



*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

The screenshot shows the Spring Initializr interface. Under 'Project', 'Maven' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', '3.3.1' is selected. Under 'Project Metadata', 'Group' is set to 'org.sid'. On the right, 'Dependencies' are configured: 'Lombok' and 'Spring Web' (WEB) are selected, while 'H2 Database' (SQL) is deselected. Buttons at the bottom include 'GENERATE' (CTRL + F), 'EXPLORE' (CTRL + SPACE), and 'SHARE...'.

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

```

1 package org.sid.bank_account_service.enums;
2
3 public enum AccountType {
4     CURRENT_ACCOUNT, SAVING_ACCOUNT;
5 }

```

```

BankAccount.java ×
1 package org.sid.bank_account_service.entities;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Builder;
5 import lombok.Data;
6 import lombok.NoArgsConstructor;
7 import org.sid.bank_account_service.enums.AccountType;
8
9 import javax.persistence.Entity;
10 import javax.persistence.Id;
11 import java.util.Date;
12
13 @Entity
14 @Data @NoArgsConstructor @AllArgsConstructor @Builder
15 public class BankAccount {
16     @Id
17     private String id;
18     private Date createdAt;
19     private double balance;
20     private String currency;
21     private AccountType type;
22 }

```

Endpoints Spring

On va créer une interface BankAccountRepository

```

BankAccountRepository.java ×
1 package org.sid.bank_account_service.repositories;
2
3 import org.sid.bank_account_service.entities.BankAccount;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 public interface BankAccountRepository extends JpaRepository<BankAccount, String> {
7 }
8

```

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

```

17  public static void main(String[] args) {
18
19
20
21     @Bean
22     CommandLineRunner start(BankAccountRepository bankAccountRepository) {
23         return args -> {
24             for (int i=0; i<10; i++) {
25                 BankAccount bankAccount = BankAccount.builder()
26                     .id(UUID.randomUUID().toString())
27                     .type(Math.random()>0.5? AccountType.CURRENT_ACCOUNT: AccountType.SAVING_ACCOUNT)
28                     .balance(10000+Math.random()*90000)
29                     .createdAt(new Date())
30                     .currency("MAD")
31                     .build();
32                 bankAccountRepository.save(bankAccount);
33             }
34         };
35     };
36 }
37
38 }
39

```

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

```

1 spring.application.name=bank-account-service
2 spring.datasource.url=jdbc:h2:mem:account-db
3 spring.h2.console.enabled=true
4 server.port=8081
5
6

```

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

The screenshot shows the H2 Database Console interface. The left sidebar lists the database schema: 'jdbc:h2:mem:account-db' containing 'BANK\_ACCOUNT', 'INFORMATION\_SCHEMA', and 'Users'. A note at the bottom says 'H2 2.1.214 (2022-06-13)'. The main area displays the results of the SQL query 'SELECT \* FROM BANK\_ACCOUNT;'. The table has columns: ID, BALANCE, CREATED\_AT, CURRENCY, and TYPE. There are 10 rows of data.

ID	BALANCE	CREATED_AT	CURRENCY	TYPE
5f04216f-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
c738ff00-ed9b-4373-b6ee-285e81804b92	81417.09278723247	2024-07-05 00:54:54.214	MAD	1
b06ce004-4ff5-4e1f-8a08-22b632f28ac5	68872.47856856225	2024-07-05 00:54:54.215	MAD	0
37a37147-03a9-44f7-8f41-3a37b41a0cd	83932.46570400729	2024-07-05 00:54:54.216	MAD	0
3a44aa4f1-ca56-4c0a-9713-6f339d45078c	66649.9494609258	2024-07-05 00:54:54.218	MAD	0
f0a7735e-d805-45d5-b4b7-39907b74723	36927.4176446482	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
3b569cd4-bc88-4e79-9d6b-ea83a1f18ce9	71869.59586760412	2024-07-05 00:54:54.222	MAD	0
e49cad69-3720-4d67-aa16-494ecccc3e5	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)

```

 9 import javax.persistence.Entity;
10 import javax.persistence.Enumerated;
11 import javax.persistence.Id;
12 import java.util.Date;
13 import javax.persistence.*;
14 import org.bson.types.ObjectId;
15 import org.springframework.data.annotation.NoArgsConstructor;
16 import org.springframework.data.annotation.AllArgsConstructor;
17 import org.springframework.data.annotation.Builder;
18 import org.springframework.data.annotation.Id;
19 import org.springframework.data.annotation.TypeReference;
20 import org.springframework.data.annotation.TypeReference;
21 import org.springframework.data.annotation.TypeReference;
22 import org.springframework.data.annotation.TypeReference;
23 import org.springframework.data.annotation.TypeReference;
24 import org.springframework.data.annotation.TypeReference;
25

```

The code shows the `BankAccount` entity definition. It includes annotations for persistence (Entity, Id, Date), validation (NoArgsConstructor, AllArgsConstructor, Builder), and type mapping (String, Date, Double, String, EnumType.STRING). The `id` field is annotated with `@Id` and `ObjectId`.

On s'exécute

The screenshot shows the H2 Database Console interface. At the top, there's a toolbar with various icons and dropdown menus. Below the toolbar, a sidebar lists database objects: 'jdbc:h2:mem:account-db', 'BANK\_ACCOUNT', 'INFORMATION\_SCHEMA', and 'Users'. A message at the bottom of the sidebar indicates the version: 'H2 2.1.214 (2022-06-13)'. The main area contains a SQL statement: 'SELECT \* FROM BANK\_ACCOUNT'. Below the statement is a table with 10 rows of data, each representing a bank account with columns: ID, BALANCE, CREATED\_AT, CURRENCY, and TYPE. The table shows various account types like SAVING\_ACCOUNT and CURRENT\_ACCOUNT.

ID	BALANCE	CREATED_AT	CURRENCY	TYPE
36c7e36a-c548-4e9d-9a35-76db170dfc7c	17373.919295452542	2024-07-05 01:41:43.706	MAD	SAVING_ACCOUNT
45fc184f-facc-48fa-97be-b6c98588974a	71267.81051011206	2024-07-05 01:41:43.764	MAD	CURRENT_ACCOUNT
5f313b84-4b12-44ff-85d6-13fe5c206bb9f	33811.67477750637	2024-07-05 01:41:43.766	MAD	SAVING_ACCOUNT
8cd6e62e-be64-4668-9a9b-c122f26a675	69217.83776369688	2024-07-05 01:41:43.767	MAD	CURRENT_ACCOUNT
832cc29-941b-488c-8d02-4e63a10125d7	15239.558878368982	2024-07-05 01:41:43.768	MAD	SAVING_ACCOUNT
85503df1-323d-44bf-8454-b19922923479	43166.66870324877	2024-07-05 01:41:43.769	MAD	SAVING_ACCOUNT
8afb4605-c53c-4691-b924-c416a3596b0	98679.17975454326	2024-07-05 01:41:43.77	MAD	CURRENT_ACCOUNT
5c678c8c-0e49-4a45-82c2-99e6a63d839a	98625.52337217783	2024-07-05 01:41:43.771	MAD	SAVING_ACCOUNT
e6eeb5b9-c29f-4a7c-b306-e6ef7b41da2	87543.28575501635	2024-07-05 01:41:43.773	MAD	CURRENT_ACCOUNT
98efabd-417b-4899-9cbc-ccf07d9e4012	57127.04036808162	2024-07-05 01:41:43.774	MAD	CURRENT_ACCOUNT

(10 rows, 8 ms)

Edit

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 déprécié 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 paramètre qui va être récupéré à partir de path

```

12
13  @RestController
14  public class AccountRestController {
15      @Autowired
16      private BankAccountRepository bankAccountRepository;
17
18      public AccountRestController(BankAccountRepository bankAccountRepository){
19          this.bankAccountRepository= bankAccountRepository;
20      }
21
22      @GetMapping("/bankAccounts")
23      public List<BankAccount> bankAccounts() {
24          return bankAccountRepository.findAll();
25      }
26
27      @GetMapping("/bankAccount/{id}")
28      public BankAccount bankAccount(@PathVariable String id) {
29          return bankAccountRepository.findById(id)
30              .orElseThrow(() ->new RuntimeException(String.format("Account %s not found")));
31      }

```

## On teste l'application

```

Impression élégante 
[  

  {  

    "id": "68ffa588-7b84-4e1e-acc7-5d3f13c381cb",  

    "createdAt": "2024-07-05T01:01:59.985+00:00",  

    "balance": 15617.7745048382,  

    "currency": "MAD",  

    "type": "SAVING_ACCOUNT"  

  },  

  {  

    "id": "f60a4863-5d4a-4e86-a228-7b611a7483b5",  

    "createdAt": "2024-07-05T01:02:00.045+00:00",  

    "balance": 72748.4369301688,  

    "currency": "MAD",  

    "type": "SAVING_ACCOUNT"  

  },  

  {  

    "id": "2eb46b54-97c0-48b2-a02c-c282c2116dc8",  

    "createdAt": "2024-07-05T01:02:00.047+00:00",  

    "balance": 41980.9684411047,  

    "currency": "MAD",  

    "type": "CURRENT_ACCOUNT"  

  },  

  {  

    "id": "440d9faf-4728-4564-b1a7-e22842be170a",  

    "createdAt": "2024-07-05T01:02:00.048+00:00",  

    "balance": 52325.5923144795,  

    "currency": "MAD",  

    "type": "CURRENT_ACCOUNT"  

  },  

  {  

    "id": "978d5818-ec33-4560-892e-1da840e2c313",  

    "createdAt": "2024-07-05T01:02:00.049+00:00",  

    "balance": 67678.4269024561,  

    "currency": "MAD",  

    "type": "CURRENT_ACCOUNT"
  }
]

```

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

```

  AccountRestController.java

32     @PostMapping("/bankAccounts")
33     public BankAccount save(@RequestBody BankAccount bankAccount) {
34         if(bankAccount.getId()==null) bankAccount.setId(UUID.randomUUID().toString());
35         return bankAccountRepository.save(bankAccount);
36     }
37     @PutMapping("/bankAccounts/{id}")
38     public BankAccount update(@PathVariable String id, @RequestBody BankAccount bankAccou
39         BankAccount account=bankAccountRepository.findById(id).orElseThrow();
40         if(bankAccount.getBalance() != null) account.setBalance(bankAccount.getBalance());
41         if(bankAccount.getCreatedAt() != null) account.setCreatedAt(new Date());
42         if(bankAccount.getType() != null) account.setType(bankAccount.getType());
43         if(bankAccount.getCurrency() != null) account.setCurrency(bankAccount.getCurrency());
44         return bankAccountRepository.save(account);
45     }
46     @DeleteMapping("/bankAccounts/{id}")
47     public void deleteAccount(@PathVariable String id) {
48         bankAccountRepository.deleteById(id);
49     }

```

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

The screenshot shows the Postman interface with the following details:

- Collection:** My first collection
- Folders:** First folder inside collection, Second folder inside collection
- Requests:**
  - First folder: GET, POST, GET
  - Second folder: GET, GET
- Selected Request:** GET http://localhost:8081/bankAccounts
- Response:**
  - Status: 200 OK
  - Time: 245 ms
  - Size: 1.72 KB
  - Body (Pretty):

```

1 [
2   {
3     "id": "1eb51344-5535-4843-b377-82ea2e0c5cab",
4     "createdAt": "2024-07-05T11:30:52.796+00:00",
5     "balance": 45474.29923200959,
6     "currency": "MAD",
7     "type": "CURRENT_ACCOUNT"
8   },
9   {
10    "id": "345db673-7bd3-48bf-ad01-641f722fbe65",
11    "createdAt": "2024-07-05T11:30:52.890+00:00",
12    "balance": 48642.3134237258,
13    "currency": "MAD",
14    "type": "CURRENT_ACCOUNT"

```

The screenshot shows the Postman interface with a successful GET request to `http://localhost:8081/bankAccounts/f8af90ee-a4a1-46e7-9484-35698d72ef5e`. The response body is:

```

1 {
2   "id": "f8af90ee-a4a1-46e7-9484-35698d72ef5e",
3   "createdAt": "2024-07-05T11:40:16.384+00:00",
4   "balance": 27149.75605046427,
5   "currency": "MAD",
6   "type": "CURRENT_ACCOUNT"
7 }
  
```

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

```

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

On teste

The screenshot shows the Postman interface with a POST request to `http://localhost:8081/bankAccounts`. The request body is:

```

1 {
2   "balance": 8000,
3   "type": "SAVING_ACCOUNT",
4   "currency": "EUR"
5 }
  
```

The response body is:

```

1 {
2   "id": "58b18697-cc0a-46b0-b575-40a8ed096352",
3   "createdAt": null,
4   "balance": 8000.0,
5   "currency": "EUR",
6   "type": "SAVING_ACCOUNT"
7 }
  
```

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

The screenshot shows the Postman interface. A collection named "Your collection" is selected. A PUT request is being made to the URL `http://localhost:8081/bankAccounts/85533cf1-cb54-4425-8e65-70c6468e8189`. The "Body" tab is selected, showing the JSON payload:

```

1 {
2   ...
3   "currency": "USD"
4 }

```

The response status is 200 OK, with a response time of 5 ms and a size of 284 B. The response body is:

```

1 {
2   "id": "85533cf1-cb54-4425-8e65-70c6468e8189",
3   "createdAt": null,
4   "balance": 8000.0,
5   "currency": "USD",
6   "type": "SAVING_ACCOUNT"
7 }

```

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

The screenshot shows a code editor displaying the `pom.xml` file for a Spring Boot application. The dependencies section includes:

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-graphql</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springdoc/springdoc-openapi-ui -->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
    <version>1.6.11</version>
</dependency>

```

Et la documentation open ai

```

{
  "openapi": "3.0.1",
  "info": {
    "title": "OpenAPI definition",
    "version": "v0"
  },
  "servers": [
    {
      "url": "http://localhost:8081",
      "description": "Generated server url"
    }
  ],
  "paths": {
    "/bankAccounts/{id)": {
      "get": {
        "tags": [
          "account-rest-controller"
        ],
        "operationId": "bankAccount",
        "parameters": [
          {
            "name": "id",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "OK",
            "content": {
              "application/json": {
                "type": "array",
                "items": {
                  "type": "object",
                  "properties": {
                    "id": "string",
                    "balance": "number",
                    "currency": "string",
                    "type": "string"
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
  
```

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

Curl

```
curl -X 'GET' \
  'http://localhost:8081/bankAccounts' \
  -H 'accept: */*'
```

Request URL

<http://localhost:8081/bankAccounts>

Server response

Code	Details
200	<p>Response body</p> <pre>[   {     "id": "224ef38f-4815-43c6-8510-5b1061299983",     "createdat": "2024-07-05T12:49:34.536+00:00",     "balance": 91042.64515312423,     "currency": "MAD",     "type": "SAVING_ACCOUNT"   },   {     "id": "8c709c8f-b334-4811-a0aa-86625fe356e5",     "createdat": "2024-07-05T12:49:34.620+00:00",     "balance": 89609.8826239631,     "currency": "MAD",     "type": "CURRENT_ACCOUNT"   },   {     "id": "79d13ef97-2419-4013-b46b-faa7dc88bcf6"   } ]</pre>

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

**Curl**

```
curl -X 'GET' \
'http://localhost:8081/bankAccounts/224ef38f-4815-43c6-8510-5b1061299983' \
-H 'accept: */*'
```

**Request URL**

`http://localhost:8081/bankAccounts/224ef38f-4815-43c6-8510-5b1061299983`

**Server response**

Code	Details
200	<b>Response body</b> <pre>{   "id": "224ef38f-4815-43c6-8510-5b1061299983",   "createdAt": "2024-07-05T12:49:34.536+00:00",   "balance": 91042.64515312423,   "currency": "MAD",   "type": "SAVING_ACCOUNT" }</pre> <div style="display: flex; justify-content: space-between;"> <span>Copy</span> <span>Download</span> </div> <b>Response headers</b> <pre>connection: keep-alive content-type: application/json date: Fri, 06 Jul 2024 13:01:04 GMT keep-alive: timeout=60 transfer-encoding: chunked</pre>

**Responses**

Si on veut ajouter un compte on fait la méthode post

**Request URL**

`http://localhost:8081/bank/accounts`

**Server response**

Code	Details
200	<b>Response body</b> <pre>{   "id": "CC1",   "createdAt": "2024-07-05T13:06:38.000Z",   "balance": 700,   "currency": "MAD",   "type": "CURRENT_ACCOUNT" }</pre> <div style="display: flex; justify-content: space-between;"> <span>Copy</span> <span>Download</span> </div> <b>Response headers</b> <pre>connection: keep-alive content-type: application/json date: Fri, 06 Jul 2024 13:08:03 GMT keep-alive: timeout=60 transfer-encoding: chunked</pre>

**Responses**

Code	Description	Links
------	-------------	-------

On vérifie

localhost:8081/bankAccounts

TP7POO

Impression élégante

```
{
  "id": "79d13f97-2419-4013-b46b-faa7dcc8becf",
  "createdAt": "2024-07-05T12:49:34.622+00:00",
  "balance": 38913.01799712295,
  "currency": "MAD",
  "type": "CURRENT_ACCOUNT"
},
{
  "id": "04377800-76a7-423a-8396-e054b29a1e1e",
  "createdAt": "2024-07-05T12:49:34.623+00:00",
  "balance": 64839.72772681827,
  "currency": "MAD"
}
]
```

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

The screenshot shows the Postman interface with the following details:

- API Network:** GET http://localhost:8081/bankAccounts / save
- Method:** POST
- Body:** (empty)
- Response:**

```

1b
16
17
18
19
20
21
22
23
24
25
26
27
  {
    "id": "79d13f97-2419-4013-b46b-faa7dcc8becf",
    "createdAt": "2024-07-05T12:49:34.622+00:00",
    "balance": 38913.01799712295,
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT"
  },
  {
    "id": "04377800-76a7-423a-8396-e054b29a1e1e",
    "createdAt": "2024-07-05T12:49:34.623+00:00",
    "balance": 64839.72772681827,
    "currency": "MAD"
  }

```

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

```

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

On ajoute à 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 ×
1 package org.sid.bank_account_service.repositories;
2
3 import org.sid.bank_account_service.entities.BankAccount;
4 import org.springframework.data.jpa.repository.JpaRepository;
5 import org.springframework.data.rest.core.annotation.RepositoryRestResource;
6
7 @RepositoryRestResource
8 public interface BankAccountRepository extends JpaRepository<BankAccount, String> {
9 }
10
```

Et on ajoute `@RequestMapping(" /api")` c'est-à-dire pour accéder à ce web service il faut écrire `/api/bankAccounts`

```

AccountRestController.java ×
10 import java.util.List;
11 import java.util.UUID;
12
13 @RestController
14 @RequestMapping("/api")
15 public class AccountRestController {
16     @Autowired
17     private BankAccountRepository bankAccountRepository;
18
19     public AccountRestController(BankAccountRepository bankAccountRepository){
20         this.bankAccountRepository= bankAccountRepository;
21     }
22     @GetMapping("/bankAccounts")
23     public List<BankAccount> bankAccounts() { return bankAccountRepository.findAll(); }
24
25     @GetMapping("/bankAccounts/{id}")
26     public BankAccount bankAccount(@PathVariable String id) {
27         return bankAccountRepository.findById(id);
28     }
29 }
```

On exécute

Impression élégante □

```
{
  "embedded": [
    {
      "bankAccounts": [
        {
          "createdAt": "2024-07-05T13:39:22.656+00:00",
          "balance": 75665.83042456249,
          "currency": "MAD",
          "type": "CURRENT_ACCOUNT",
          "links": {
            "self": {
              "href": "http://localhost:8081/bankAccounts/c32535d6-5e40-4503-8b07-378fba063dbc"
            },
            "bankAccount": {
              "href": "http://localhost:8081/bankAccounts/c32535d6-5e40-4503-8b07-378fba063dbc"
            }
          }
        },
        {
          "createdAt": "2024-07-05T13:39:22.740+00:00",
          "balance": 91990.86680206259,
          "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": {
            "self": {
              "href": "http://localhost:8081/bankAccounts/37953bb2-9fa4-4ce1-b981-f4a4fc1881f0"
            },
            "bankAccount": {
              "href": "http://localhost:8081/bankAccounts/37953bb2-9fa4-4ce1-b981-f4a4fc1881f0"
            }
          }
        }
      ]
    }
  ]
}
```

Impression élégante ✓

```
[
  {
    "id": "c32535d6-5e40-4503-8b07-378fba063dbc",
    "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": "86aef837-e411-4a31-b0e3-0fb9167fb1",
    "createdAt": "2024-07-05T13:39:22.742+00:00",
    "balance": 19825.262626052,
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT"
  },
  {
    "id": "851bc737-8688-467f-99d7-142d5f924e5f",
    "createdAt": "2024-07-05T13:39:22.743+00:00",
    "balance": 62106.2779869433,
    "currency": "MAD",
    "type": "SAVING_ACCOUNT"
  },
  {
    "id": "ed960e21-085f-4b27-baa9-c84eb0ae1bff",
    "createdAt": "2024-07-05T13:39:22.744+00:00",
    "balance": 12046.2375955852,
    "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

```

1 package org.sid.bank_account_service.repositories;
2
3 import org.sid.bank_account_service.entities.BankAccount;
4 import org.sid.bank_account_service.enums.AccountType;
5 import org.springframework.data.jpa.repository.JpaRepository;
6 import org.springframework.data.rest.core.annotation.RepositoryRestResource;
7
8 import java.util.List;
9
10 @RepositoryRestResource
11 public interface BankAccountRepository extends JpaRepository<BankAccount, String> {
12
13     List<BankAccount> findByType(AccountType type) ;
14 }
15

```

On accède directement à la méthode

```

Impression élégante □
{
    "balance": 96022.22830500496,
    "currency": "MAD",
    "type": "SAVING_ACCOUNT",
    "links": {
        "self": {
            "href": "http://localhost:8081/bankAccounts/0b93e61d-1870-4e73-b08d-19b157afed64"
        },
        "bankAccount": {
            "href": "http://localhost:8081/bankAccounts/0b93e61d-1870-4e73-b08d-19b157afed64"
        }
    },
    "createdAt": "2024-07-05T15:48:04.606+00:00",
    "balance": 43659.993679986066,
    "currency": "MAD",
    "type": "SAVING_ACCOUNT",
    "links": {
        "self": {
            "href": "http://localhost:8081/bankAccounts/dfe8919e-00f6-4393-b042-3c1694b527fc"
        },
        "bankAccount": {
            "href": "http://localhost:8081/bankAccounts/dfe8919e-00f6-4393-b042-3c1694b527fc"
        }
    },
    "createdAt": "2024-07-05T15:48:04.608+00:00",
    "balance": 27784.694714276924,
    "currency": "MAD",
    "type": "SAVING_ACCOUNT",
    "links": {
        "self": {
            "href": "http://localhost:8081/bankAccounts/82201eb7-bb79-4600-b09a-021e4cc0541e"
        },
        "bankAccount": {
            "href": "http://localhost:8081/bankAccounts/82201eb7-bb79-4600-b09a-021e4cc0541e"
        }
    }
}

```

```
{
  "_embedded": {
    "bankAccounts": [
      {
        "createdAt": "2024-07-05T15:48:04.528+00:00",
        "balance": 61259.61337791058,
        "currency": "MAD",
        "type": "CURRENT_ACCOUNT",
        "links": {
          "self": {
            "href": "http://localhost:8081/bankAccounts/b8f734a0-f051-4e64-b4dd-6d1796ac9e4f"
          },
          "bankAccount": {
            "href": "http://localhost:8081/bankAccounts/b8f734a0-f051-4e64-b4dd-6d1796ac9e4f"
          }
        }
      },
      {
        "createdAt": "2024-07-05T15:48:04.604+00:00",
        "balance": 51264.09237450511,
        "currency": "MAD",
        "type": "CURRENT_ACCOUNT",
        "links": {
          "self": {
            "href": "http://localhost:8081/bankAccounts/ff4c3596-d6ed-4973-82d0-4678460d8da2"
          },
          "bankAccount": {
            "href": "http://localhost:8081/bankAccounts/ff4c3596-d6ed-4973-82d0-4678460d8da2"
          }
        }
      },
      {
        "createdAt": "2024-07-05T15:48:04.607+00:00",
        "balance": 23630.126177170925,
        "currency": "MAD",
        "type": "CURRENT_ACCOUNT",
        "links": {
          "self": {
            "href": "http://localhost:8081/bankAccounts/ff4c3596-d6ed-4973-82d0-4678460d8da2"
          },
          "bankAccount": {
            "href": "http://localhost:8081/bankAccounts/ff4c3596-d6ed-4973-82d0-4678460d8da2"
          }
        }
      }
    ]
  }
}
```

```
{
  "_embedded": {
    "bankAccounts": [
      {
        "createdAt": "2024-07-05T15:48:04.528+00:00",
        "balance": 61259.61337791058,
        "currency": "MAD",
        "type": "CURRENT_ACCOUNT",
        "links": {
          "self": {
            "href": "http://localhost:8081/bankAccounts/b8f734a0-f051-4e64-b4dd-6d1796ac9e4f"
          },
          "bankAccount": {
            "href": "http://localhost:8081/bankAccounts/b8f734a0-f051-4e64-b4dd-6d1796ac9e4f"
          }
        }
      },
      {
        "createdAt": "2024-07-05T15:48:04.604+00:00",
        "balance": 51264.09237450511,
        "currency": "MAD",
        "type": "CURRENT_ACCOUNT",
        "links": {
          "self": {
            "href": "http://localhost:8081/bankAccounts/ff4c3596-d6ed-4973-82d0-4678460d8da2"
          },
          "bankAccount": {
            "href": "http://localhost:8081/bankAccounts/ff4c3596-d6ed-4973-82d0-4678460d8da2"
          }
        }
      },
      {
        "createdAt": "2024-07-05T15:48:04.607+00:00",
        "balance": 23630.126177170925,
        "currency": "MAD",
        "type": "CURRENT_ACCOUNT",
        "links": {
          "self": {
            "href": "http://localhost:8081/bankAccounts/ff4c3596-d6ed-4973-82d0-4678460d8da2"
          },
          "bankAccount": {
            "href": "http://localhost:8081/bankAccounts/ff4c3596-d6ed-4973-82d0-4678460d8da2"
          }
        }
      }
    ]
  }
}
```

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

```

1 package org.sid.bank_account_service.entities;
2
3 import org.sid.bank_account_service.enums.AccountType;
4 import org.springframework.data.rest.core.config.Projection;
5
6 @Projection(types = BankAccount.class, name = "p1")
7 public interface AccountProjection {
8     public String getId();
9     public AccountType getType();
10    public Double getBalance();
11 }
12

```

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

```

{
  "_embedded": {
    "bankAccounts": [
      {
        "id": "d1b60def-859b-44e5-b208-4bd772d4095c",
        "type": "CURRENT_ACCOUNT",
        "balance": 24953.058343217825,
        "links": {
          "self": {
            "href": "http://localhost:8081/bankAccounts/d1b60def-859b-44e5-b208-4bd772d4095c"
          },
          "bankAccount": {
            "href": "http://localhost:8081/bankAccounts/d1b60def-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": {
            "href": "http://localhost:8081/bankAccounts/4251feb4-0875-4aa8-b514-1f48282bf8c7"
          }
        }
      }
    ]
  }
}

```

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

```
{
  "embedded": {
    "bankAccounts": [
      {
        "createdAt": "2024-07-05T16:16:11.280+00:00",
        "balance": 83516.97498618242,
        "currency": "MAD",
        "type": "CURRENT_ACCOUNT",
        "links": {
          "self": {
            "href": "http://localhost:8081/bankAccounts/4e45d0ff-7a5b-4439-a812-176e031df3b6"
          },
          "bankAccount": {
            "href": "http://localhost:8081/bankAccounts/4e45d0ff-7a5b-4439-a812-176e031df3b6?projection",
            "templated": true
          }
        }
      },
      {
        "createdAt": "2024-07-05T16:16:11.282+00:00",
        "balance": 62582.00008181257,
        "currency": "MAD",
        "type": "CURRENT_ACCOUNT",
        "links": {
          "self": {
            "href": "http://localhost:8081/bankAccounts/cc07bfa9-fe20-4066-8e63-2aa54a1cced00"
          },
          "bankAccount": {
            "href": "http://localhost:8081/bankAccounts/cc07bfa9-fe20-4066-8e63-2aa54a1cced00?projection",
            "templated": true
          }
        }
      },
      {
        "createdAt": "2024-07-05T16:16:11.285+00:00",
        "balance": 33295.765692054236,
        "currency": "MAD"
      }
    ]
  }
}
```

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 @Transactional 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)

```

15  @Service
16  @Transactional
17  public class AccountServiceImpl implements AccountService {
18
19      @Autowired
20      private BankAccountRepository bankAccountRepository;
21
22      @Override
23      public BankAccountResponceDTO addAccount(BankAccountRequestDTO bankAccountRequestDTO) {
24
25          BankAccount bankAccount= BankAccount.builder()
26              .id(UUID.randomUUID().toString())
27              .createdAt(new Date())
28              .balance(bankAccountRequestDTO.getBalance())
29              .type(bankAccountRequestDTO.getType())
30              .currency(bankAccountRequestDTO.getCurrency())
31              .build();
32
33          BankAccount saveBankAccount = bankAccountRepository.save(bankAccount);
34          BankAccountResponceDTO bankAccountResponceDTO= new BankAccountResponceDTO().builder()
35              .id(saveBankAccount.getId())
36              .type(saveBankAccount.getType())
37              .createdAt(saveBankAccount.getCreatedAt())
38              .currency(saveBankAccount.getCurrency())
39
40      }
41
42  }

```

Et on va changer dans le contrôleur la méthode save

```

22
23     public AccountRestController(BankAccountRepository bankAccountRepository, AccountService accountService) {
24         this.bankAccountRepository= bankAccountRepository;
25
26         this.accountService = accountService;
27     }
28
29     @GetMapping("/bankAccounts")
30     public List<BankAccount> bankAccounts() { return bankAccountRepository.findAll(); }
31
32
33     @GetMapping("/{id}")
34     public BankAccount bankAccount(@PathVariable String id) {
35         return bankAccountRepository.findById(id)
36             .orElseThrow(() ->new RuntimeException(String.format("Account %s not found", id)));
37     }
38
39     @PostMapping("/bankAccounts")
40     public BankAccountResponceDTO save(@RequestBody BankAccountRequestDTO requestDTO) {
41         return accountService.addAccount(requestDTO);
42     }
43     @PutMapping("/{id}")
44     public BankAccount update(@PathVariable String id, @RequestBody BankAccount bankAccount)

```

On teste

Curl

```
curl -X 'POST' \
  'http://localhost:8081/api/bankAccounts' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
    "balance": 800,
    "currency": "USD",
    "type": "CURRENT_ACCOUNT"
}'
```

Request URL

http://localhost:8081/api/bankAccounts

Server response

Code	Details
200	<p>Response body</p> <pre>{   "id": "e0fcac61-3805-4f89-a1d1-1c847e1ffa94",   "createdAt": "2024-07-05T16:59:32.743+00:00",   "balance": 800,   "currency": "USD",   "type": "CURRENT_ACCOUNT" }</pre> <p>Download</p> <p>Response headers</p> <pre>connection: keep-alive content-type: application/json</pre>

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

```

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;
    }
}

```

```

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

    return bankAccountResponceDTO;
}
}

```

```

@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();
}

```

```

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 déclare les types comme le type Query dont on spécifie les requêtes 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)

```
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'ajouter 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

```

1 *query {
2   accountsList {
3     id
4   }
5 }
```

```

{
  "data": {
    "accountsList": [
      {
        "id": "12141fa7-1c7d-4241-be3f-06e30b2f2fef"
      },
      {
        "id": "8458c9cb-a244-4609-a640-247a189ddb4c"
      },
      {
        "id": "02455fb-2d67-4e3d-8a82-f43906bb1bbf"
      },
      {
        "id": "d02d2880-b5f7-4568-822d-ffb588968ad9"
      },
      {
        "id": "c1026a16-e277-4061-8281-aeb5c1e82d62"
      },
      {
        "id": "4d9a55cf-e4c6-478e-9d9b-9330a914deae"
      }
    ]
  }
}
```

```

1 *query {
2   accountsList {
3     id, balance, currency, type
4   }
5 }
```

```

{
  "data": {
    "accountsList": [
      {
        "id": "12141fa7-1c7d-4241-be3f-06e30b2f2fef",
        "balance": 44170.7123308203,
        "currency": "MAD",
        "type": "CURRENT_ACCOUNT"
      },
      {
        "id": "8458c9cb-a244-4609-a640-247a189ddb4c",
        "balance": 10623.114868460218,
        "currency": "MAD",
        "type": "CURRENT_ACCOUNT"
      },
      {
        "id": "02455fb-2d67-4e3d-8a82-f43906bb1bbf",
        "balance": 41477.384242104876,
        "currency": "MAD",
        "type": "SAVING_ACCOUNT"
      }
    ]
  }
}
```

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

```
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 schème

```
type Query{  
    accountsList : [BankAccount],  
    accountById (id: String) : BankAccount  
}  
  
type BankAccount {  
    id: String,  
    createdAt: Float,  
    balance: Float,  
    currency: String,  
    type: String  
}
```

On teste

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

```

query {
  bankAccountById(id: "b3be5e64-83f2-47e1-b99b-e2") {
    currency, type, balance
  }
}

```

On the right, the results of the query are displayed in a JSON-like format:

```

{
  "data": {
    "bankAccountById": {
      "currency": "MAD",
      "type": "CURRENT_ACCOUNT",
      "balance": 74443.6425201022
    }
  }
}

```

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

```

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

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

```

1 * query {
2 *   bankAccountById (id: "aaaa") {
3 *     currency, type, balance
4 *   }
5 * }
6 *

```

On the right, the results of the query are displayed in a JSON-like format:

```

{
  "errors": [
    {
      "message": "Account aaaa not found"
    }
  ],
  "data": {
    "bankAccountById": null
  }
}

```

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

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

The screenshot shows a Java code editor with a dark theme. The code is contained within a class named `BankAccountGraphQLController`:

```

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")));
    }

    @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

```

@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

The screenshot shows the Android Studio GraphQL plugin interface. On the left, there is a code editor window titled "untitled" containing a GraphQL mutation query:

```

1 mutation {
2   addAccount(bankAccount : {
3     type:" SAVING_ACCOUNT",
4     balance: 4000,
5     currency: "USD"
6   })
7   {
8     id, type, balance
9   }
10 }

```

On the right, the results of the mutation are displayed in a JSON tree view:

```

{
  "data": {
    "addAccount": {
      "id": "2607dc80-1d09-494d-92b5-7ace309a1509",
      "type": "SAVING_ACCOUNT",
      "balance": 4000
    }
  }
}

```

Below the code editor, there are tabs for "Variables" and "Headers". At the bottom center, it says "Android Studio".

```

query {
  accountsList {
    id, balance
  }
}

```

The response shows a list of accounts with their IDs and balances:

```

[{"balance": "3153.53/06/20032", "id": "3b49b94c-1928-4585-bd34-ba9de0c20a90", "balance": 12903.753683161674}, {"balance": "84139.27558608798", "id": "5b93c66a-298e-4f53-ad93-a1933c7fc00f", "balance": 84139.27558608798}, {"balance": "44874.10945259202", "id": "6b6dd489-40c1-4da7-8df4-cb322579abe9", "balance": 44874.10945259202}, {"balance": "4000", "id": "2607dc80-1d09-494d-92b5-7ace309a1509", "balance": 4000}]

```

On peut utiliser une requête paramétrée

```

mutation($t: String, $b: Float, $c: String) {
  addAccount(bankAccount : {
    type: $t,
    balance: $b,
    currency: $c
  })
  id, type, b t ice
}

```

The response shows the added account details:

```

{
  "data": {
    "addAccount": {
      "id": "5bda104b-1377-4c02-a160-4983bb0d30b6",
      "type": "CURRENT_ACCOUNT",
      "balance": 3000
    }
  }
}

```

On va ajouter une méthode update

```

        }
    @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;
    }
}

```

```

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

```

On va déclarer la méthode dans le fichier schema.graphqls

```

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

```

On ajoute la méthode deleteAccount

```

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

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

```

On teste pour la méthode update

```

mutation($id: String!, $t: String!, $b: Float!, $c: String!) {
  updateAccount(
    id: $id
    bankAccount: {
      type: $t,
      balance: $b,
      currency: $c
    }
  ) {
    id, type, balance
  }
}

```

The playground shows the response to the mutation:

```

{
  "data": {
    "updateAccount": {
      "id": "bad6e292-8331-4eab-ba10-f579987cc124",
      "type": "CURRENT_ACCOUNT",
      "balance": 32000
    }
  }
}

```

Pour tester la méthode deleteAccount

```

mutation {
  deleteAccount(id: "bad6e292-8331-4eab-ba10-f579987cc124")
}

```

The playground shows the response to the mutation:

```

{
  "data": {
    "deleteAccount": true
  }
}

```

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

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

```

1 * mutation {
2   deleteAccount(id: "9593a93e-c0a4-488f-adcb-df204
3
4
5 }

```

On the right, the results of the mutation are displayed in a JSON-like tree structure:

```

{
  "data": {
    "deleteAccount": null
  }
}

```

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

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

```

@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)

```

@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 : AccountType.SAVINGS)
                    .balance(10000 + Math.random() * 90000)
            }
        });
    });
}

```

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

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

```

1 query {
2   accountsList {
3     id, balance, customer {name, bankAccounts{id}}
4   }
5 }
6

```

On the right, the results are displayed in a JSON-like tree view:

```

{
  "data": {
    "accountsList": [
      {
        "id": "eccd4f4e-dd61-430b-a06f-a5be3f3b6cef",
        "balance": 29026.4553890405,
        "customer": {
          "name": "Mohamed",
          "bankAccounts": [
            {
              "id": "eccd4f4e-dd61-430b-a06f-a5be3f3b6cef"
            },
            {
              "id": "685a3216-eb74-46f6-9471-72d6d771c951"
            },
            {
              "id": "27115a18-eff5-49dc-b2fc-74f1c2b218be"
            },
            {
              "id": "dadadf5b-f614-4c18-88ac-3e91cbad63a5"
            }
          ]
        }
      }
    ]
  }
}

```

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]
}

```

```

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

```

The screenshot shows a browser window with multiple tabs. The active tab is 'localhost:8081/graphiql?path=/graphql'. On the left, there is a code editor with a GraphQL query:

```

query {
  customers {
    id, name, bankAccounts {balance, id, type}
  }
}

```

On the right, the results of the query are displayed in a tree-view JSON format:

```

{
  "data": {
    "customers": [
      {
        "id": "1",
        "name": "Mohamed",
        "bankAccounts": [
          {
            "balance": 94782.3187917525,
            "id": "a8182786-b249-4ca1-802d-1978c31ce939",
            "type": "SAVING_ACCOUNT"
          },
          {
            "balance": 38705.94147400234,
            "id": "c8f96cc8-a004-49ac-afe0-3b6d3146b031",
            "type": "SAVING_ACCOUNT"
          },
          {
            "balance": 89373.23439797755,
            "id": "b4154ca9-24ae-4fbb-9beb-08c5248c599d",
            "type": "CURRENT_ACCOUNT"
          }
        ]
      }
    ]
  }
}

```

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

```

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

```

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

```

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)

```

[
  {
    "id": "6b22310-0a54-45d3-a2f6-956700008158",
    "createdAt": "2024-07-06T11:48:59.351+00:00",
    "balance": 50872.6242240465,
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT",
    "customer": {
      "id": 1,
      "name": "Mohamed"
    }
  },
  {
    "id": "200b8176-cc66-40bb-ad94-ef63f394a143",
    "createdAt": "2024-07-06T11:48:59.370+00:00",
    "balance": 86737.7142369525,
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT",
    "customer": {
      "id": 1,
      "name": "Mohamed"
    }
  },
  {
    "id": "9084ef10-3065-4890-bfe8-58b533e6d8fb",
    "createdAt": "2024-07-06T11:48:59.372+00:00",
    "balance": 73925.8672701958,
    "currency": "MAD",
    "type": "SAVING_ACCOUNT",
    "customer": {
      "id": 1,
      "name": "Mohamed"
    }
  }
]

```

## Deuxième partie : Développer une architecture micro-service

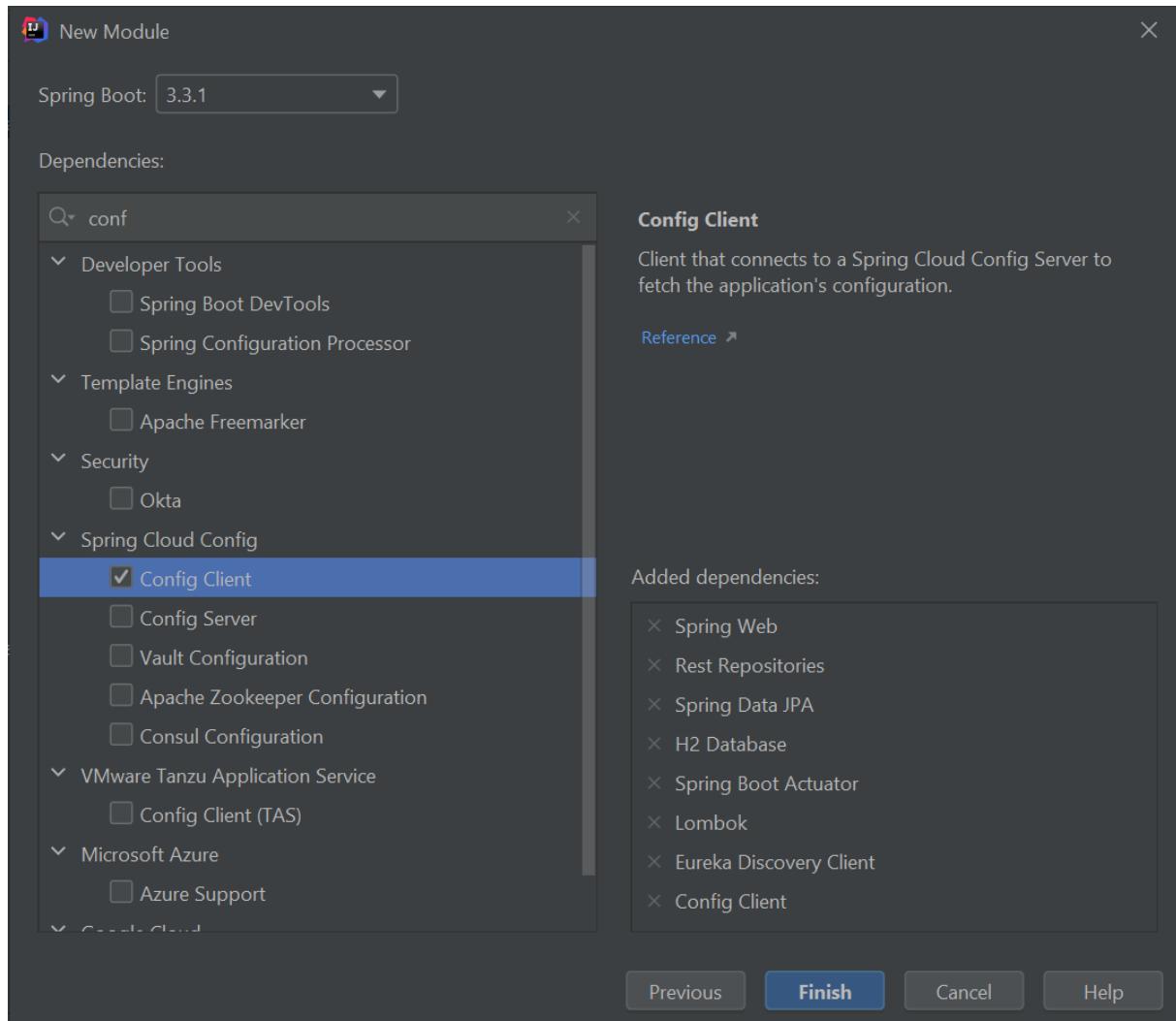
Partie 1 :

L'approche micro service est un style d'architecture qui consiste à découper un grand domaine métier 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

```
C Customer.java ×

4  import jakarta.persistence.Entity;
5  import jakarta.persistence.GeneratedValue;
6  import jakarta.persistence.GenerationType;
7  import jakarta.persistence.Id;
8  import lombok.*;
9
10 @Entity
11 @Getter @Setter @ToString @NoArgsConstructor @AllArgsConstructor @Builder
12 public class Customer {
13     @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private Long id;
15     private String firstName;
16     private String lastName;
17     private String email;
18 }
19
```

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

```
C CustomerRestController.java ×  
6 import org.springframework.web.bind.annotation.PathVariable;  
7 import org.springframework.web.bind.annotation.RestController;  
8  
9 import java.util.List;  
10  
11 @RestController  
12 public class CustomerRestController {  
13  
14     private CustomerRepository customerRepository;  
15  
16     public CustomerRestController(CustomerRepository customerRepository) {  
17         this.customerRepository = customerRepository;  
18     }  
19     @GetMapping("/customers")  
20     public List<Customer> customerList() {  
21         return customerRepository.findAll();  
22     }  
23     @GetMapping("/customers/{id}")  
24     public Customer customerById(@PathVariable Long id) {  
25         return customerRepository.findById(id).get();  
26     }  
}
```

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 x 2 ^

14
15  public static void main(String[] args) { SpringApplication.run(CustomerServiceApplication.class, args);
16
17  }
18
19  CommandLineRunner 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
|
```

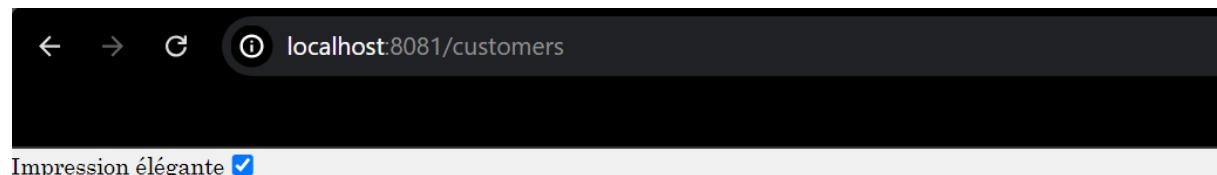
Pour résoudre ce problème on va désactiver les services de configuration de discovery dans fichier properties et on fait la configuration qu'on a besoin

```

application.properties x

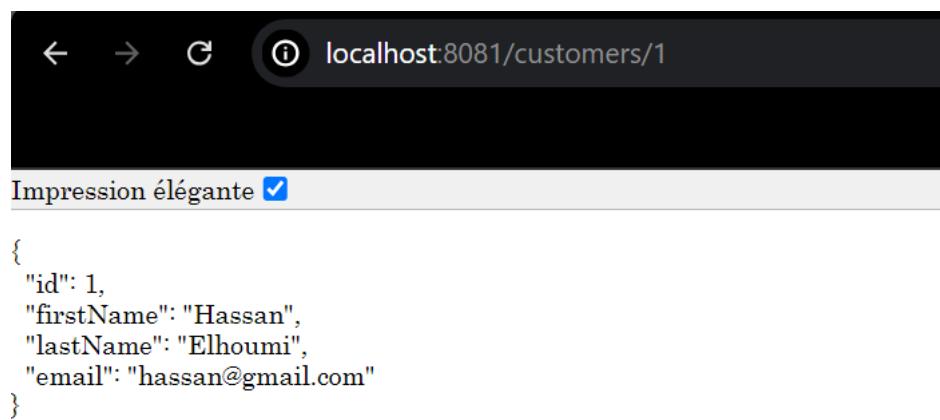
1  spring.application.name=customer-service
2  server.port=8081
3  spring.datasource.url=jdbc:h2:mem:customer-db
4  spring.h2.console.enabled=true
5  spring.cloud.discovery.enabled=false
6  spring.cloud.config.enabled=false
7
```

On teste



Impression élégante

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



Impression élégante

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

Si on veut consulter la base de données H2

localhost:8081/h2-console/login.do?jsessionid=5d279245c0ffe04aaf1446031c440fc8

Auto commit | Max rows: 1000 | Auto complete | Off | Auto select | On | ?

jdbc:h2:mem:customer-db | Run | Run Selected | Auto complete | Clear | SQL statement:

SELECT \* FROM CUSTOMER |

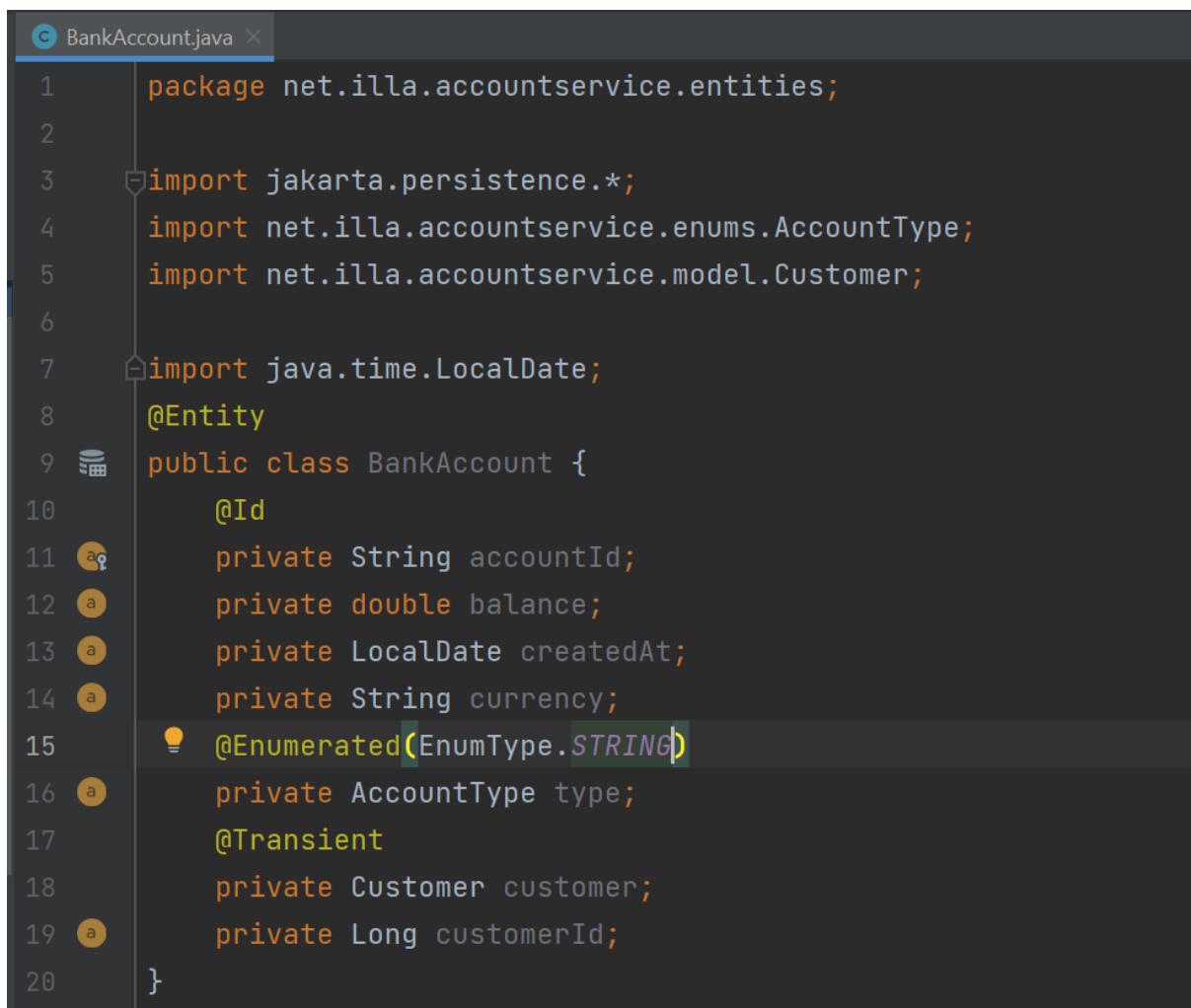
SELECT \* FROM CUSTOMER;

ID	EMAIL	FIRST_NAME	LAST_NAME
1	hassan@gmail.com	Hassan	Elhoumi
2	hassan@gmail.com	Mohamed	Elhannaoui

(2 rows, 5 ms)

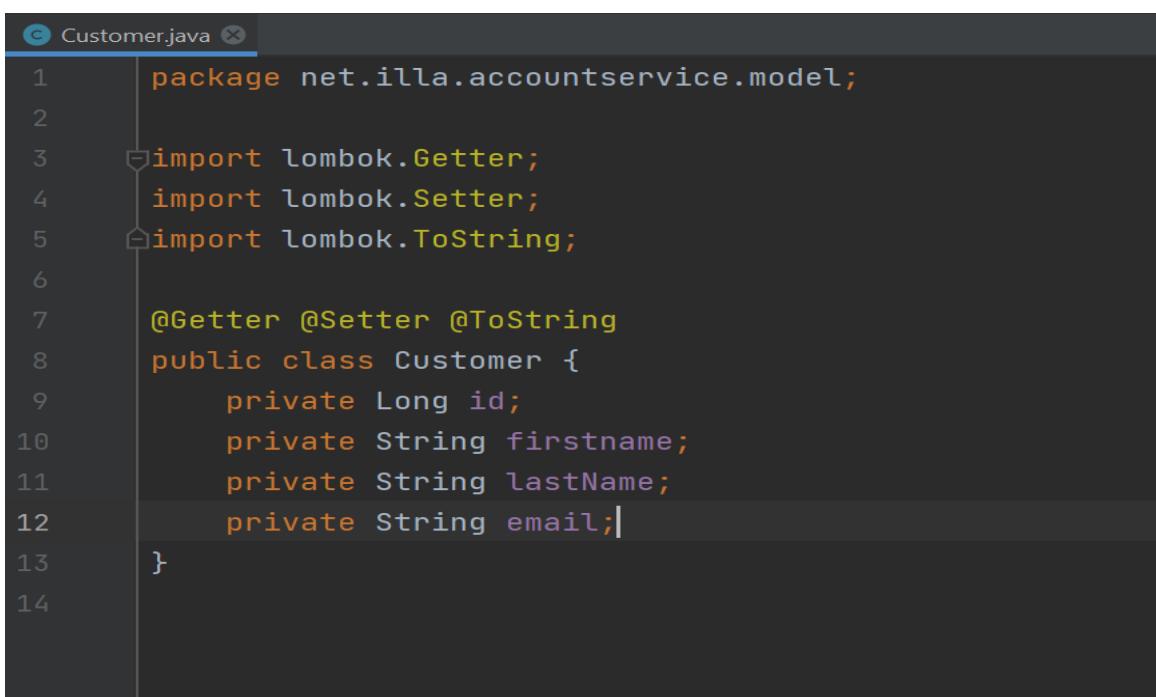
Edit

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 @Transient pour ignorer cet attribut



```
1 package net.illa.accountservic.entities;
2
3 import jakarta.persistence.*;
4 import net.illa.accountservic.enums.AccountType;
5 import net.illa.accountservic.model.Customer;
6
7 import java.time.LocalDate;
8
9 @Entity
10 public class BankAccount {
11     @Id
12     private String accountId;
13     private double balance;
14     private LocalDate createdAt;
15     private String currency;
16     @Enumerated(EnumType.STRING)
17     private AccountType type;
18     @Transient
19     private Customer customer;
20     private Long customerId;
21 }
```

Voilà la classe simple Customer



```
1 package net.illa.accountservic.model;
2
3 import lombok.Getter;
4 import lombok.Setter;
5 import lombok.ToString;
6
7 @Getter @Setter @ToString
8 public class Customer {
9     private Long id;
10    private String firstname;
11    private String lastName;
12    private String email;
13 }
14
```

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

```
c AccountRestController.java ×
9
10 import java.util.List;
11
12 @RestController
13 public class AccountRestController {
14     private BankAccountRepository accountRepository;
15
16     public AccountRestController(BankAccountRepository accountRepository) {
17         this.accountRepository=accountRepository;
18     }
19     @GetMapping("accounts")
20     public List<BankAccount> accountList() {
21         return accountRepository.findAll();
22     }
23     @GetMapping("accounts/{id}")
24     public BankAccount bankAccountById(@PathVariable String id){
25         return accountRepository.findById(id).get();
26     }
27 }
28 }
```

Alors on va faire le test

```

AccountServiceApplication.java ×
20     CommandLineRunner commandLineRunner(BankAccountRepository bankAccountRepository) {
21         return args ->{
22             BankAccount bankAccount1=new BankAccount().builder()
23                 .accountId(UUID.randomUUID().toString())
24                 .currency("MAD")
25                 .balance(98000)
26                 .createdAt(LocalDate.now())
27                 .type(AccountType.CURRENT_ACCOUNT)
28                 .customerId(Long.valueOf(1))
29                 .build();
30             BankAccount bankAccount2=new BankAccount().builder()
31                 .accountId(UUID.randomUUID().toString())
32                 .currency("MAD")
33                 .balance(12000)
34                 .createdAt(LocalDate.now())
35                 .type(AccountType.SAVING_ACCOUNT)
36                 .customerId(Long.valueOf(2))
37                 .build();
38             bankAccountRepository.save(bankAccount1);
39             bankAccountRepository.save(bankAccount2);
40

```

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

```

application.properties ×
1 spring.application.name=account-service
2 spring.datasource.url=jdbc:h2:mem:account-db
3 spring.h2.console.enabled=true
4 server.port=8082
5 spring.cloud.discovery.enabled=false
6 spring.cloud.config.enabled=false
7

```

On a le résultat

← → ⌂ ⓘ localhost:8082/accounts

[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",  

```

← → ⌂ ⓘ localhost:8082/accounts/e459ccc5-f535-4648-bd5a-889e79a1c444

[Impression élégante

```
{  
  "accountId": "e459ccc5-f535-4648-bd5a-889e79a1c444",  
  "balance": 98000,  
  "createdAt": "2024-07-07",  
  "currency": "MAD",  

```

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

```

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

```

On teste

Request URL: <http://localhost:8082/accounts>

Server response:

Code	Details
200	Response body

```
[
  {
    "accountId": "70e06d02-40b7-432f-805d-1ad8c4439516",
    "balance": 98000,
    "createdAt": "2024-07-07",
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT",
    "customer": null,
    "customerId": 1
  },
  {
    "accountId": "8c4a65d5-a70c-404f-abaa4-48c805ca3815",
    "balance": 12000,
    "createdAt": "2024-07-07",
    "currency": "MAD",
    "type": "SAVING_ACCOUNT",
    "customer": null,
    "customerId": 2
  }
]
```

Response headers:

Responses

Curl:

```
curl -X 'GET' \
'http://localhost:8082/accounts/70e06d02-40b7-432f-805d-1ad8c4439516' \
-H 'accept: application/hal+json'
```

Request URL: <http://localhost:8082/accounts/70e06d02-40b7-432f-805d-1ad8c4439516>

Server response:

Code	Details
200	Response body

```
{
  "accountId": "70e06d02-40b7-432f-805d-1ad8c4439516",
  "balance": 98000,
  "createdAt": "2024-07-07",
  "currency": "MAD",
  "type": "CURRENT_ACCOUNT",
  "customer": null,
  "customerId": 1
}
```

Response headers:

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'annotation @RestController

On désactive la dépendance rest

The screenshot shows the Swagger UI interface for an OpenAPI definition. At the top, it says "OpenAPI definition v0 OAS 3.0". Below that, there's a dropdown for "Servers" set to "http://localhost:8082 - Generated server url". The main area is titled "account-rest-controller". It contains two GET operations: "/accounts" and "/accounts/{id}". The "/accounts" operation is expanded to show its schema, which is labeled "BankAccount".

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

The screenshot shows the Swagger UI interface for testing a POST request to the "/customers" endpoint. The request URL is "http://localhost:8081/customers". The request body is a JSON object with fields: "firstName": "aa", "lastName": "bb", and "email": "cc". The response code is 201, and the response body is a JSON object with the same fields and a "links" field containing a self-link.

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

The screenshot shows the Swagger UI interface for an OpenAPI definition. At the top, it says "OpenAPI definition v0 OAS 3.0". Below that, there's a "Servers" dropdown set to "http://localhost:8081 - Generated server url". The main section is titled "customer-rest-controller". It contains two "GET" requests: one for "/customers" and another for "/customers/{id}".

Maintenant, on va voir 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/sorties 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

```

application.yml
1  spring:
2    cloud:
3      gateway:
4        routes:
5          - id: r1
6            uri: http://localhost:8081/
7            predicates:
8              - Path=/customers/**
9          - id: r2
10            uri: http://localhost:8082/
11            predicates:
12              - 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

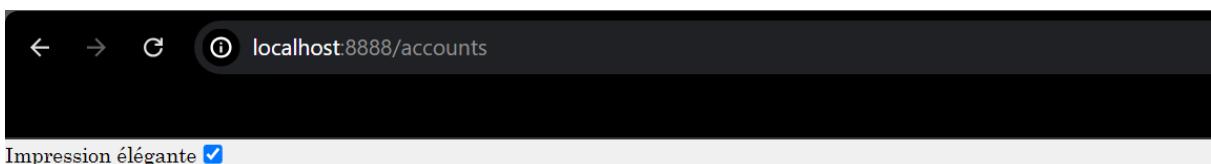
```

    - id: r2
      uri: http://localhost:8082/
      predicates:
        - Path=/accounts/**

    application:
      name: gateway-service

server:
  port: 8888
  
```

On va démarrer ce gateway



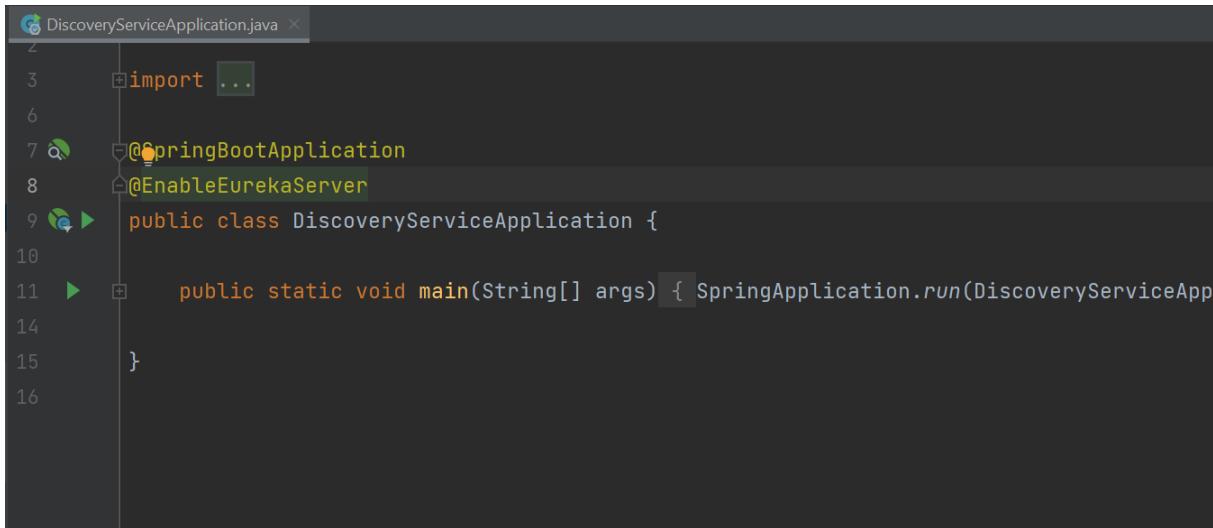
```
[
{
  "accountId": "78ccb92a-b36d-41aa-8ce3-0de228ac3753",
  "balance": 98000,
  "createdAt": "2024-07-07",
  "currency": "MAD",
  "type": "CURRENT_ACCOUNT",
  "customer": null,
  "customerId": 1
},
{
  "accountId": "baa7b014-f205-4b43-be6f-7e761e4a1e16",
  "balance": 12000,
  "createdAt": "2024-07-07",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



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

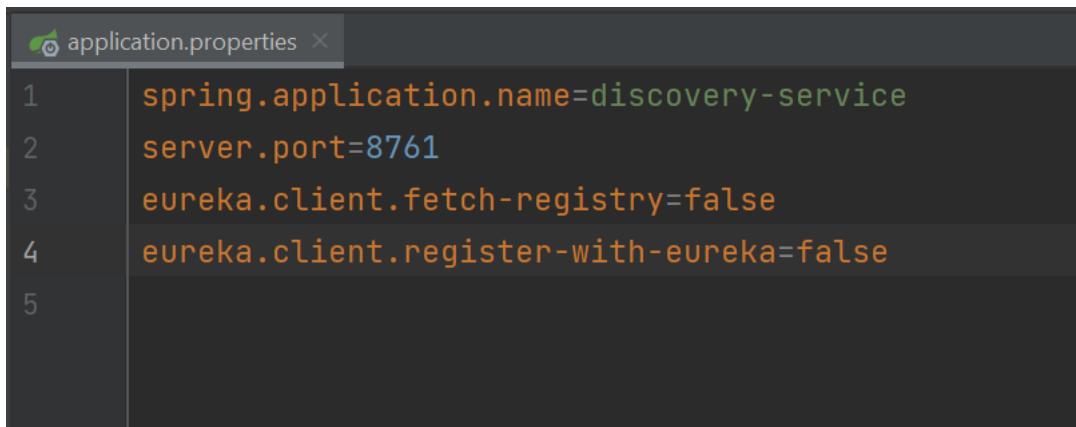
Ça c'est une configuration statique où on suppose qu'on connaît les adresses des microservices mais dans la pratique, on ne connaît pas donc la solution c'est de faire la configuration dynamique (utiliser Discovery service)

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



```
DiscoveryServiceApplication.java ×  
1 import ...  
2  
3 @SpringBootApplication  
4 @EnableEurekaServer  
5 public class DiscoveryServiceApplication {  
6  
7     public static void main(String[] args) { SpringApplication.run(DiscoveryServiceApp...  
8 }  
9 }
```

On configure application.properties



```
application.properties ×  
1 spring.application.name=discovery-service  
2 server.port=8761  
3 eureka.client.fetch-registry=false  
4 eureka.client.register-with-eureka=false  
5
```

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

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACCOUNT-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-I0MOA39:account-service:8082
CUSTOMER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-I0MOA39:customer-service:8081
GATEWAY-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-I0MOA39:gateway-service:8888

General Info

Name	Value
total-avail-memory	100mb
num-of-cpus	8
current-memory-usage	45mb (44%)
server-uptime	00:00

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}
```

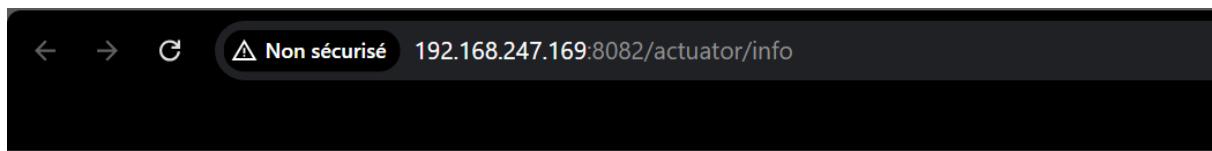
localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACCOUNT-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-I0MOA39:account-service:8082
CUSTOMER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-I0MOA39:customer-service:8081
GATEWAY-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-I0MOA39:gateway-service:8888

General Info

Name	Value
------	-------



## 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).

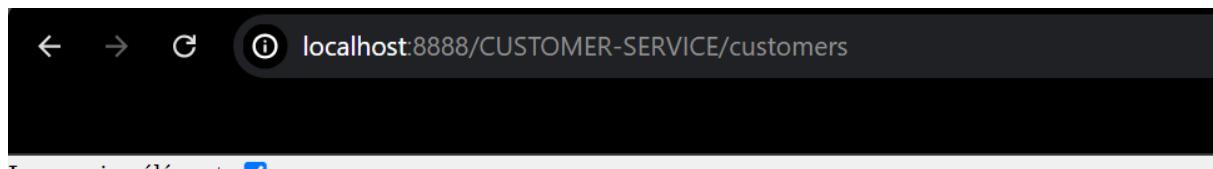
Maintenant notre gateway, on va le configurer pour qu'il puisse chercher les adresses dans discovery

On fait une configuration dynamique



```
GatewayServiceApplication.java
13
14  public static void main(String[] args) { SpringApplication.run(GatewayServiceApplication.class, args); }
15
16
17
18  @Bean
19  DiscoveryClientRouteDefinitionLocator locator(ReactiveDiscoveryClient rdc, DiscoveryLister dlp) {
20      return new DiscoveryClientRouteDefinitionLocator(rdc, dlp);
21  }
22
23 }
```

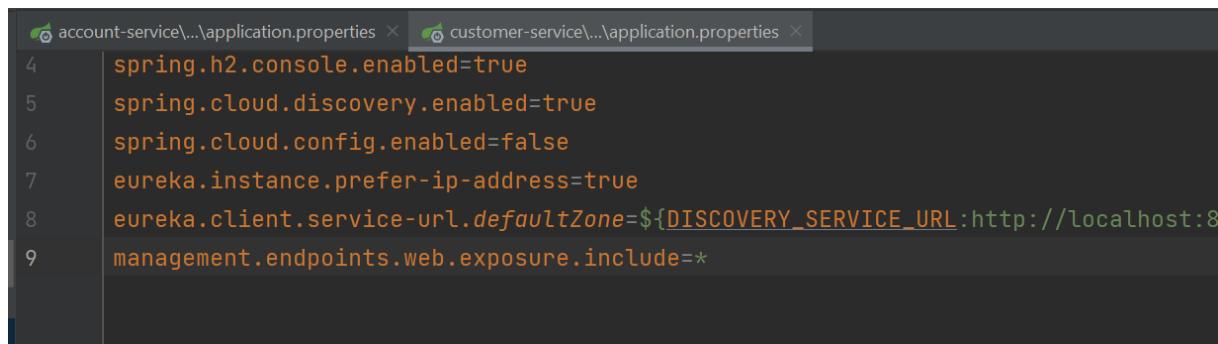
Après on démarre



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



```
account-service\\application.properties x customer-service\\application.properties x  
4  spring.h2.console.enabled=true  
5  spring.cloud.discovery.enabled=true  
6  spring.cloud.config.enabled=false  
7  eureka.instance.prefer-ip-address=true  
8  eureka.client.service-url.defaultZone=${DISCOVERY_SERVICE_URL:http://localhost:8  
9  management.endpoints.web.exposure.include=*
```

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": {  
          "aliases": []  
        }  
      }  
    }  
  }  
}
```

← → ⌂ ⓘ localhost:8888/ACCOUNT-SERVICE/actuator/health

Impression élégante

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

← → ⌛ ⓘ localhost:8888/ACCOUNT-SERVICE/actuator/metrics

Impression élégante

```
"names": [  
    "application.ready.time",  
    "application.started.time",  
    "disk.free",  
    "disk.total",  
    "executor.active",  
    "executor.completed",  
    "executor.pool.core",  
    "executor.pool.max",  
    "executor.pool.size",  
    "executor.queue.remaining",  
    "executor.queued",  
    "hikaricp.connections",  
    "hikaricp.connections.acquire",  
    "hikaricp.connections.active",  
    "hikaricp.connections.creation",  
    "hikaricp.connections.idle",  
    "hikaricp.connections.max",  
    "hikaricp.connections.min",  
    "hikaricp.connections.pending",  
    "hikaricp.connections.timeout",  
    "hikaricp.connections.usage",  
    "http.client.requests",  
    "http.client.requests.active",  
    "http.server.requests",  
    "http.server.requests.active",  
    "jdbc.connections.active",  
    "jdbc.connections.idle",  
    "jdbc.connections.max",  
    "jdbc.connections.min",  
    "jvm.buffer.count",  
    "jvm.buffer.memory.used",
```

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 connaître son nom et pour ce faire nous aurons besoin d'envoyer une requête vers customer-service du côté 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>
    <groupId>org.springframework.boot</groupId>
```

Pour utiliser open Feign, 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
1 package net.illa.accounts.service.clients;
2
3 import net.illa.accounts.service.model.Customer;
4 import org.springframework.cloud.openfeign.FeignClient;
5 import org.springframework.web.bind.annotation.GetMapping;
6
7 import java.util.List;
8 @FeignClient(name= "CUSTOMER-SERVICE")
9 public interface CustomerRestClient {
10     @GetMapping("/customers/{id}")
11     Customer findCustomerById(Long id);
12     @GetMapping("/customers")
13     List<Customer> allCustomers();
14 }
15
```

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

```

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

    public AccountRestController(BankAccountRepository accountRepository, CustomerRestClient customerRestClient) {
        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 openFeign si on veut l'utiliser

```

package net.illa.accountservic;

import ...

@SpringBootApplication
@EnableFeignClients
public class AccountServiceApplication {

    public static void main(String[] args) { SpringApplication.run(AccountServiceApplication.class, args); }

    @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 resilience4j

```

      </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 ...

```
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 Available");
        customer.setLastName("Not Available");
        customer.setEmail("Not Available");
        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 Available");
    customer.setLastName("Not Available");
    customer.setEmail("Not Available");
    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, CustomerRestClient customerRestClient) {
    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 :

localhost:8888/ACCOUNT-SERVICE/accounts

```
<?xml version="1.0" encoding="UTF-8"?>
<accounts>
    <item>
        <accountId>8fc1d0a6-6a25-4bee-ab56-0a2fe4d4ba7d</accountId>
        <balance>16680.770837401094</balance>
        <createdAt>2024-07-09</createdAt>
        <currency>MAD</currency>
        <type>CURRENT_ACCOUNT</type>
        <customer/>
        <customerId>1</customerId>
    </item>
    <item>
        <accountId>fb8ebca3-b931-468e-83c3-e7ddbe6b1ab4</accountId>
        <balance>2144.847049885923</balance>
        <createdAt>2024-07-09</createdAt>
        <currency>MAD</currency>
        <type>SAVING_ACCOUNT</type>
        <customer/>
        <customerId>1</customerId>
    </item>
    <item>
        <accountId>e62c683d-5099-4060-883c-0f02e90907f2</accountId>
        <balance>43978.806080134185</balance>
        <createdAt>2024-07-09</createdAt>
        <currency>MAD</currency>
        <type>CURRENT_ACCOUNT</type>
        <customer/>
        <customerId>2</customerId>
    </item>
    <item>
        <accountId>5339d8d7-fc7f-43f4-94a9-4df2025183ac</accountId>
        <balance>8204.130446748166</balance>
        <createdAt>2024-07-09</createdAt>
        <currency>MAD</currency>
        <type>SAVING_ACCOUNT</type>
        <customer/>
        <customerId>2</customerId>
    </item>
</accounts>
```

## 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.