جامعة مولاي إسماعيل
ⵜⴰⵙⴷⴰⵡⵉⵜ ⵎⵓⵍⴰⵢ ⵉⵙⵎⴰⵄⵉⵍ
UNIVERSITÉ MOULAY ISMAÏL

كلية العلوم
ⵜⴰⵖⵔⵉⵎⵜ ⵏ ⵜⵣⵔⴼⵜ
FACULTÉ DES SCIENCES

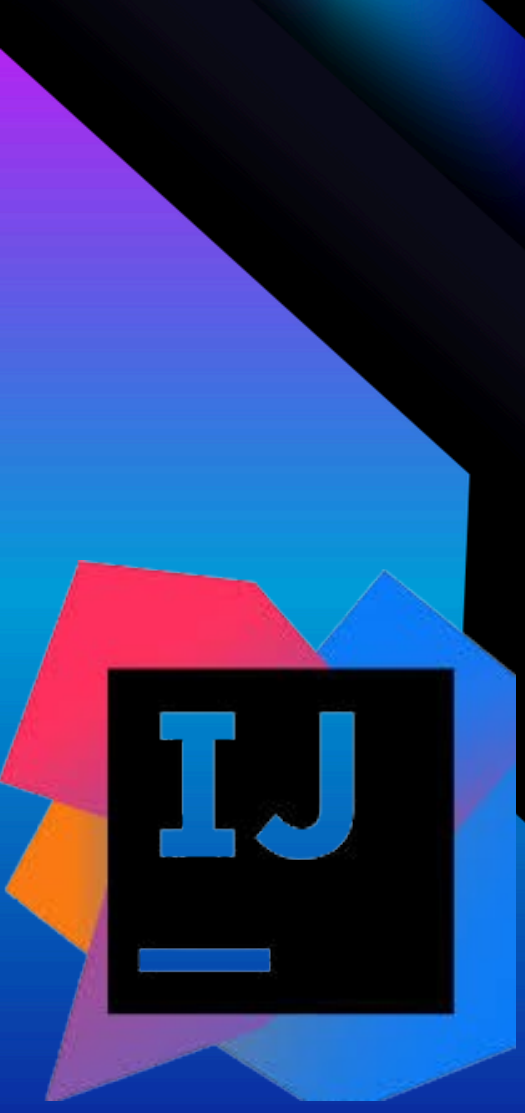
01-07-2024

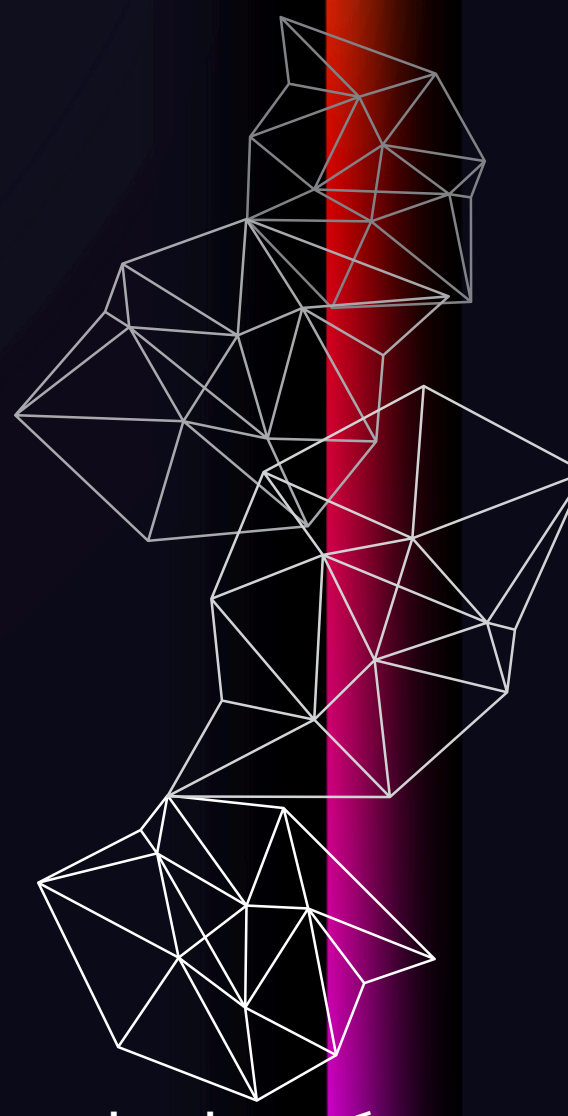
RAPPORT TP6

ARCHITECTURES MICRO-SERVICES

Réalisé par :
Bouchama Hajar

Demandé par :
Mohamed Youssfi





Master : Intelligence artificielle et analyse de données

Introduction

Dans le cadre de l'Activité Pratique N°6 sur les architectures micro-services, nous avons entrepris le développement d'une application basée sur les micro-services, dont l'objectif est de gérer des clients et des comptes bancaires, chaque compte étant associé à un client spécifique. Cette activité pratique se décompose en deux parties principales.

Dans la première partie, nous avons développé des micro-services en utilisant deux types de services web : RESTful et GraphQL. Cette section nous a permis de comprendre les différences entre ces deux approches et leurs applications dans la création de micro-services. Les ressources pédagogiques suivantes ont été particulièrement utiles :

- Micro-service avec Web Service RESTful
- Micro-service avec Web Service GraphQL

Dans la deuxième partie, nous avons conçu et implémenté une architecture de micro-services complète pour notre application. Cette architecture comprend plusieurs services interdépendants, chacun ayant une responsabilité spécifique :

1. Discovery Service : Ce service permet la découverte et l'enregistrement des micro-services, facilitant ainsi leur interaction.
2. Config Service : Ce service centralise la gestion des configurations pour l'ensemble des micro-services, assurant la cohérence et la facilité de maintenance.
3. Customer Service : Ce micro-service gère les informations relatives aux clients.
4. Account Service : Ce micro-service gère les informations relatives aux comptes bancaires.

Grâce à cette activité pratique, nous avons pu acquérir une compréhension approfondie des avantages et des défis des architectures micro-services. Nous avons également développé des compétences techniques essentielles pour la création, la gestion et l'orchestration de micro-services dans un environnement distribué. Ce rapport détaille les étapes suivies, les technologies utilisées et les résultats obtenus dans la réalisation de cette solution.

Enoncé !

Première Partie : Développer un micro-service
- MIcro Service avec Web Service RESTFUL :
  https://www.youtube.com/watch?v=2-qIoZcvhAw
- Micro-SERVICE wvec web service GRAPHQL :
  https://www.youtube.com/watch?v=FsdR09jlqaE

Deuxième partie : Développer une architecture micro-service :

Objectif :
Créer une application basée sur les micro-services qui permet des gérer des clients et des comptes bancaire. chaque compte appartient à un client.
Travail à faire :
1. Créer le micro-service Discovery Service
2. Créer le micro-service Config Service
3. Créer le Micro-service Customer Service
4. Créer le micro-service Account Service
5. Créer un frontend basé sur Angular
6. Automatiser le déploiement de l'ensemble des micro-services en utilisant Docker ret Docker Compose

Vidéos :

1. https://www.youtube.com/watch?v=BqNZJwCvnAE&authuser=0
2. https://www.youtube.com/watch?v=D0Vzlmczups&authuser=0
3. https://www.youtube.com/watch?v=tApkq6u4sh4&authuser=0
4. https://www.youtube.com/watch?v=aQ5lIkEJaCc&authuser=0
5. https://www.youtube.com/watch?v=lNJwMXEUUtA&authuser=0
6. https://www.youtube.com/watch?v=AieHUU_9Kwg&authuser=0

Code et explications :

Partie 1

1. MIcro-Service avec Web Service RESTFUL

On cree la classe BankAccoun

- cette classe BankAccount représente une entité de compte bancaire dans le service de gestion des comptes bancaires de notre application

```java
package ma.bouchama.bankaccountservice.entities;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import ma.bouchama.bankaccountservice.enums.AccountType;

import java.util.Date;
5 usages
@Entity @Data @NoArgsConstructor @AllArgsConstructor @Builder

public class BankAccount {
    @Id
    private String id;
    private Date createdAt;
    private double balance;
    private String currency;
    private AccountType type;

}
```

- @Entity : Indique que cette classe est une entité JPA.
- @Data, @NoArgsConstructor, @AllArgsConstructor, @Builder : Annotations Lombok pour générer automatiquement les méthodes getters, setters, les constructeurs, et un pattern de conception builder.

Créer le enum AccountType

```java
package ma.bouchama.bankaccountservice.enums;

5 usages
public enum AccountType {
    1 usage
    CURRENT_ACCOUNT, SAVING_ACCOUNT
}
```

Créer l'interface BankAccountRepository

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

}
```

Creer des nouveaux accounts :

- Cette classe BankAccountServiceApplication constitue le point d'entrée principal de notre application Spring Boot :

```java
package ma.bouchama.bankaccountservice;

import ...

@SpringBootApplication
public class BankAccountServiceApplication {

    public static void main(String[] args) {

        SpringApplication.run(BankAccountServiceApplication.class, args);
    }
    @Bean
    CommandLineRunner start(BankAccountRepository bankAccountRepository) {
        return args -> {
            for (int i = 0; i < 10; i++) {
                BankAccount bankAccount = BankAccount.builder()
                        .id(UUID.randomUUID().toString())
                        .type(Math.random() > 0.5 ? AccountType.CURRENT_ACCOUNT : AccountType.SAVING_ACCOUNT)
                        .balance(10000 + Math.random() *60000)
                        .createdAt(new Date())
                        .currency("MAD")
                        .build();
                bankAccountRepository.save(bankAccount);
            }
        };
    }
```

- @SpringBootApplication : Indique que cette classe est l'application principale Spring Boot. Elle active également la configuration automatique, l'analyse des composants, et la configuration de beans.

SELECT * FROM BANK_ACCOUNT;

BALANCE	CREATED_AT	CURRENCY	ID	TYPE
30271.117649080537	2024-07-07 12:10:32.174	MAD	a414c996-8bd5-48a7-bddb-5b993a3c3988	CURRENT_ACCOUNT
69120.72630437696	2024-07-07 12:10:32.27	MAD	ec3ab097-520b-4d9d-8976-b477b40519df	SAVING_ACCOUNT
15081.741497985538	2024-07-07 12:10:32.271	MAD	0c67fc66-2b10-4f14-a4ed-841e7ef2e16c	SAVING_ACCOUNT
21957.906429785253	2024-07-07 12:10:32.272	MAD	44ea3f4f-ddb8-4798-991b-edec9c2547ab	CURRENT_ACCOUNT
65846.20629557275	2024-07-07 12:10:32.273	MAD	dafa8ad5-a3c2-416b-aea5-d7c38af9eeb7	SAVING_ACCOUNT
19262.28740062313	2024-07-07 12:10:32.275	MAD	b3621be9-bfa1-43ea-8d59-c2630f971b79	SAVING_ACCOUNT
39563.11077320091	2024-07-07 12:10:32.276	MAD	0344b30a-5ec7-4677-ab7f-bb79aeef8da5	SAVING_ACCOUNT
28769.024450266363	2024-07-07 12:10:32.277	MAD	64e220f1-e158-4e87-a4e8-ab88ecec6cda	SAVING_ACCOUNT
31778.428250665696	2024-07-07 12:10:32.278	MAD	2fd7a2ca-90bf-413c-9c66-22fc88136f73	SAVING_ACCOUNT
56649.892080958816	2024-07-07 12:10:32.279	MAD	6a93b8de-d99b-4ba0-a794-4ecd71d99bb9	SAVING_ACCOUNT

(10 rows, 3 ms)

Edit

Créer le package web et le controller AccountRestController

- Cette classe AccountRestController gère les opérations RESTful pour les comptes bancaires dans notre application

```java
@RestController
public class AccountRestController {
    3 usages
    private BankAccountRepository bankAccountRepository;
    public AccountRestController(BankAccountRepository bankAccountRepository) {
        this.bankAccountRepository = bankAccountRepository;
    }

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

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

- @RestController : Indique que cette classe est un contrôleur REST, permettant de gérer les requêtes HTTP.

- bankAccounts() : Gère les requêtes GET pour récupérer tous les comptes bancaires.
- bankAccount(String id) : Gère les requêtes GET pour récupérer un compte bancaire spécifique par son identifiant.


```
{
    "id": "7eca5bb6-100d-4a80-84b5-402d1ce32463",
    "createdAt": "2024-07-07T11:17:49.096+00:00",
    "balance": 66607.26871187732,
    "currency": "MAD",
    "type": "SAVING_ACCOUNT"
},
{
    "id": "41273571-bd83-442c-8a7a-6ee9a70be948",
    "createdAt": "2024-07-07T11:17:49.207+00:00",
    "balance": 56483.81325815492,
    "currency": "MAD",
    "type": "SAVING_ACCOUNT"
},
{
    "id": "f31fe06f-6daf-40b2-ac5d-a1b3229bb315",
    "createdAt": "2024-07-07T11:17:49.209+00:00",
    "balance": 66395.02066731907,
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT"
},
{
    "id": "0eb64e5e-8981-4c4c-9ea9-03f3e5265895",
    "createdAt": "2024-07-07T11:17:49.210+00:00",
    "balance": 48128.464245151714,
    "currency": "MAD",
    "type": "SAVING_ACCOUNT"
},
{
    "id": "cadf4cb4-e984-4cd0-b8fc-f1a1dc8bd880",
    "createdAt": "2024-07-07T11:17:49.211+00:00",
    "balance": 18595.99346822522,
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT"
},
{
```

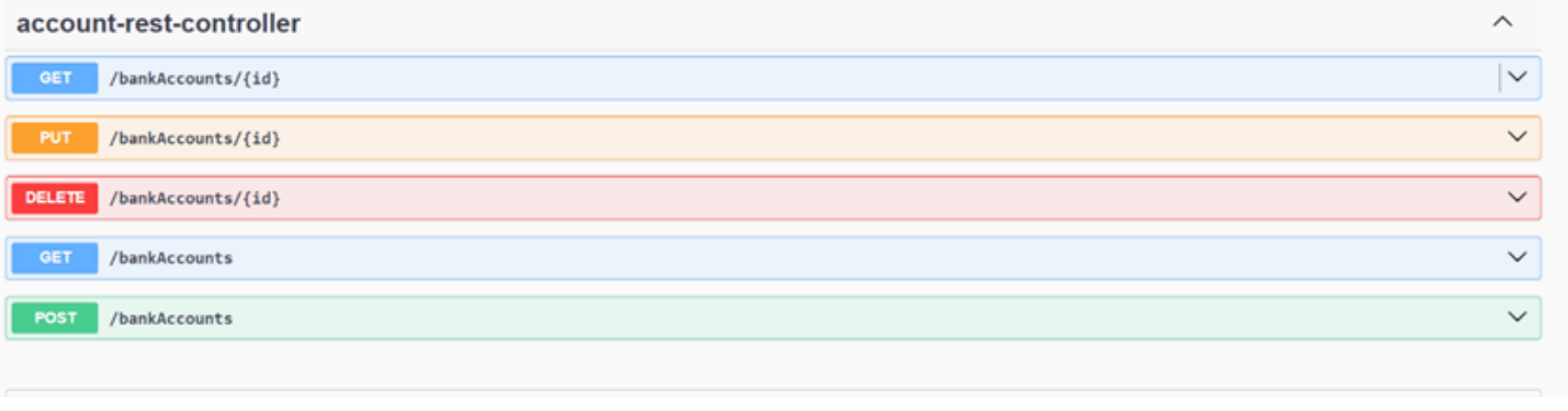

- save(BankAccountRequestDTO requestDTO) : Gère les requêtes POST pour créer un nouveau compte bancaire.
- update(String id, BankAccount bankAccount) : Gère les requêtes PUT pour mettre à jour un compte bancaire existant.
- deleteBankAccount(String id) : Gère les requêtes DELETE pour supprimer un compte bancaire par son identifiant.

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

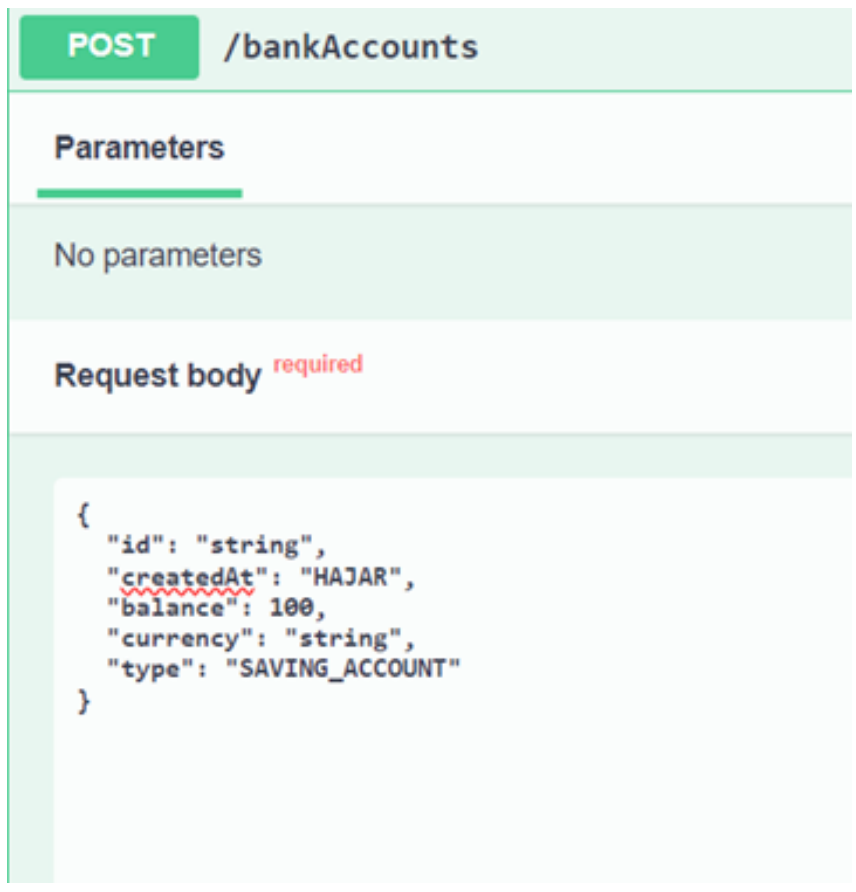
```
@PutMapping("/bankAccounts/{id}")
public BankAccount update(@PathVariable String id, @RequestBody BankAccount bankAccount) {
    BankAccount account = bankAccountRepository.findById(id).orElseThrow();
    if (bankAccount.getBalance()!= null) account.setBalance(bankAccount.getBalance());
    if (bankAccount.getCreatedAt()!= null) account.setCreatedAt(new Date());
    if (bankAccount.getType()!= null) account.setType(bankAccount.getType());
    if (bankAccount.getCurrency()!= null) account.setCurrency(bankAccount.getCurrency());
    return bankAccountRepository.save(account);
}

@DeleteMapping("/bankAccounts/{id}")
public void deleteBankAccount(@PathVariable String id) {
    bankAccountRepository.deleteById(id);
}
```

Ajouter les dependence pour le swagger

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



On test la méthode post :




```
  "id": "string",
  "createdAt": "HAJAR",
  "balance": 100,
  "currency": "string",
  "type": "SAVING_ACCOUNT"
}'
```

Pour utiliser spring database :


```
{
    "_embedded": {
        "bankAccounts": [
            {
                "createdAt": "2024-07-07T12:13:53.719+00:00",
                "balance": 51734.730556111215,
                "currency": "MAD",
                "type": "CURRENT_ACCOUNT",
                "_links": {
                    "self": {
                        "href": "http://localhost:8081/bankAccounts/95ac0606-c87f-4f78-a8a9-198133fe0203"
                    },
                    "bankAccount": {
                        "href": "http://localhost:8081/bankAccounts/95ac0606-c87f-4f78-a8a9-198133fe0203"
                    }
                }
            },
            {
                "createdAt": "2024-07-07T12:13:53.819+00:00",
```


Creer l’interface AccountProjection

- Cette interface AccountProjection utilise la projection Spring Data REST pour exposer des vues personnalisées des entités BankAccount
- @Projection(types = BankAccount.class, name = "p1") : Définit une projection pour l'entité BankAccount avec le nom "p1". Cela permet de sélectionner et d'exposer uniquement certains attributs de l'entité lors des requêtes REST

```java
package ma.bouchama.bankaccountservice.entities;

import ma.bouchama.bankaccountservice.enums.AccountType;
import org.springframework.data.rest.core.config.Projection;

no usages
@Projection(types = BankAccount.class, name = "p1")
public interface AccountProjection {
    no usages
    public String getId();
    no usages
    public AccountType getType();
}
```

- Si je tape localhost:8081/bankAccounts: il me donner touts les attributs


```
"bankAccounts": [
    {
        "createdAt": "2024-07-07T12:28:05.832+00:00",
        "balance": 66328.66878388794,
        "currency": "MAD",
        "type": "SAVING_ACCOUNT",
        "_links": {
            "self": {
                "href": "http://localhost:8081/bankAccounts/56503f45-53bb-413a-9bde-13255e0637d8"
            },
            "bankAccount": {
                "href": "http://localhost:8081/bankAccounts/56503f45-53bb-413a-9bde-13255e0637d8{?projection}",
                "templated": true
            }
        }
    },
    {
```


- Si je tape localhost:8081/bankAccounts?projection=p1: il me donne seulement le id et le type


```
bedded": {
"bankAccounts": [
    {
        "id": "56503f45-53bb-413a-9bde-13255e0637d8",
        "type": "SAVING_ACCOUNT",
        "_links": {
            "self": {
                "href": "http://localhost:8081/bankAccounts/56503f45-53bb-413a-9bde-13255e0637d8"
            },
            "bankAccount": {
                "href": "http://localhost:8081/bankAccounts/56503f45-53bb-413a-9bde-13255e0637d8{?projection}",
                "templated": true
            }
        }
    },
```


Créer le package service:

```java
import ma.bouchama.bankaccountservice.dto.BankAccountRequestDTO;
import ma.bouchama.bankaccountservice.dto.BankAccountResponseDTO;

no usages
public interface AccountService {
    no usages
    public BankAccountResponseDTO addAccount(BankAccountRequestDTO bankAccountDTO);
}
```

Créer le package dto:

- **BankAccountRequestDTO** : Cette classe simplifie le transfert des données nécessaires pour créer ou mettre à jour un compte bancaire, en encapsulant les attributs pertinents et en utilisant Lombok pour réduire le code boilerplate.

```java
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import ma.bouchama.bankaccountservice.enums.AccountType;
2 usages
@Data @NoArgsConstructor @AllArgsConstructor @Builder
public class BankAccountRequestDTO {
    private Double balance;
    private String currency;
    private AccountType type;
}
```

- **BankAccountResponseDTO** : Cette classe encapsule les informations de réponse de l'API concernant les comptes bancaires, facilitant ainsi le transfert des données pertinentes au client tout en utilisant Lombok pour réduire le code répétitif.

```java
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class BankAccountResponseDTO {
    private String id;
    private Date createdAt;
    private Double balance;
    private String currency;
    private AccountType type;
}
```

Créer l'implementation de l'interface AccountService :

- Cette classe AccountServiceImpl implémente le service AccountService et gère la logique métier pour les opérations liées aux comptes bancaires

```java
@Transactional
public class AccountServiceImpl implements AccountService {
    @Autowired
    private BankAccountRepository bankAccountRepository;
    no usages
    @Override
    public BankAccountResponseDTO addAccount(BankAccountRequestDTO bankAccountDTO) {
        BankAccount bankAccount = BankAccount.builder()
                .id(UUID.randomUUID().toString())
                .createdAt(new Date())
                .balance(bankAccountDTO.getBalance())
                .type(bankAccountDTO.getType())
                .currency(bankAccountDTO.getCurrency())
                .build();
        BankAccount saveBankAccount = bankAccountRepository.save(bankAccount);
        BankAccountResponseDTO bankAccountResponseDTO = BankAccountResponseDTO.builder()
                .id(saveBankAccount.getId())
                .type(saveBankAccount.getType())
                .createdAt(saveBankAccount.getCreatedAt())
                .currency(saveBankAccount.getCurrency())
                .balance(saveBankAccount.getBalance())
                .build();
        return bankAccountResponseDTO;
    }
}
```

- @Service : Indique que cette classe est un service Spring, une couche de la logique métier.
- @Transactional : Assure que les méthodes de cette classe sont exécutées dans une transaction, garantissant la cohérence des données.
- @Autowired : Injection automatique du dépôt de comptes bancaires (BankAccountRepository).

Modifier la methode post :

```
@PostMapping("/bankAccounts")
public BankAccountResponseDTO save(@RequestBody BankAccountRequestDTO requestDTO) {

    return accountService.addAccount(requestDTO);
}
```

2. 1. MIcro-Service avec Web Service RESTFUL

Dans le dossier graphql on crée le fichier schema.graphqls

- Ce schéma GraphQL structure les interactions avec le système de gestion des comptes bancaires, permettant des opérations CRUDsur les comptes bancaires et les clients

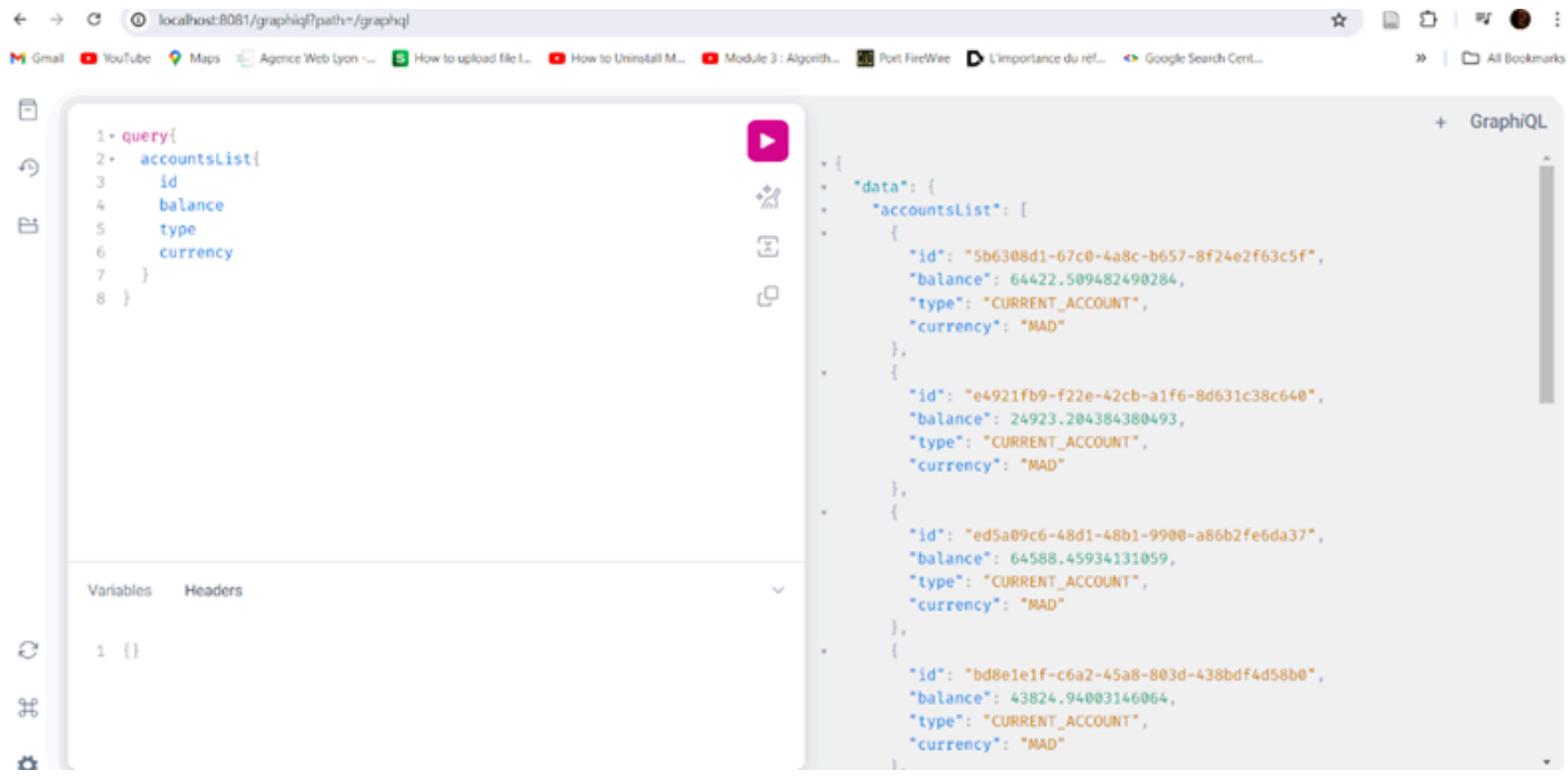
accountsList : Récupère la liste de tous les comptes bancaires**.**

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

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

créer la classe BankAccountGraphQLController

```
import java.util.List;

@Controller
public class BankAccountGraphQLController {

    @Autowired
    private BankAccountRepository bankAccountRepository;

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

- graphiql nous permet d’envoyer des requetes et de recevoir les reponse selon nos besoins :



chercher un compte par son ID :

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

```
bankAccountById (id:String) : BankAccount
```

```
query{
  bankAccountById (id:"d2ed3e83-fd9f-455b-a096-0b0e41238ce3"){
    balance
    type
    currency
  }
}
```

```
{
  "data": {
    "bankAccountById": {
      "balance": 48371.12706985498,
      "type": "CURRENT_ACCOUNT",
      "currency": "MAD"
    }
  }
}
```

crée le package exceptions :

- pour gérer et bien afficher le message d’erreurs

```
@Component
public class CustomDataFetcherExceptionResolver extends DataFetcherExceptionResolverAdapter {
    no usages
    @Override
    protected GraphQLError resolveToSingleError(Throwable ex, DataFetchingEnvironment env) {

        return new GraphQLError() {

            @Override
            public String getMessage() {
                return ex.getMessage();
            }
            no usages
            @Override
            public List<SourceLocation> getLocations() {
                return null;
            }
            no usages
            @Override
            public ErrorClassification getErrorType() {
                return null;
            }
```

```
query{

  bankAccountById (id:"hajar"){
    balance
    type
    currency
  }
}
```

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

Pour la methode d’ajout

- BankAccountDTO : Décrit les données nécessaires pour créer ou mettre à jour un compte bancaire, incluant le solde (balance), la devise (currency), et le type (type).

```
type Mutation{
    addAccount(bankAccount : BankAccountDTO) : BankAccount
}
input BankAccountDTO {
    balance: Float,
    currency: String,
    type: String

}
```

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

```
mutation{
  addAccount(bankAccount:{
    balance : 10000
    type: "SAVING_ACCOUNT"
    currency : "USD"
  }){
    id
    type
    currency
    balance
  }
}
```

```
{
  "data": {
    "addAccount": {
      "id": "9350a17e-bd28-4d17-b19e-fe2e28bd0d01",
      "type": "SAVING_ACCOUNT",
      "currency": "USD",
      "balance": 10000
    }
  }
}
```

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

```
{
  "data": {
    "addAccount": {
      "id": "e13c8d05-31ab-417a-9401-2b84fcef23ca",
      "type": "CURRENT_ACCOUNT",
      "balance": 40000
    }
  }
}
```

ables Headers

```
{"t": "CURRENT_ACCOUNT","b": 40000,"c": "MAD"}
```

modification d’un compte :

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

- Cette methode met à jour les détails d'un compte bancaire identifié par id avec les informations fournies dans bankAccountDTO. Il crée un nouvel objet BankAccount, l'enregistre en base de données via bankAccountRepository.save, puis construit et renvoie un objet BankAccountResponseDTO contenant les détails mis à jour du compte.

```
public BankAccountResponseDTO updateAccount(String id, BankAccountRequestDTO bankAccountDTO) {
    BankAccount bankAccount = BankAccount.builder()
            .id(id)
            .createdAt(new Date())
            .balance(bankAccountDTO.getBalance())
            .type(bankAccountDTO.getType())
            .currency(bankAccountDTO.getCurrency())
            .build();
    BankAccount saveBankAccount = bankAccountRepository.save(bankAccount);
    BankAccountResponseDTO bankAccountResponseDTO = BankAccountResponseDTO.builder()
            .id(saveBankAccount.getId())
            .type(saveBankAccount.getType())
            .createdAt(saveBankAccount.getCreatedAt())
            .currency(saveBankAccount.getCurrency())
            .balance(saveBankAccount.getBalance())
            .build();
    return bankAccountResponseDTO;
}
```

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

riables Headers

```
{"i": "56f8bc31-7c88-4c40-ad2e-d0243ee00eb4", "t": "CURRENT_ACCOUNT","b"
```

```
"accountsList": [
  {
    "id": "56f8bc31-7c88-4c40-ad2e-d0243ee00eb4",
    "balance": 48414.60951061646,
    "type": "SAVING_ACCOUNT",
    "currency": "MAD"
  },
```

```
{
  "data": {
    "updateAccount": {
      "id": "56f8bc31-7c88-4c40-ad2e-d0243ee00eb4",
      "type": "CURRENT_ACCOUNT",
      "balance": 40000
    }
  }
```

la suppression d’un compte :

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

```
mutation {
    deleteAccount(id:"56f8bc31-7c88-4c40-ad2e-d0243ee00eb4")

    }

```

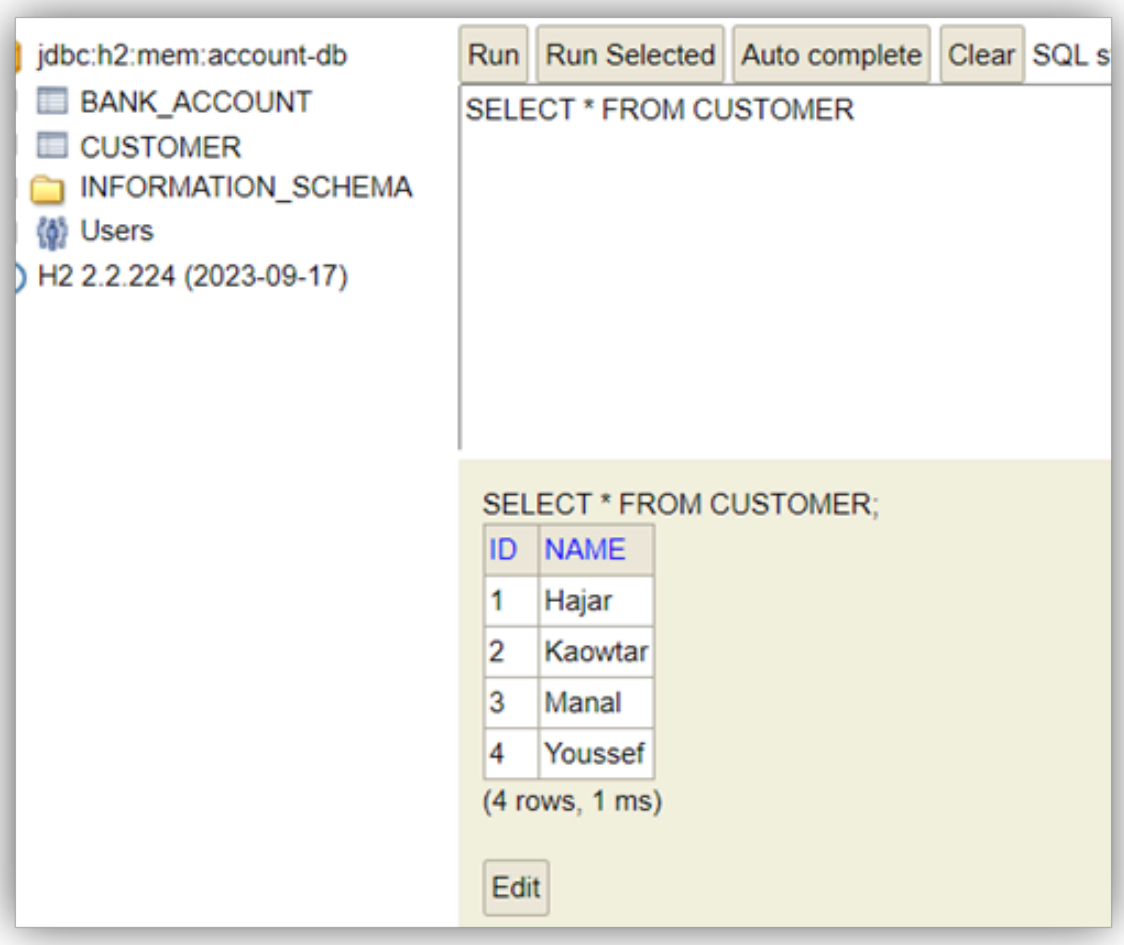
```
{
  "data": {
    "deleteAccount": true
  }
}
```

Créer la classe Customer

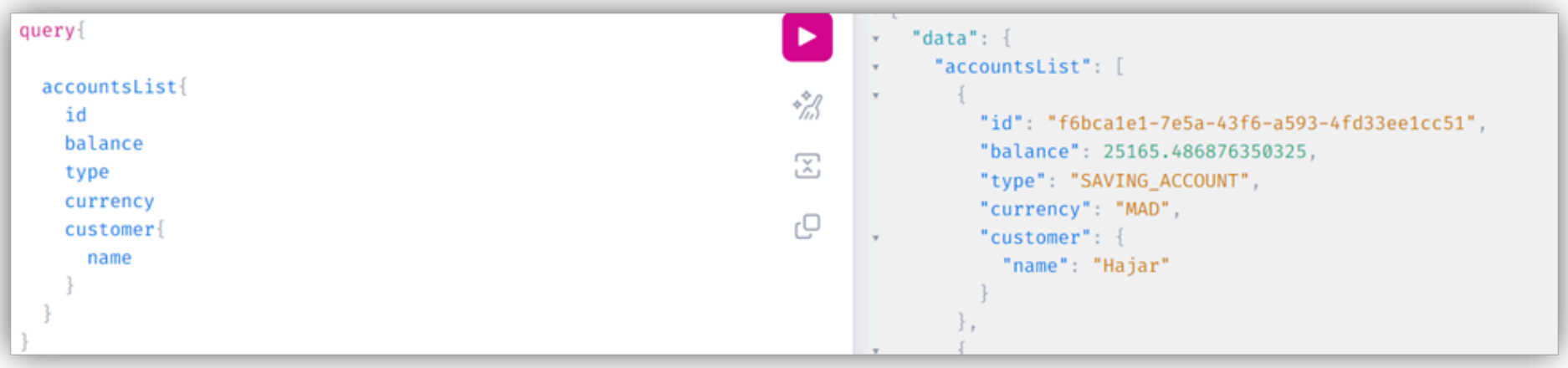
```java
3 usages
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
@Builder
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @OneToMany(mappedBy = "customer")
    private List<BankAccount> bankAccounts;
}
```

- configure une tâche de démarrage (CommandLineRunner) pour initialiser des données dans les repositories bankAccountRepository et customerRepository. Il crée plusieurs clients

```java
CommandLineRunner start(BankAccountRepository bankAccountRepository, CustomerRepository customerRepository) {
    return args -> {
        Stream.of( ...values: "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.SAVING_ACCOUNT)
                        .balance(10000 + Math.random() *60000)
                        .createdAt(new Date())
                        .currency("MAD")
                        .customer(customer)
                        .build();
                bankAccountRepository.save(bankAccount);
            }
```

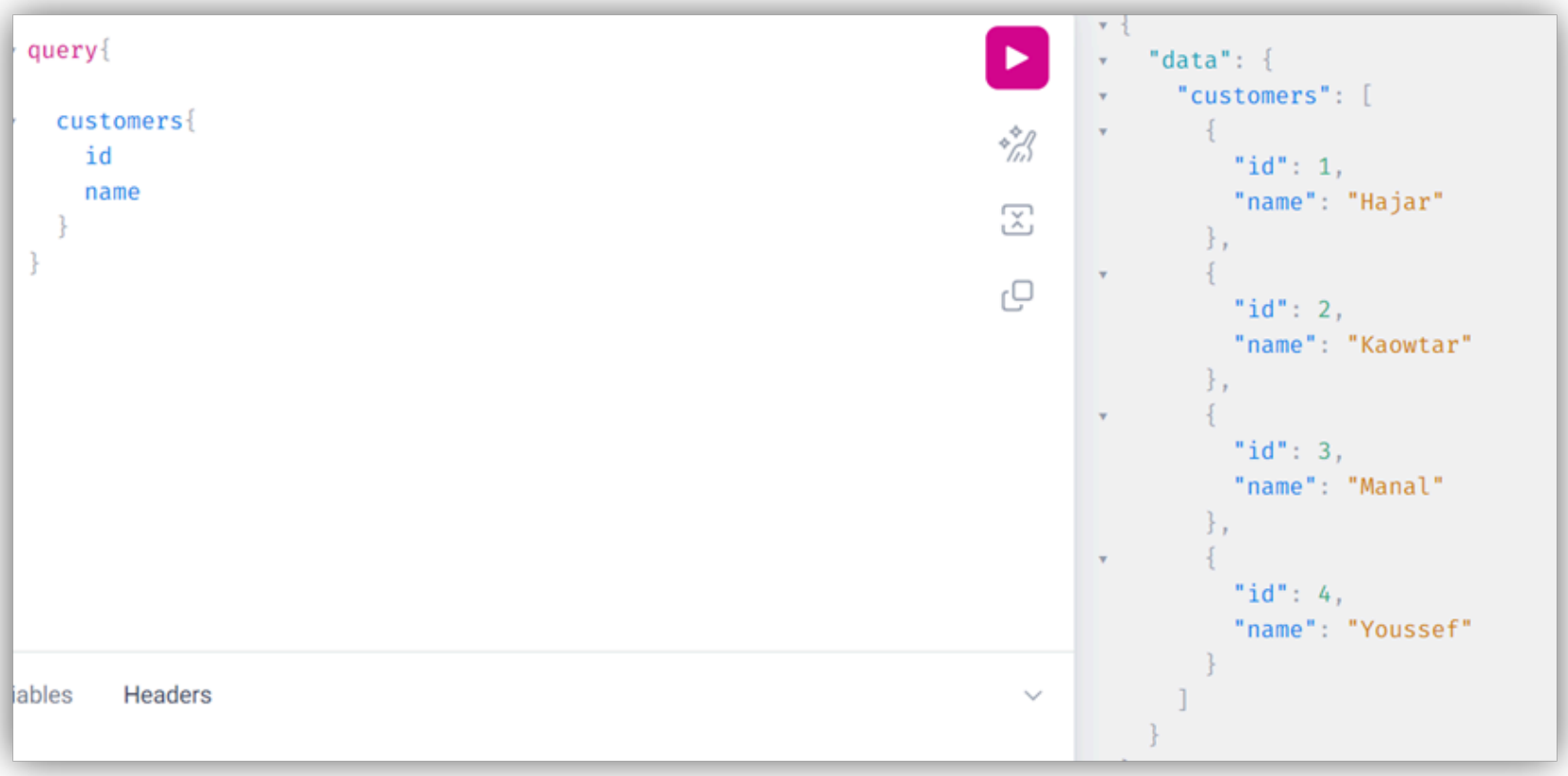



```
type Customer{
    id: Float,
    name:String
    bankAccount:[BankAccount]
}

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



afficher la liste des customers

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



Utiliser json pour evité le problème de boucle infini

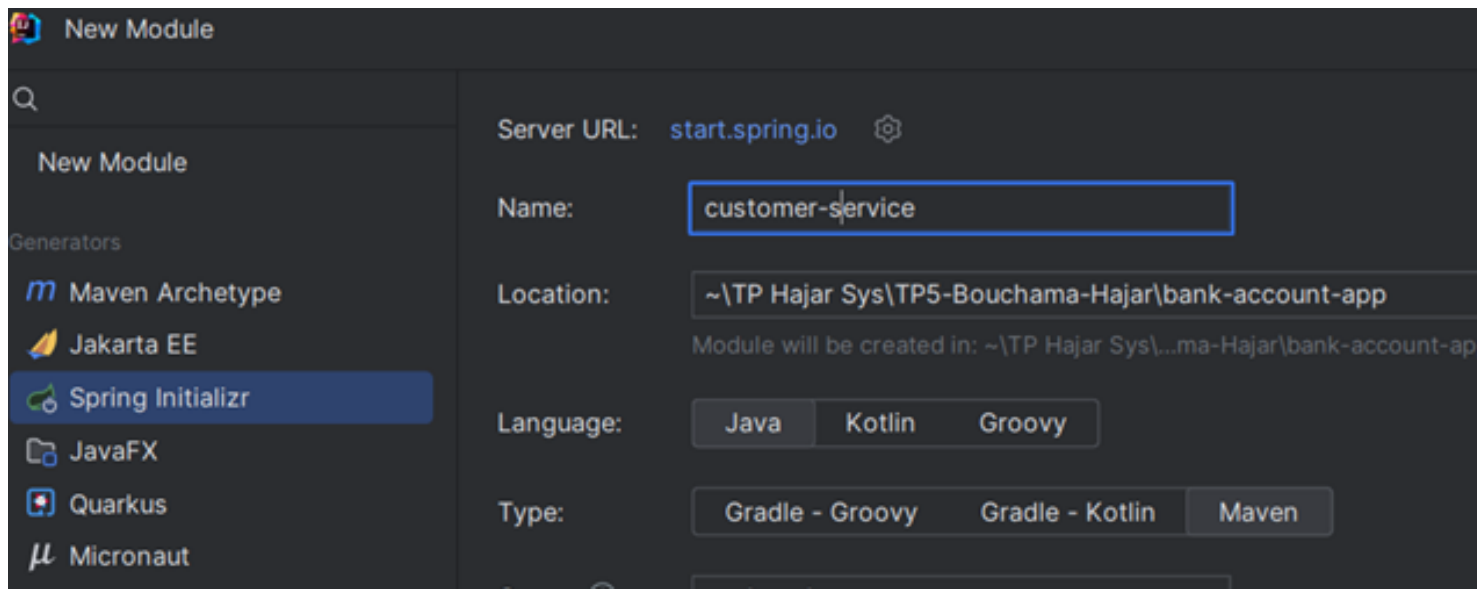
```
@JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
private List<BankAccount> bankAccounts;
```

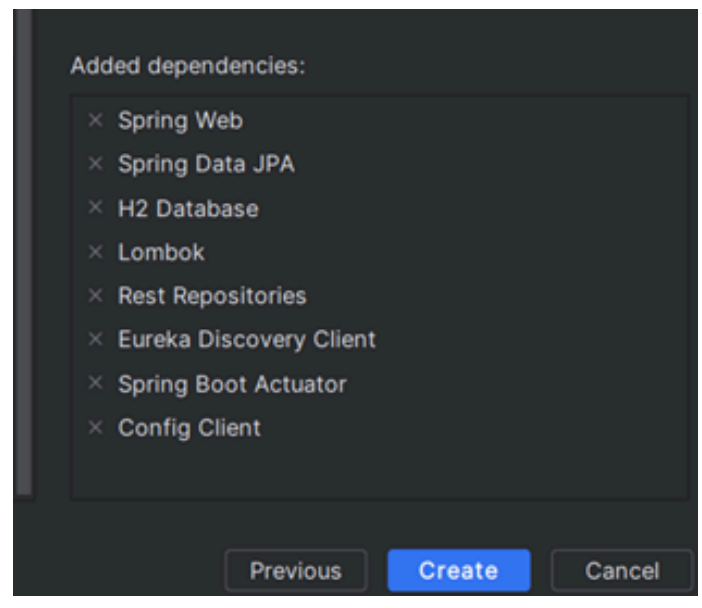
Partie 2

Créer une application basée sur les micro-services qui permet des gérer des clients et des comptes bancaire. chaque compte appartient à un client.

1. Créer le micro-service Customer Service :

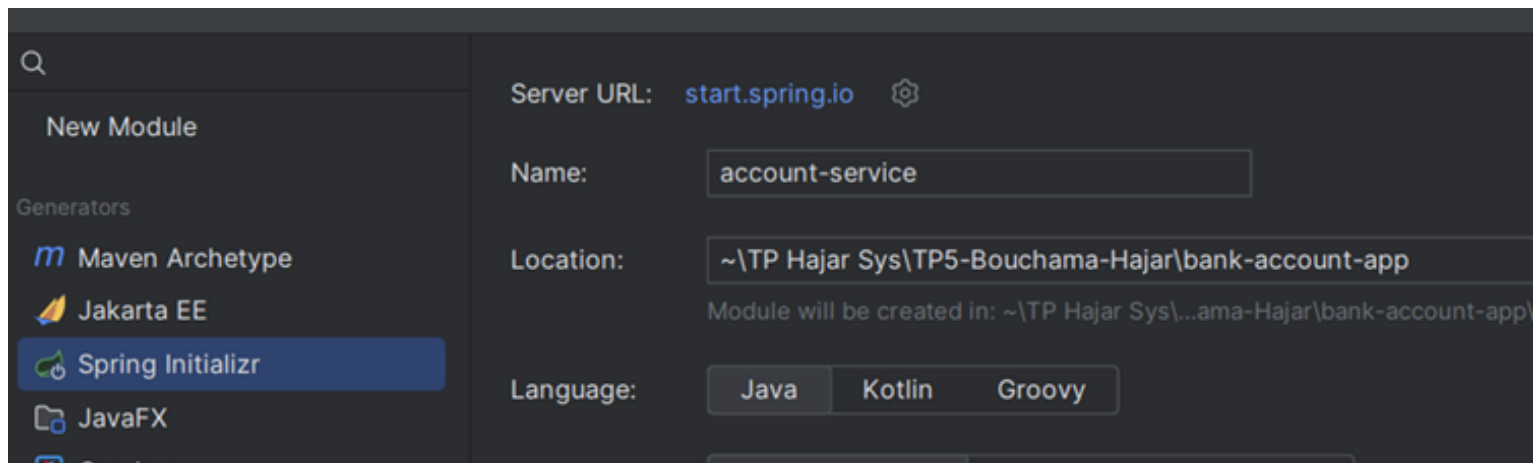
Micro-service dédié à la gestion des informations clients.

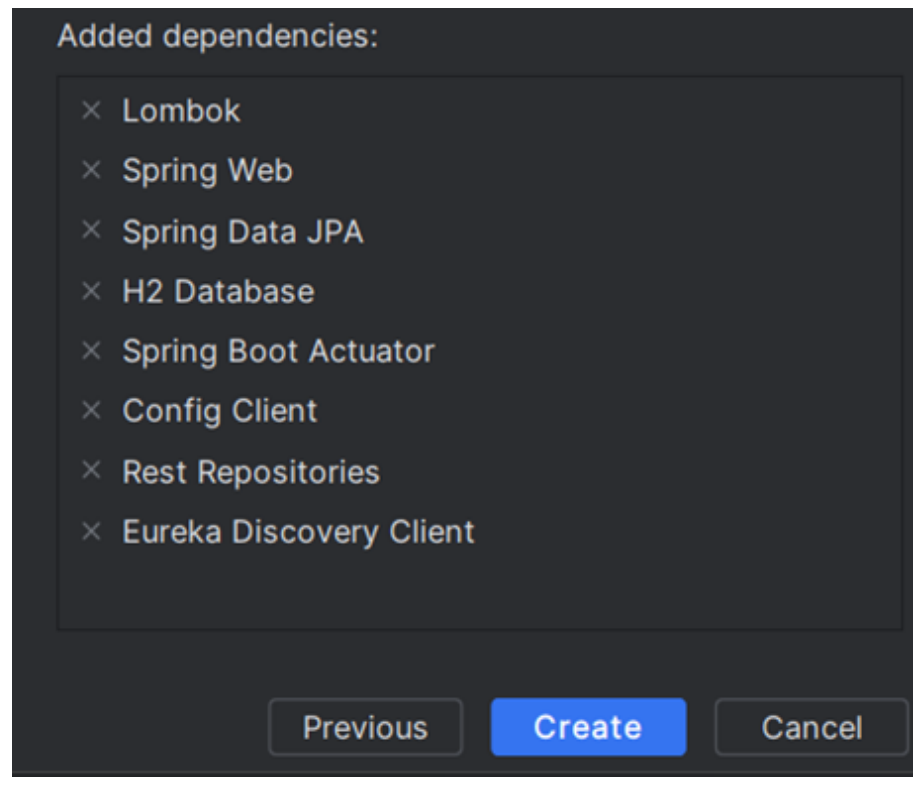




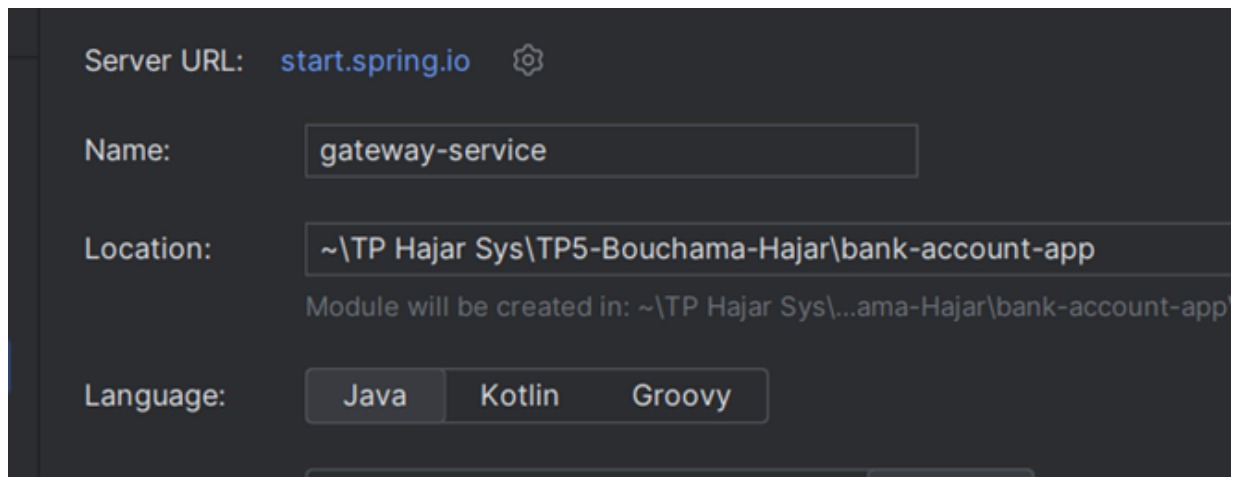
2. Créer le micro-service Account Service :

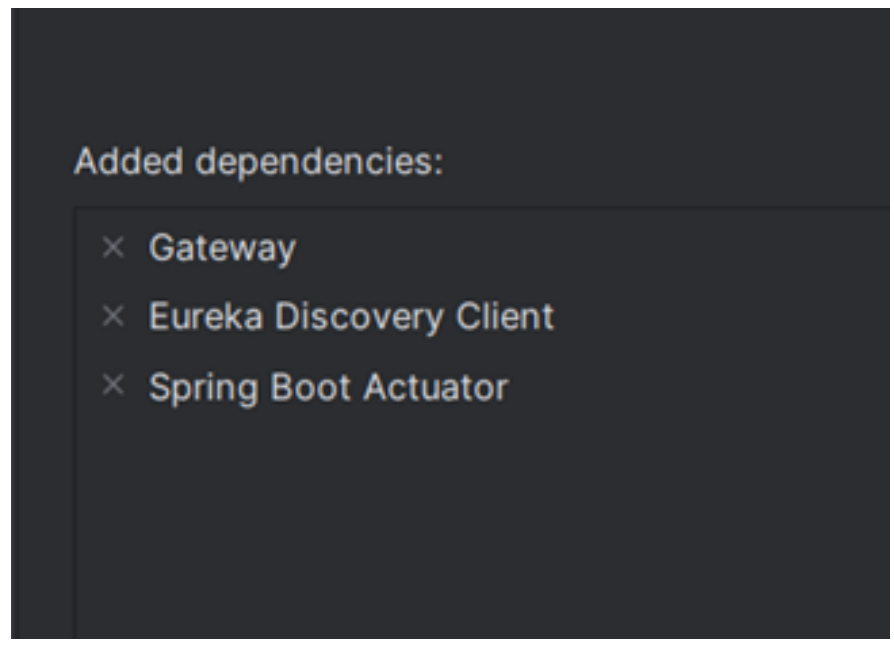
Micro-service dédié à la gestion des comptes bancaires.





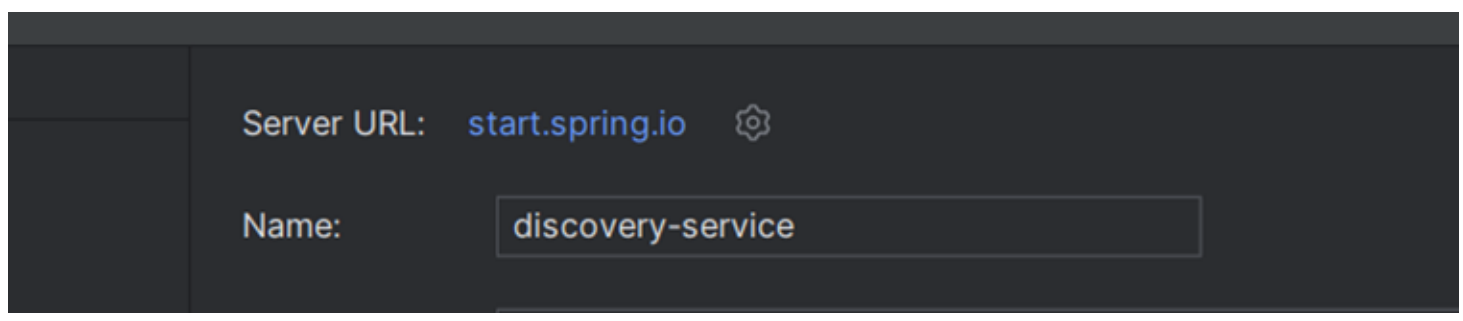
3. Créer le Micro-service Gateway Service :

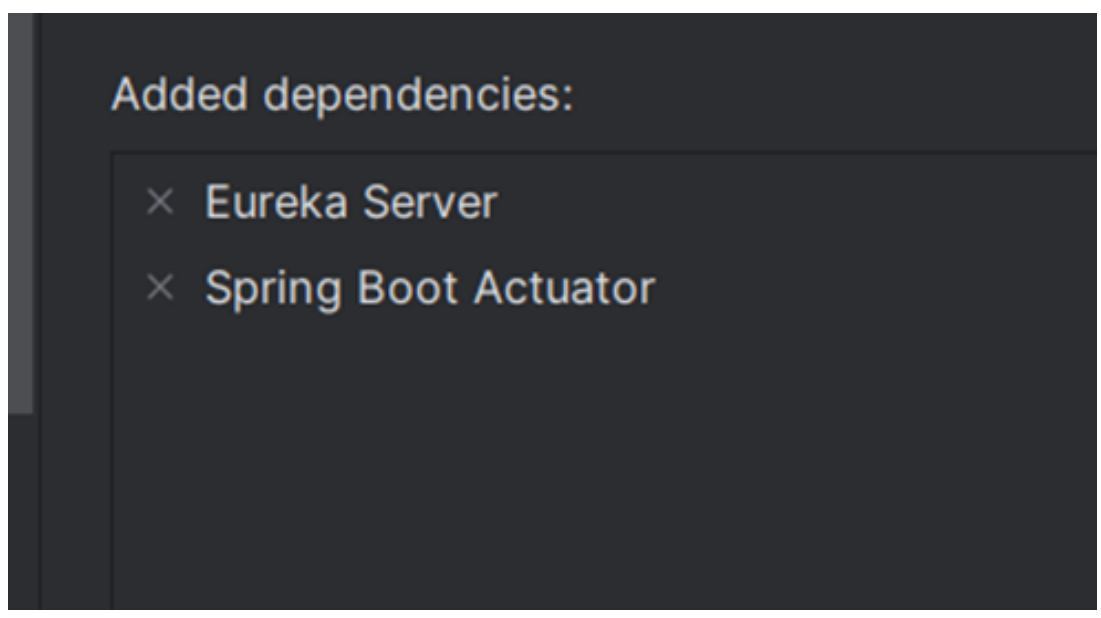




4. Créer le micro-service Discovery Service :

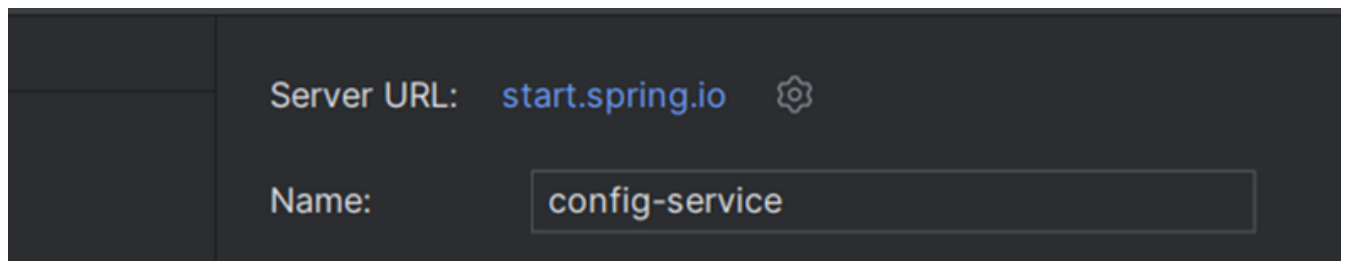
Service de découverte des micro-services pour faciliter la communication entre eux.

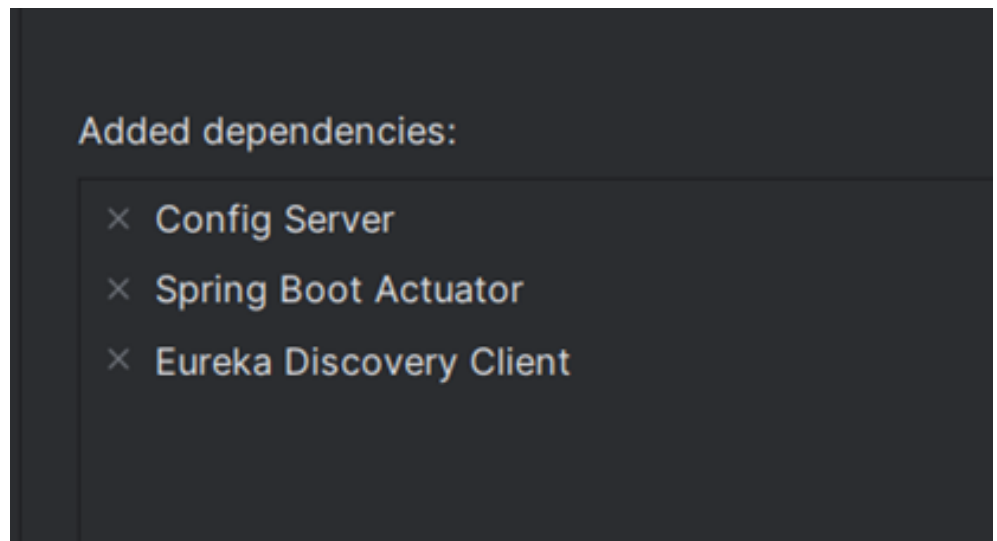




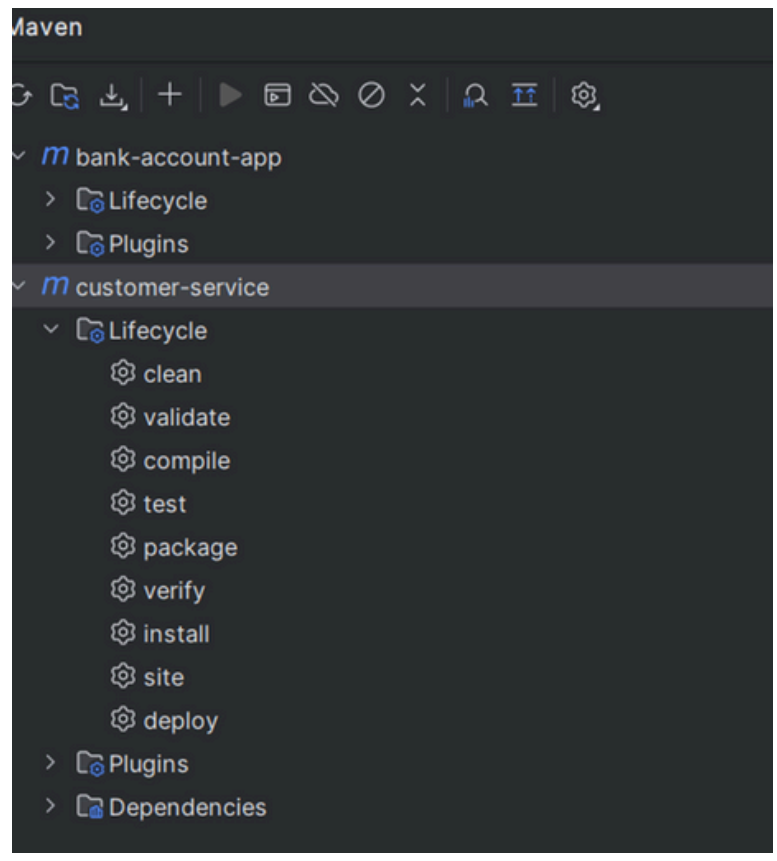
2. Créer le micro-service Config Service :

Service de gestion de la configuration centralisée pour tous les micro-services.

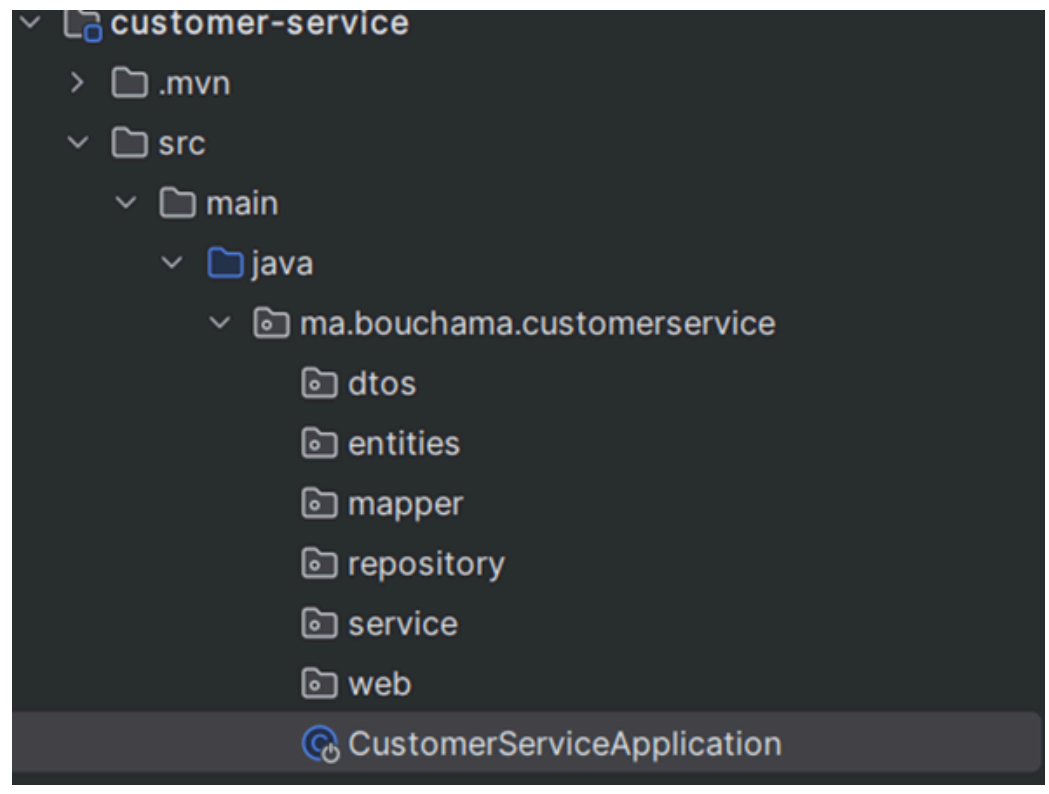




Ajouter les micro service autant que maven :



Creer les package suivant dans customer-service



Dans le package entities :

Créer la class Customer :

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

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

Dans le package repo :

Créer l'interface CustomerRepository :

```java
import ma.bouchama.customerservice.entities.Customer;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource
public interface CustomerRepository extends JpaRepository<Customer, Long> {

}
```

Dans le package web :

Créer la class CustomerRestController :

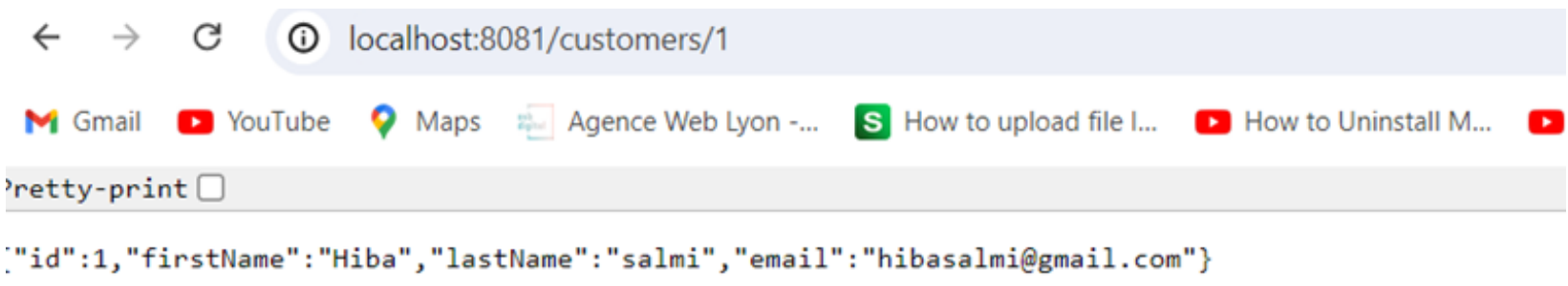
```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
@RefreshScope
public class CustomerRestController {

    3 usages
    private CustomerRepository customerRepository;

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

}
```

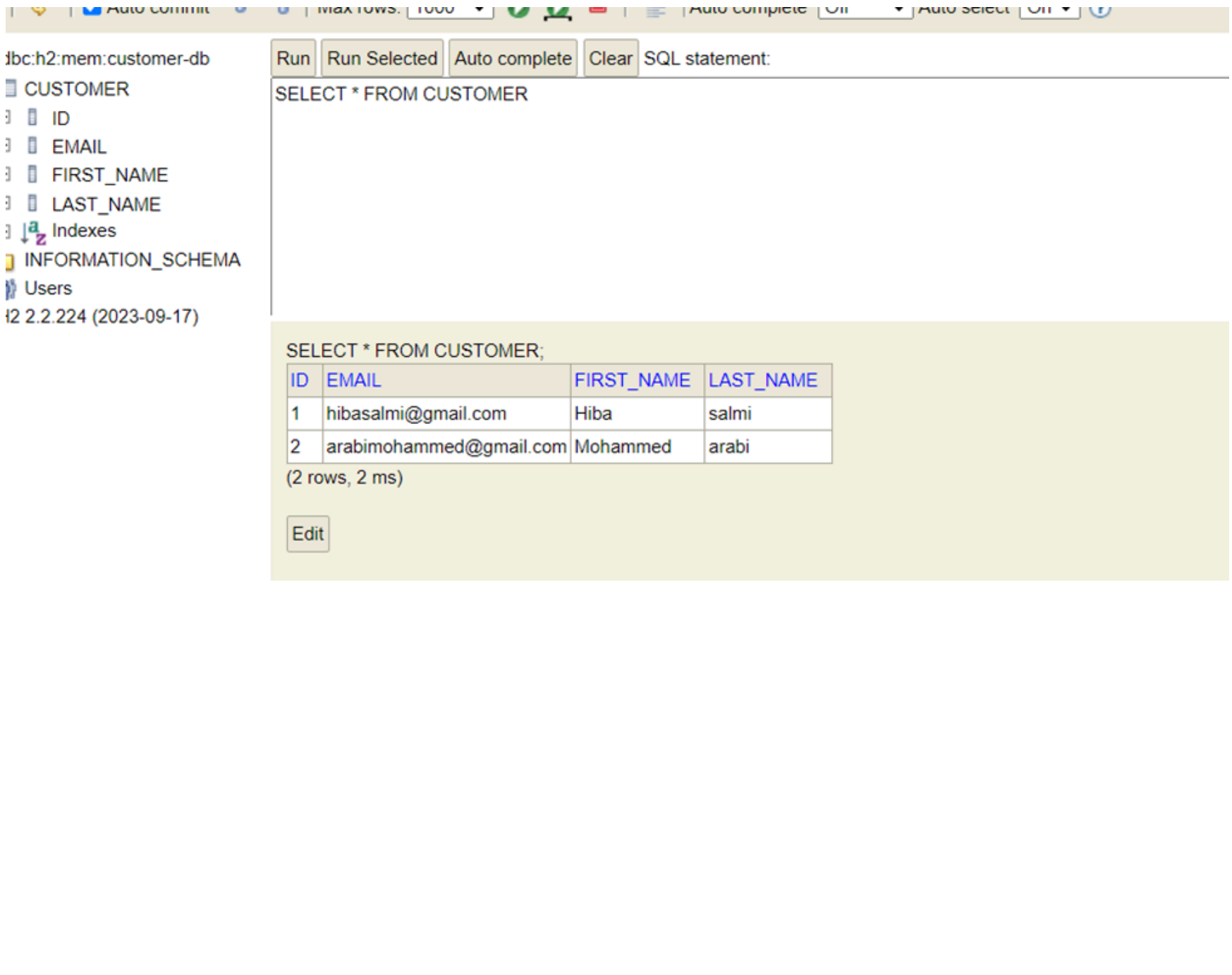
Dans la classe CustomerServiceApplication :

```java
public static void main(String[] args) {

    SpringApplication.run(CustomerServiceApplication.class, args);
}
@Bean
CommandLineRunner start(CustomerRepository customerRepository){
    return args -> {
        List<Customer> customerList=List.of(
                Customer.builder()
                        .firstName("Hiba")
                        .lastName("salmi")
                        .email("hibasalmi@gmail.com")
                        .build(),
                Customer.builder()
                        .firstName("Mohammed")
                        .lastName("arabi")
                        .email("arabimohammed@gmail.com")
                        .build()

        );
        customerRepository.saveAll(customerList);



    };
}
```

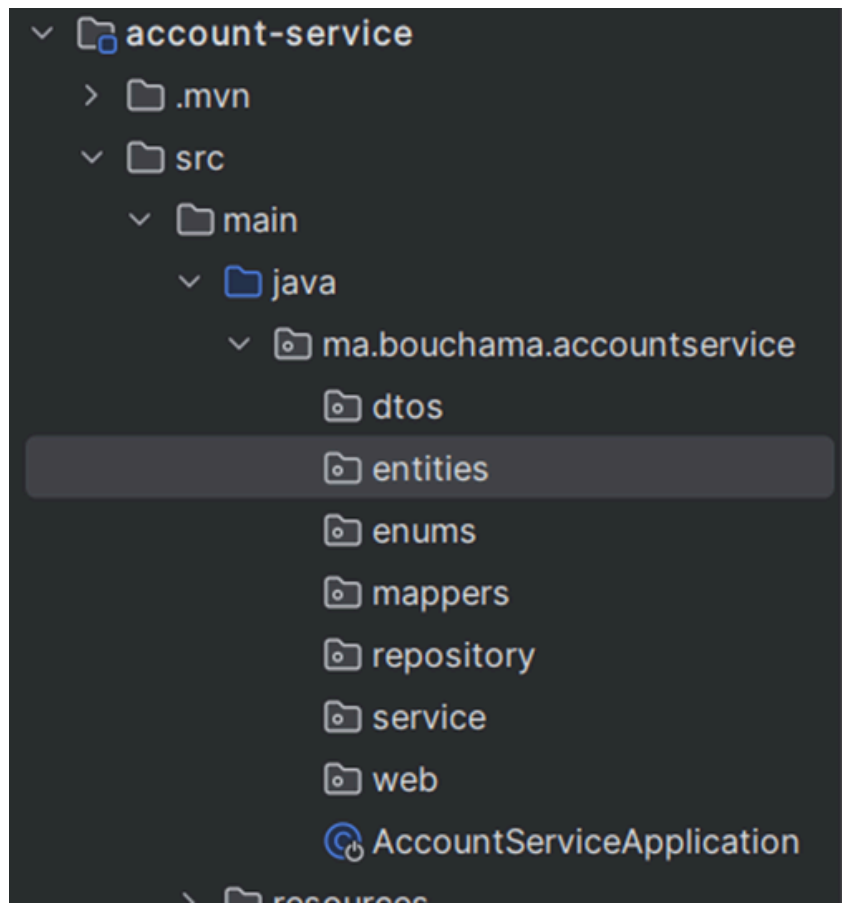
application.proprties :

```
spring.application.name=customer-service
spring.cloud.discovery.enabled=false
server.port=8081
spring.datasource.url=jdbc:h2:mem:customer-db
spring.h2.console.enabled=true
spring.cloud.config.enabled=false
```





Dans la micro service account-service créer les packages suivant :



Dans le package entities :

Créer une classe BankAcccount

```java
import jakarta.persistence.*;
import lombok.*;
import ma.bouchama.accountservice.enums.AccountType;
import ma.bouchama.accountservice.model.Customer;

import java.time.LocalDate;
no usages
@Entity
@Getter @Setter @ToString @NoArgsConstructor @AllArgsConstructor @Builder
public class BankAccount {
    @Id
    private String accountId;
    private double balance;
    private LocalDate createAt;
    private String currency;
    @Enumerated(EnumType.STRING)
    private AccountType type;
    @Transient
    private Customer customer;
    private Long customerId;
}
```

Dans le package enums :

Créer AccountType

```java
2 usages
public enum AccountType {
    no usages
    CURRENT_ACCOUNT, SAVING_ACCOUNT
}
```

Créer un package model :

Créer la classe Customer :

```java
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;

2 usages
@Getter
@Setter
@ToString

public class Customer {
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
}
```

Dans le package web :

Créer une class AccountRestController :

```java
import java.util.List;
@RestController
public class AccountRestController {
    3 usages
    private BankAccountRepository accountRepository;

    public AccountRestController(BankAccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }

    @GetMapping("/accounts")
    public List<BankAccount> accountList() {
        return accountRepository.findAll();
    }

    @GetMapping("/accounts/{id}")
    public BankAccount bankAccountById(@PathVariable String id) {
        return accountRepository.findById(id).get();
    }
}
```

Dans notre classe d’application :

AccountServiceApplication:

```java
@SpringBootApplication
public class AccountServiceApplication {

    public static void main(String[] args) {

        SpringApplication.run(AccountServiceApplication.class, args);
    }
    @Bean
    CommandLineRunner commandLineRunner(BankAccountRepository accountRepository) {
        return args -> {
            BankAccount bankAccount1 = BankAccount.builder()
                    .accountId(UUID.randomUUID().toString())
                    .currency("MAD")
                    .balance(98080)
                    .createAt(LocalDate.now())
                    .type(AccountType.CURRENT_ACCOUNT)
                    .customerId(Long.valueOf( l: 1))
                    .build();
            BankAccount bankAccount2 = BankAccount.builder()
                    .accountId(UUID.randomUUID().toString())
                    .currency("MAD")
                    .balance(100000)
                    .createAt(LocalDate.now())
                    .type(AccountType.SAVING_ACCOUNT)
                    .customerId(Long.valueOf( l: 2))
                    .build();
            accountRepository.save(bankAccount1);
            accountRepository.save(bankAccount2);
        };
}
```

Dans application.properties :

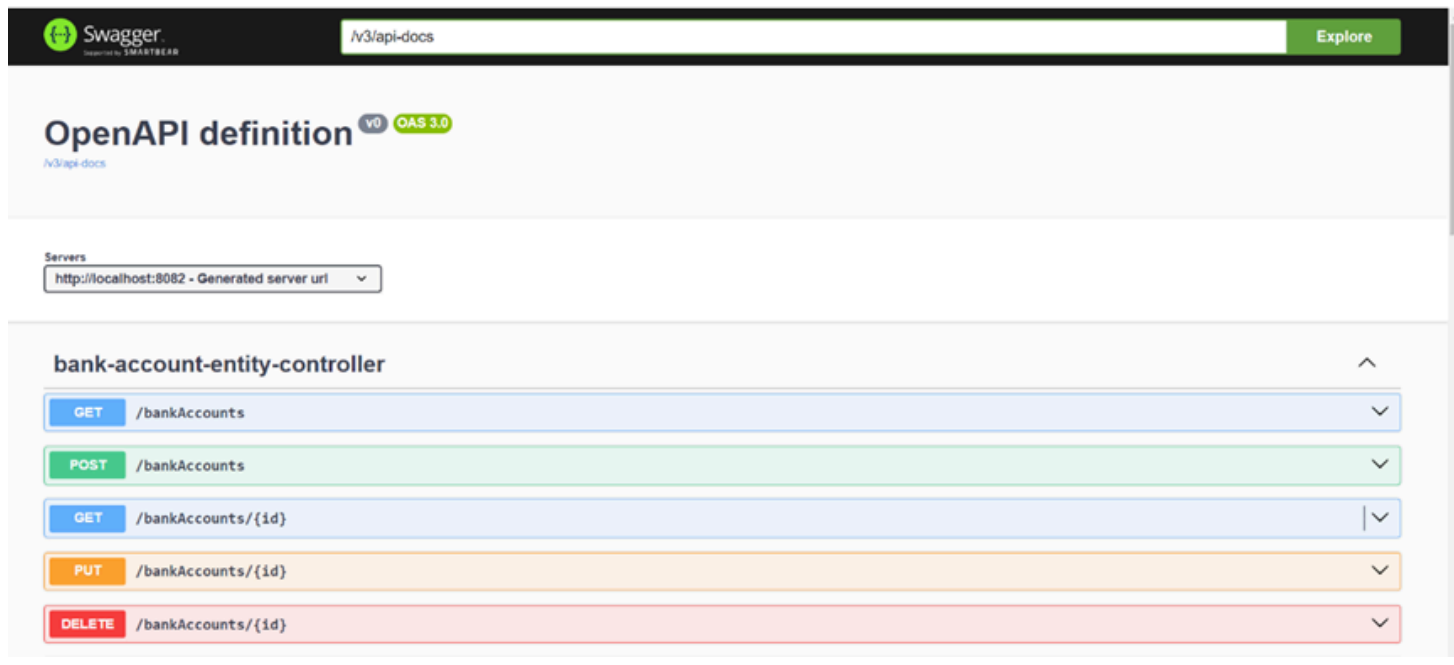
Dans application.properties :

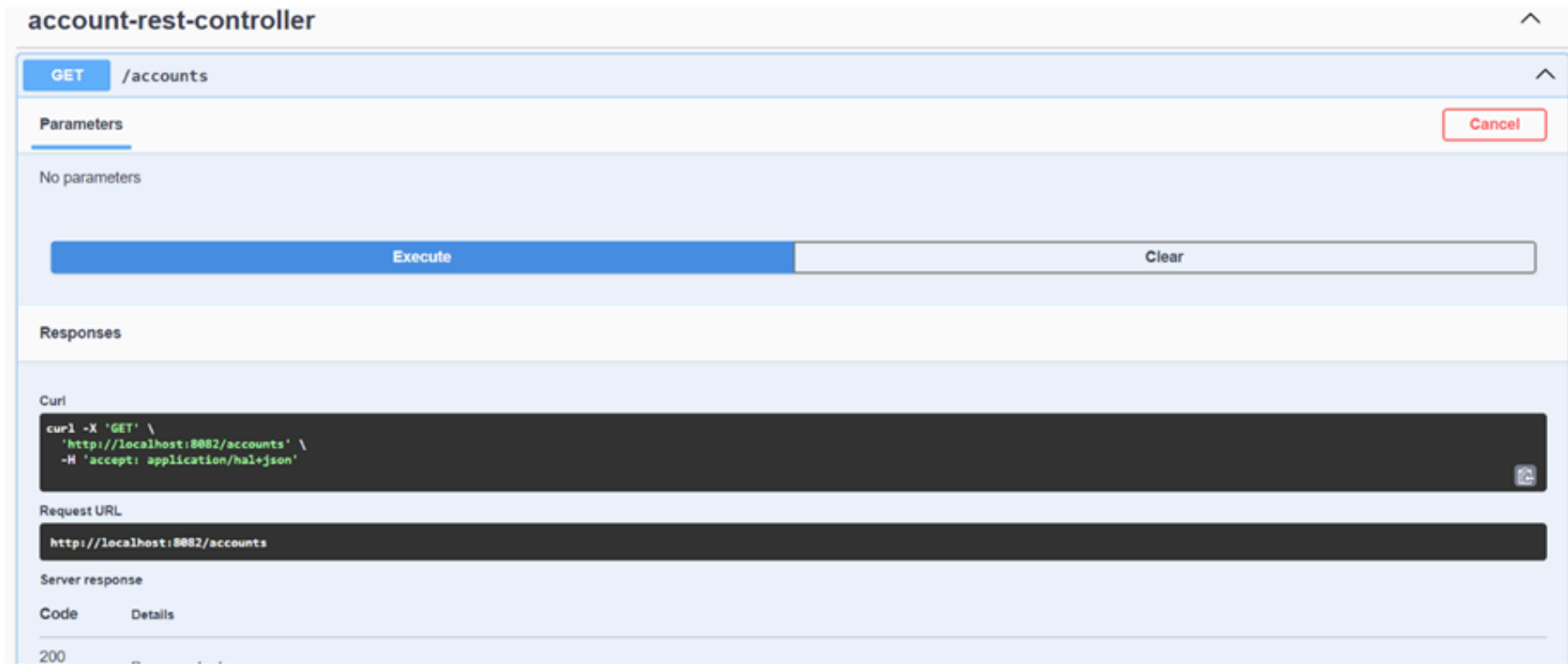
```
spring.application.name=account-service
spring.cloud.discovery.enabled=false
spring.datasource.url=jdbc:h2:mem:account-db
spring.h2.console.enabled=true
server.port=8082
spring.cloud.config.enabled=false
```

Ajouter a pom :

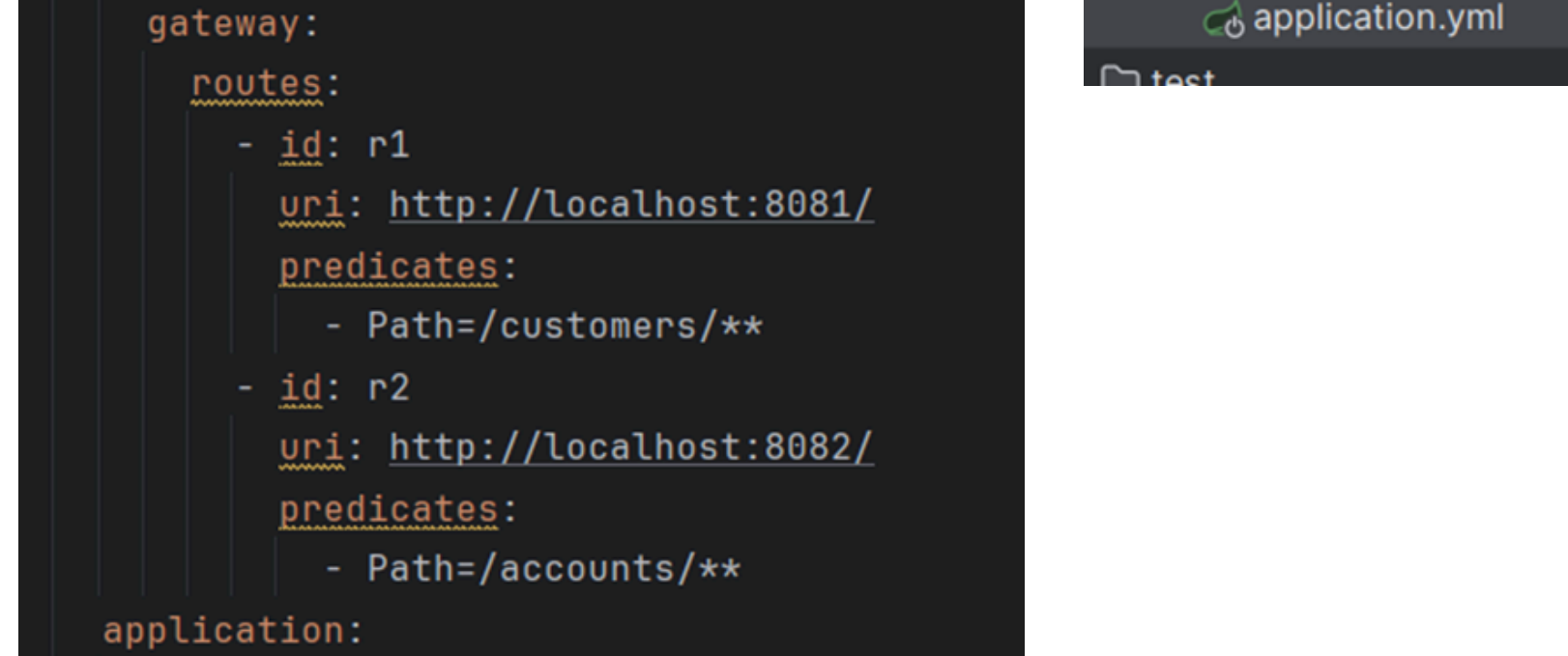
```xml
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.6.0</version>
</dependency>
```

localhost :8082/swagger-ui.html :



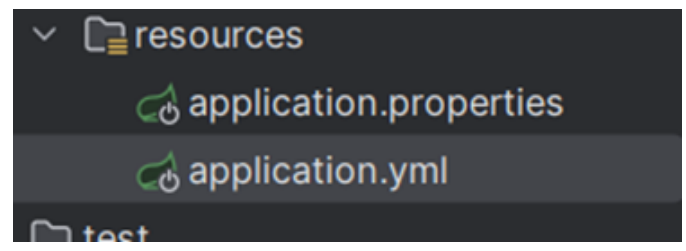


on fait la meme chose pour Customer service :
localhost :8081/swagger-ui.html



Dans le micro service gateway on créer application.yml:

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: r1
          uri: http://localhost:8081/
          predicates:
            - Path=/customers/**
        - id: r2
          uri: http://localhost:8082/
          predicates:
            - Path=/accounts/**
  application:
    name: gateway-service
server:
  port: 8888
```



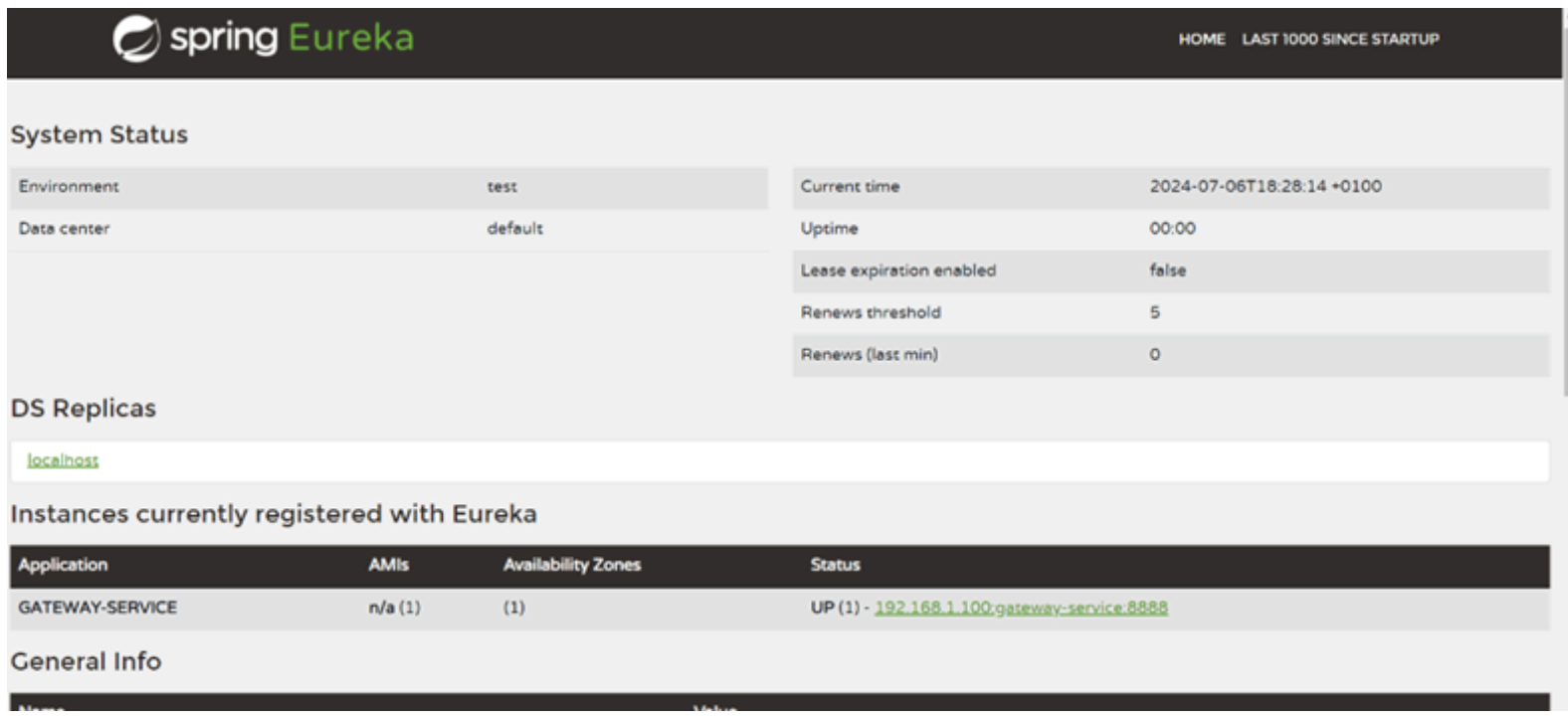
Dans le microservice Discovery service :

```java
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryServiceApplication {

    public static void main(String[] args) {

        SpringApplication.run(DiscoveryServiceApplication.class, args);
    }
```

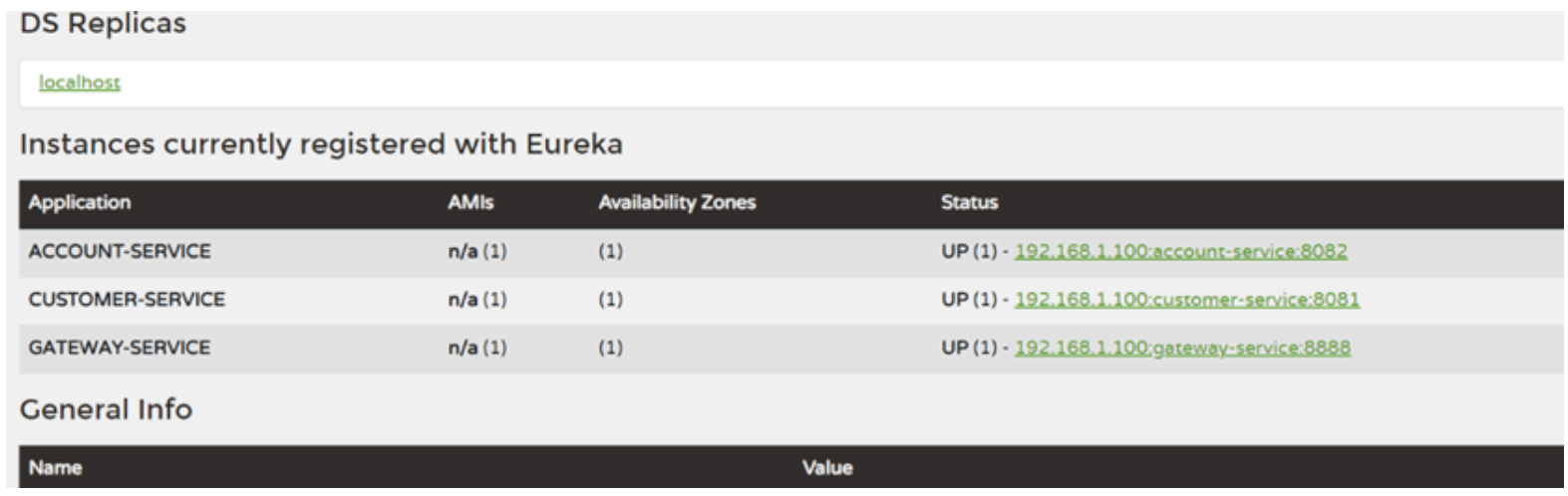
Dans application.proprties :

```
spring.application.name=discovery-service
server.port=8761
#dont register server itself as a client
eureka.client.fetch-registry=false
#does not register itself in the service registry
eureka.client.register-with-eureka=false
```

Localhost :8761 :



On met la proprieté de discovery= true :



Modifier application properties dans customer service :

```
spring.application.name=customer-service
spring.cloud.discovery.enabled=true
server.port=8081
spring.datasource.url=jdbc:h2:mem:customer-db
spring.h2.console.enabled=true
spring.cloud.config.enabled=false
eureka.instance.prefer-ip-address=true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
```

Modifier application properties dans account service :

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

Modifier application.properties dans gateway service

```
spring.application.name=gateway-service
server.port=8888
eureka.instance.prefer-ip-address=true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
```

Dans gateway service :

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.ReactiveDiscoveryClient;
import org.springframework.cloud.gateway.discovery.DiscoveryClientRouteDefinitionLocator;
import org.springframework.cloud.gateway.discovery.DiscoveryLocatorProperties;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class GatewayServiceApplication {

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

    @Bean
    DiscoveryClientRouteDefinitionLocator locator(ReactiveDiscoveryClient rdc, DiscoveryLocatorProperties dlp){
        return new DiscoveryClientRouteDefinitionLocator(rdc,dlp);
    }
```

Localhost :8888/CUSTOMER-SERVICE/customers :

{"id":1,"firstName":"Hiba","lastName":"salmi","email":"hibasalmi@gmail.com"},{"id":2,"firstName":"Mohammed","lastName":"arabi"

Dans accont service :
Ajouter dans les dependency suivante a pom:

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

Créer le package Clients :

```
3 usages
@FeignClient(name = "CUSTOMER-SERVICE")
public interface CustomerRestClient {

    @GetMapping("/customers/{id}")
    Customer findCustomerById(@PathVariable Long id);

    @GetMapping("/customers")
    List<Customer> allCustomers();
```

Dans AccountRestController :

```java
@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;
}
```

Ajouter ces dependences a Account service :

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

```java
public interface CustomerRestClient {

    @GetMapping("/customers/{id}")
    @CircuitBreaker(name = "customerService", fallbackMethod = "getDefaultCustomer")
    Customer findCustomerById(@PathVariable Long id);

    @GetMapping("/customers")
    List<Customer> allCustomers();

    no usages
    default Customer getDefaultCustomer(Long id, Exception exception) {
        Customer customer = new Customer();
        customer.setId(id);
        customer.setFirstName("Not Vailable");
        customer.setLastName("Not Vailable");
        customer.setEmail("Not Vailable");
        return customer;
    }
```

Modifier AccountServiceApplicationTests

```java
CommandLineRunner commandLineRunner(BankAccountRepository accountRepos
    return args -> {
        customerRestClient.allCustomers().forEach(c -> {
            BankAccount bankAccount1 = BankAccount.builder()
                    .accountId(UUID.randomUUID().toString())
                    .currency("MAD")
                    .balance(Math.random()*80000)
                    .createAt(LocalDate.now())
                    .type(AccountType.CURRENT_ACCOUNT)
                    .customerId(c.getId())
                    .build();
            BankAccount bankAccount2 = BankAccount.builder()
                    .accountId(UUID.randomUUID().toString())
                    .currency("MAD")
                    .balance(Math.random()*1928)
                    .createAt(LocalDate.now())
                    .type(AccountType.SAVING_ACCOUNT)
                    .customerId(c.getId())
                    .build();
            accountRepository.save(bankAccount1);
            accountRepository.save(bankAccount2);

        });
```

```
[
    {
        "accountId": "7512a2e7-5da5-4b96-b929-e8fbb38190da",
        "balance": 50462.69388385577,
        "createAt": "2024-07-06",
        "currency": "MAD",
        "type": "CURRENT_ACCOUNT",
        "customer": null,
        "customerId": 1
    },
    {
        "accountId": "94ca4104-b63b-4afc-949a-a821309b7e6a",
        "balance": 518.2924851203469,
        "createAt": "2024-07-06",
        "currency": "MAD",
        "type": "SAVING_ACCOUNT",
        "customer": null,
        "customerId": 1
    },
    {
        "accountId": "cc616036-e945-4527-93b1-65d52360d49e",
        "balance": 54296.94652054009,
        "createAt": "2024-07-06",
        "currency": "MAD",
        "type": "CURRENT_ACCOUNT",
        "customer": null,
        "customerId": 2
    },
    {
        "accountId": "2335b2f2-c30e-46aa-8fd5-28e713a0020e",
        "balance": 1322.703738344788,
        "createAt": "2024-07-06",
        "currency": "MAD",
        "type": "SAVING_ACCOUNT",
        "customer": null,
        "customerId": 2
    }
]
```