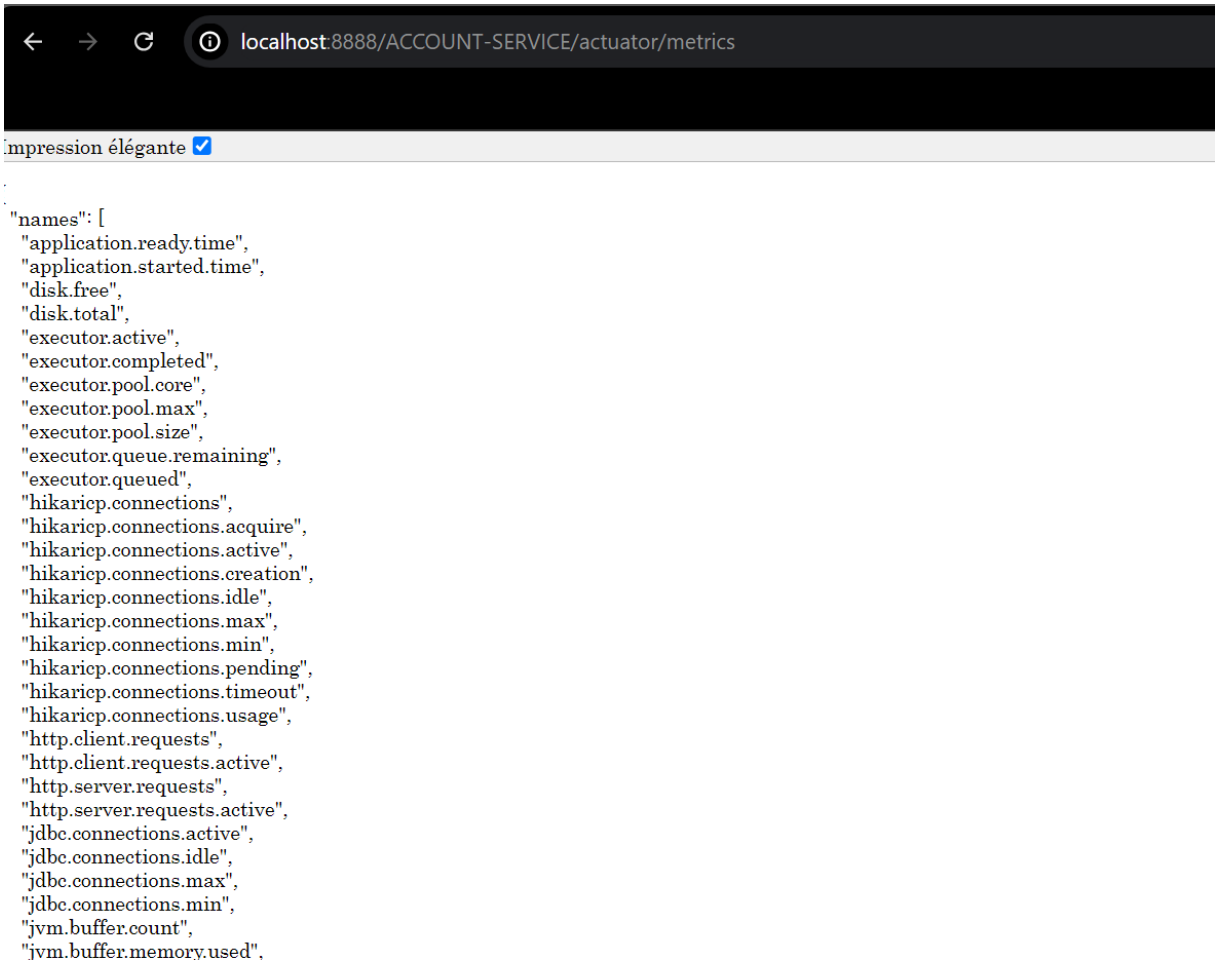
Impression élégante ☑


```
{
  "contexts": {
    "account-service": {
      "beans": {
        "spring.jpa-org.springframework.boot.autoconfigure.orm.jpa.JpaProperties": {
          "aliases": [],
          "scope": "singleton",
          "type": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties",
          "dependencies": []
        },
        "loadBalancerServiceInstanceCookieTransformer": {
          "aliases": [],
          "scope": "singleton",
          "type": "org.springframework.cloud.loadbalancer.core.LoadBalancerServiceInstanceCookieTransformer",
          "resource": "class path resource [org/springframework/cloud/loadbalancer/config/BlockingLoadBalancerClientAutoConfiguration.class]",
          "dependencies": [
            "org.springframework.cloud.loadbalancer.config.BlockingLoadBalancerClientAutoConfiguration",
            "loadBalancerClientFactory"
          ]
        },
        "discoveryClientHealthIndicator": {
          "aliases": [],
          "scope": "singleton",
          "type": "org.springframework.cloud.client.discovery.health.DiscoveryClientHealthIndicator",
          "resource": "class path resource [org/springframework/cloud/client/CommonsClientAutoConfiguration$DiscoveryLoadBalancerConfiguration.class]",
          "dependencies": [
            "org.springframework.cloud.client.CommonsClientAutoConfiguration$DiscoveryLoadBalancerConfiguration",
            "spring.cloud.discovery.client.health-indicator-org.springframework.cloud.client.discovery.health.DiscoveryClientHealthIndicatorProperties"
          ]
        },
        "org.springframework.cloud.netflix.eureka.EurekaClientAutoConfiguration$RefreshableEurekaClientConfiguration": {
```


localhost:8888/ACCOUNT-SERVICE/actuator/health

Impression élégante ☐


```
{"status":"UP"}
```



Si on veut consulter la liste des comptes

localhost:8888/ACCOUNT-SERVICE/accounts

Impression élégante


```
[
  {
    "accountId": "0180c06d-39e3-40b4-9a35-44c0a8e1bacf",
    "balance": 98000,
    "createdAt": "2024-07-08",
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT",
    "customer": null,
    "customerId": 1
  },
  {
    "accountId": "f6137819-0667-40f7-8385-8f44cb50ca70",
    "balance": 12000,
    "createdAt": "2024-07-08",
    "currency": "MAD",
    "type": "SAVING_ACCOUNT",
    "customer": null,
    "customerId": 2
  }
]
```


localhost:8888/ACCOUNT-SERVICE/accounts/0180c06d-39e3-40b4-9a35-44c0a8e1bacf

Impression élégante


```
{
  "accountId": "0180c06d-39e3-40b4-9a35-44c0a8e1bacf",
  "balance": 98000,
  "createdAt": "2024-07-08",
  "currency": "MAD",
  "type": "CURRENT_ACCOUNT",
  "customer": null,
  "customerId": 1
}
```


Maintenant, on a l'id de client mais aussi on a besoin de connaitre son nom et pour ce faire nous aurons besoin d'envoyer une requête vers customer-service du coté account-service

On va utiliser le framework open Feign qui va se charger de rest c'est le plus simple et lui qui permettre de faire facilement les microservices alors pour utiliser open feign on a besoin d'ajouter une dépendance dans account-service

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
<dependency>
```

Pour utiliser open Fiegn, on va crée un package clients contient une interface CustomerRestClient et on veut que cette interface soit open feign c'est pour cela on utilise l'annotation @FeignClient(name = CUSTOMER-SERVICE)

CustomerRestClient.java

```java
package net.illa.accountservice.clients;

import net.illa.accountservice.model.Customer;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;

import java.util.List;
@FeignClient(name= "CUSTOMER-SERVICE")
public interface CustomerRestClient {
    @GetMapping("/customers/{id}")
    Customer findCustomerById(Long id);
    @GetMapping("/customers")
    List<Customer> allCustomers();
}
```

On va tester, pour ça on va déclarer CustomerRestClient dans AccountRestController

```java
@RestController
public class AccountRestController {
    private BankAccountRepository accountRepository;
    private CustomerRestClient customerRestClient;

    public AccountRestController(BankAccountRepository accountRepository, CustomerRestClier
        this.accountRepository=accountRepository;
        this.customerRestClient = customerRestClient;
    }
    @GetMapping("/accounts")
    public List<BankAccount>  accountList() { return accountRepository.findAll(); }
    @GetMapping("/accounts/{id}")
    public BankAccount bankAccountById(@PathVariable String id){
       BankAccount bankAccount= accountRepository.findById(id).get();
       Customer customer=customerRestClient.findCustomerById(bankAccount.getCustomerId());
       bankAccount.setCustomer(customer);
       return bankAccount;

    }
}
```

Et il faut activer openFiegn si on veut l'utiliser

```java
package net.illa.accountservice;

import ...

@SpringBootApplication
@EnableFeignClients
public class AccountServiceApplication {

    public static void main(String[] args) { SpringApplication.run(AccountServiceApplicat
    @Bean
    CommandLineRunner commandLineRunner(BankAccountRepository accountRepository) {
        return args ->{
            BankAccount bankAccount1=new BankAccount().builder()
                    .accountId(UUID.randomUUID().toString())
                    .currency("MAD")
                    .balance(98000)
                    .createdAt(LocalDate.now())
                    .type(AccountType.CURRENT_ACCOUNT)
                    .customerId(Long.valueOf(1))
                    .build();
            BankAccount bankAccount2=new BankAccount().builder()
```

On redémarre

```
▼<List>
  ▼<item>
      <accountId>8b1c4ede-2a03-4d9d-90cd-4e20a8d6782b</accountId>
      <balance>98000.0</balance>
      <createdAt>2024-07-09</createdAt>
      <currency>MAD</currency>
      <type>CURRENT_ACCOUNT</type>
      <customer/>
      <customerId>1</customerId>
    </item>
  ▼<item>
      <accountId>fcf7331c-df36-4c7b-8b64-9e6b475a3d24</accountId>
      <balance>12000.0</balance>
      <createdAt>2024-07-09</createdAt>
      <currency>MAD</currency>
      <type>SAVING_ACCOUNT</type>
      <customer/>
      <customerId>2</customerId>
    </item>
</List>
```


Alors openFiegn permet de créer un client rest déclaratif

Maintenant, on a besoin d'ajouter la dépendance reselience4j

```
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
```

Après on va vers CustomerRestClient et on va utiliser l'annotation @CircuitBreaker qui est un handler d'exception qui permet de gérer un circuit breaker avec les modes open / half …

```java
import org.springframework.web.bind.annotation.PathVariable;

import java.util.List;
@FeignClient(name= "CUSTOMER-SERVICE")
public interface CustomerRestClient {
    @GetMapping("/customers/{id}")
    @CircuitBreaker(name="customerService", fallbackMethod = "getDefaultCustomer")
    Customer findCustomerById(@PathVariable Long id);
    @GetMapping("/customers")
    List<Customer> allCustomers();

    default Customer getDefaultCustomer(Long id) {
        Customer customer=new Customer();
        customer.setId(id);
        customer.setFirstname("Not Vailable");
        customer.setLastName("Not Vailable");
        customer.setEmail("Not Vailable");
        return customer;
    }
}
```

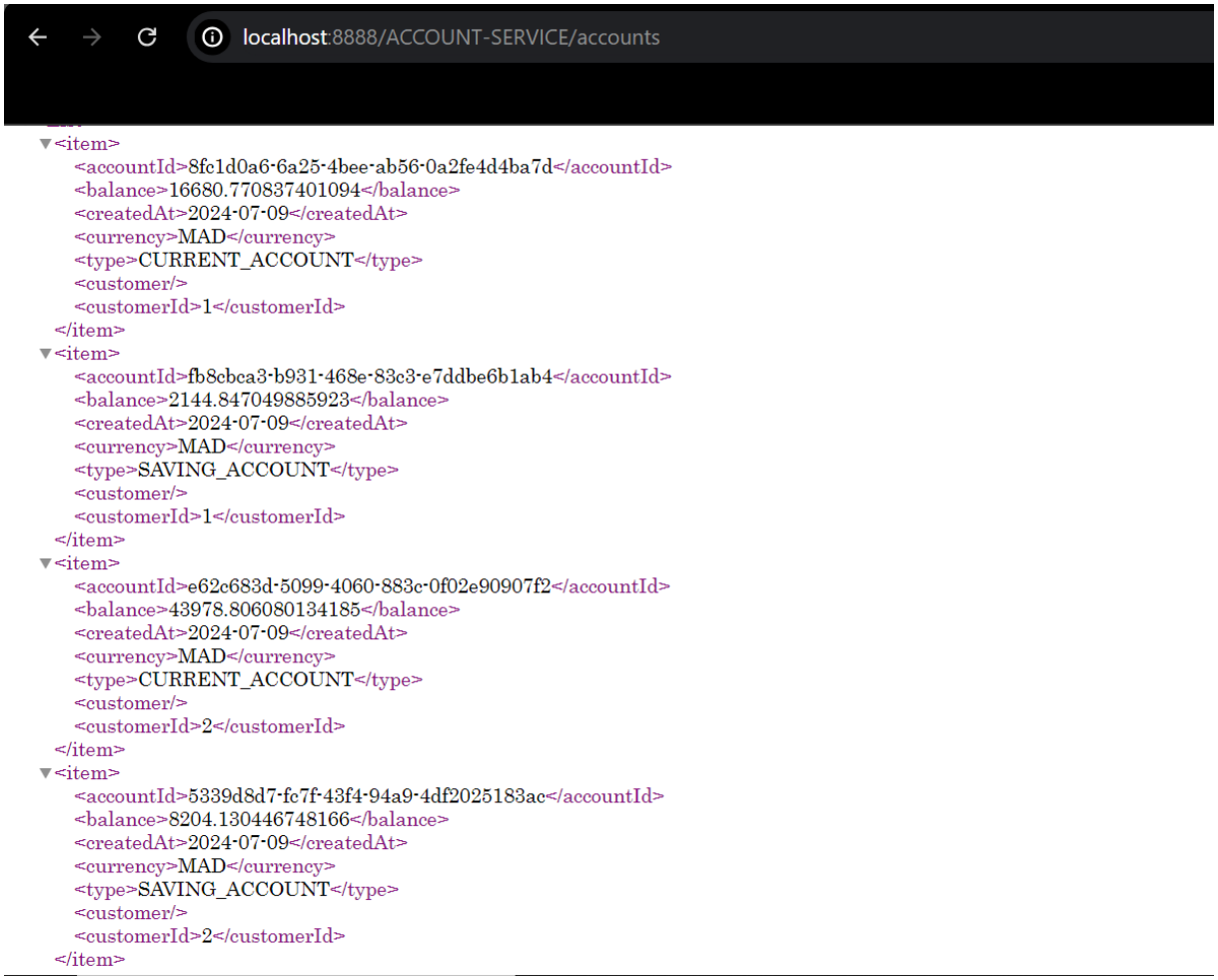
On utilise l'annotation aussi sur allCustomers

```
    Customer findCustomerById(@PathVariable Long id);
    @GetMapping("/customers")
    @CircuitBreaker(name="customerService", fallbackMethod = "getAllCustomers")
    List<Customer> allCustomers();

    default Customer getDefaultCustomer(Long id) {
        Customer customer=new Customer();
        customer.setId(id);
        customer.setFirstname("Not Vailable");
        customer.setLastName("Not Vailable");
        customer.setEmail("Not Vailable");
        return customer;
    }
    default List<Customer> getAllCustomers(Exception exception) {
        return List.of();

    }
}
```

Si on veut consulter la liste des customers a partir de l'application account-service

```
    }
    @Bean
    CommandLineRunner commandLineRunner(BankAccountRepository accountRepository, Custom
        return args ->{
            customerRestClient.allCustomers().forEach(c ->{
                BankAccount bankAccount1=new BankAccount().builder()
                        .accountId(UUID.randomUUID().toString())
                        .currency("MAD")
                        .balance(Math.random()*98000)
                        .createdAt(LocalDate.now())
                        .type(AccountType.CURRENT_ACCOUNT)
                        .customerId(c.getId())
                        .build();
                BankAccount bankAccount2=new BankAccount().builder()
                        .accountId(UUID.randomUUID().toString())
                        .currency("MAD")
                        .balance(Math.random()*12000)
                        .createdAt(LocalDate.now())
                        .type(AccountType.SAVING_ACCOUNT)
                        .customerId(c.getId())
                        .build();
```

On redémarre :



Partie 2 :

Cette partie explique comment continuer l'implémentation de l'architecture microservices avec Spring Cloud, en mettant en place un service de configuration centralisé pour gérer les configurations de manière efficace, et en utilisant des outils comme Spring Data et OpenFeign pour faciliter la communication entre microservices.