

Master Intelligence Artificielle et Analyse des Données

Systemes Distribués

---

***COMPTE RENDU DU TP6***

---

Réalisé par :

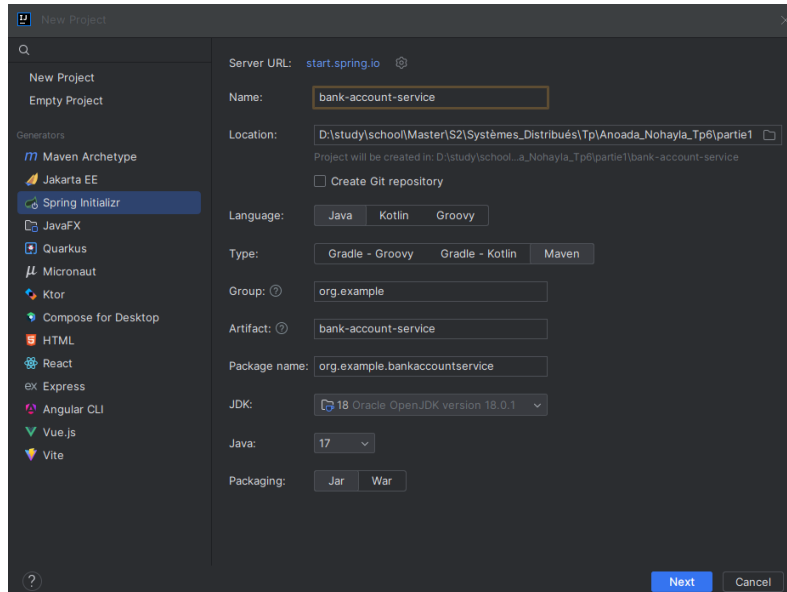
**HALIMA DAOUDI**

Année universitaire : 2023 – 2024

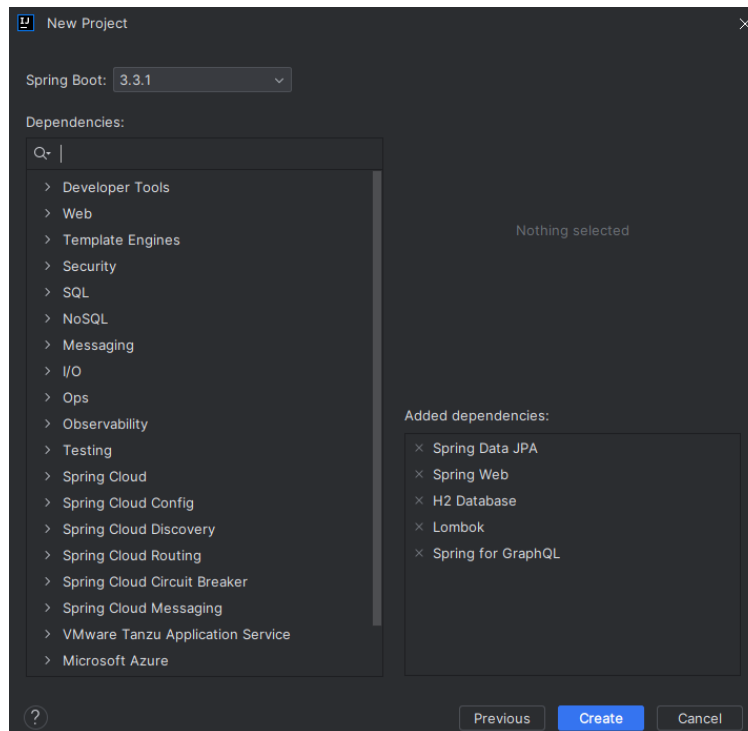
## Partie 1 :

### 1. Micro Service avec Web Service RESTFUL :

J'ai créé un nouveau projet Spring Boot en utilisant IntelliJ IDEA en sélectionnant "New Project" et "Spring Initializr".



Puis j'ai ajouté les dépendances nécessaires : Spring Web, Spring Data JPA, H2 Database, et Lombok, Spring For GraphQL



J'ai créé l'entité BankAccount avec les annotations nécessaires (@Entity, @Id, @Data, @NoArgsConstructor, @AllArgsConstructor, @Builder) pour représenter un compte bancaire avec des attributs tels que id, createdAt, balance, currency, et type (utilisant AccountType comme énumération).

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

J'ai créé l'énumération AccountType pour définir les différents types de comptes bancaires disponibles

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

J'initialise une application Spring Boot et je crée 10 comptes bancaires aléatoires avec des propriétés comme un identifiant unique, un type de compte, un solde, une date de création et une devise, puis je les enregistre dans un dépôt de comptes bancaires.

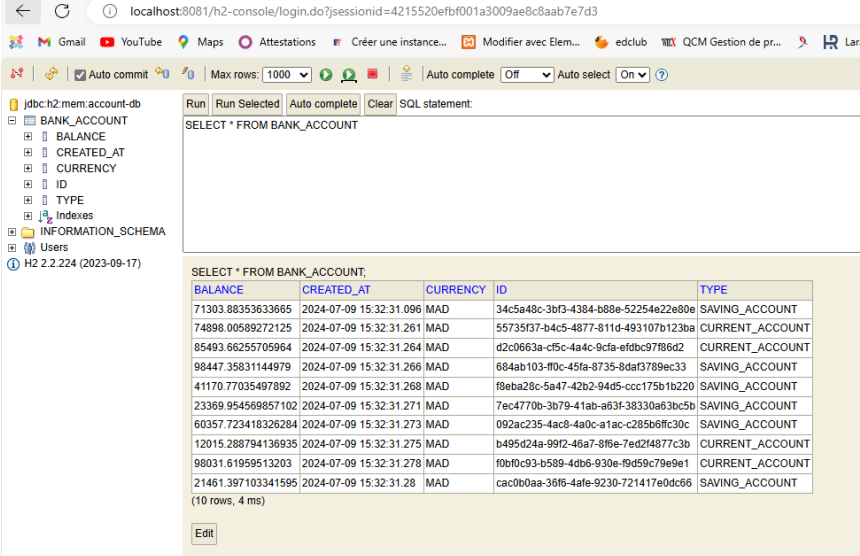
```
@SpringBootApplication
public class BankAccountServiceApplication
{
    public static void main(String[] args)
    {
        SpringApplication.run(BankAccountServiceApplication.class, args);
    }

    @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()*90000)
                    .createdAt(new Date())
                    .currency("MAD")
                    .build();
                bankAccountRepository.save(bankAccount);
            }
        };
    }
}
```

Je configure mon application Spring Boot pour s'appeler "bank-account-service", utiliser une base de données H2 en mémoire avec l'URL "jdbc:h2:mem:", activer la console H2 et écouter sur le port 8081.

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

Ensuite, je vérifie la présence des enregistrements via la console web H2.



The screenshot shows the H2 web console interface in a browser. The left sidebar displays the database structure, including the 'BANK\_ACCOUNT' table. The main area shows the SQL statement 'SELECT \* FROM BANK\_ACCOUNT' and its results. The results are displayed in a table with columns: BALANCE, CREATED\_AT, CURRENCY, ID, and TYPE. There are 10 rows of data, all with a CURRENCY of 'MAD' and a TYPE of either 'SAVING\_ACCOUNT' or 'CURRENT\_ACCOUNT'.

BALANCE	CREATED_AT	CURRENCY	ID	TYPE
71303.88353633665	2024-07-09 15:32:31.096	MAD	34c5a48c-3bf3-4384-b88e-52254e22e80e	SAVING_ACCOUNT
74898.00589272125	2024-07-09 15:32:31.261	MAD	55735f37-b4c5-4877-811d-493107b123ba	CURRENT_ACCOUNT
85493.66255705964	2024-07-09 15:32:31.264	MAD	d2c0663a-cf5c-4a4c-9cfa-efdbc97f86d2	CURRENT_ACCOUNT
98447.35831144979	2024-07-09 15:32:31.266	MAD	684ab103-f0c-45fa-8735-8daf3789ec33	SAVING_ACCOUNT
41170.77035497892	2024-07-09 15:32:31.268	MAD	f8eba28c-5a47-42b2-94d5-ccc175b1b220	SAVING_ACCOUNT
23369.954569857102	2024-07-09 15:32:31.271	MAD	7ec4770b-3b79-41ab-a63f-38330a63bc5b	SAVING_ACCOUNT
60357.723418326284	2024-07-09 15:32:31.273	MAD	092ac235-4ac8-4a0c-a1ac-c285b6ffc30c	SAVING_ACCOUNT
12015.288794136935	2024-07-09 15:32:31.275	MAD	b495d24a-99f2-46a7-8f0e-7ed2f4877c3b	CURRENT_ACCOUNT
98031.61959513203	2024-07-09 15:32:31.278	MAD	f0b0c93-b589-4db6-930e-f9d59c79e9e1	CURRENT_ACCOUNT
21461.397103341595	2024-07-09 15:32:31.28	MAD	cac0b0aa-36f6-4afe-9230-721417e0dc66	SAVING_ACCOUNT

Je crée un contrôleur REST nommé AccountRestController pour l'application Spring Boot, définissant des endpoints pour récupérer tous les comptes bancaires ou un compte spécifique par son identifiant

```
@RestController
public class AccountRestController {
    6 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 bankAccounts(@PathVariable String id) {
        return bankAccountRepository.findById(id)
            .orElseThrow(() -> new RuntimeException(String.format("Account %s not found", id)));
    }
}
```

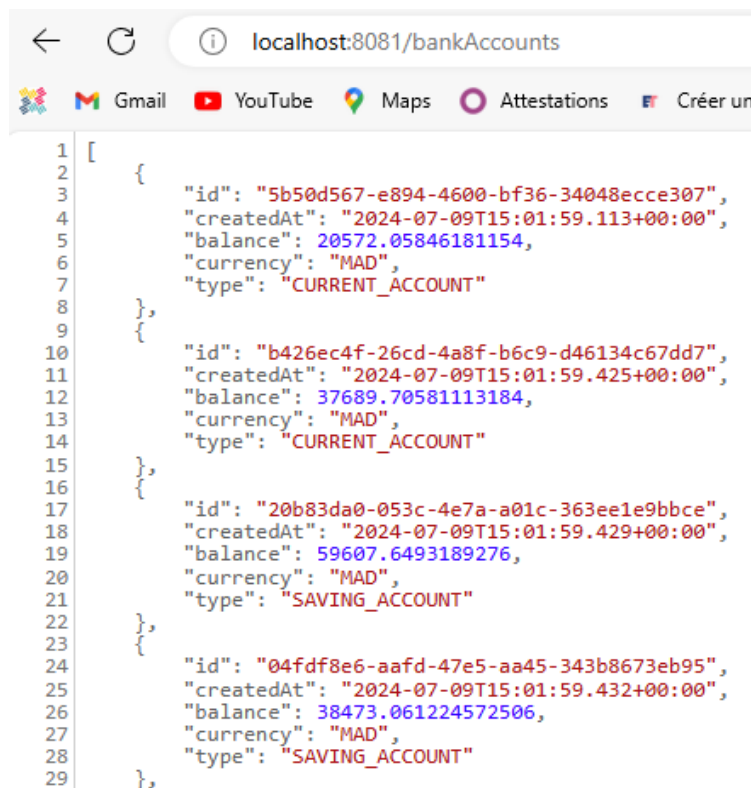
Ensuite, j'ajoute des endpoints pour créer, mettre à jour et supprimer des comptes bancaires.

```
@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(bankAccount);
}

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

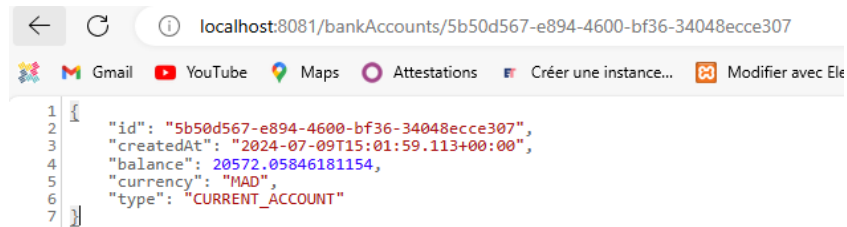
J'accède à l'URL suivante : <http://localhost:8081/bankAccounts> pour voir la liste des comptes bancaires



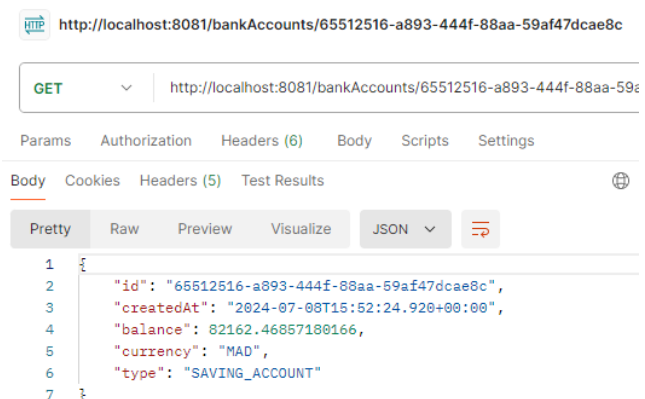
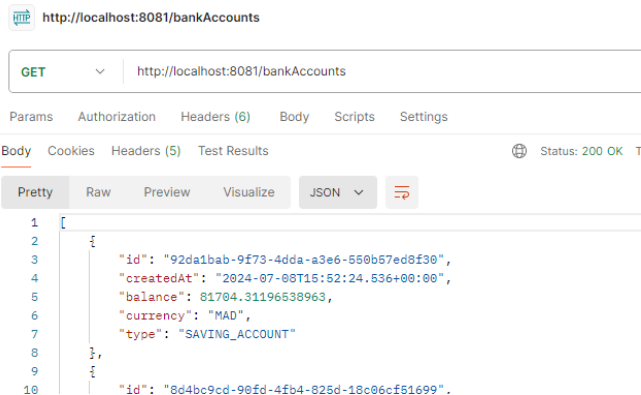
The screenshot shows a web browser window with the address bar displaying `localhost:8081/bankAccounts`. Below the address bar, there are several icons for Gmail, YouTube, Maps, Attestations, and a button labeled 'Créer un'. The main content area of the browser displays a JSON array of four bank account objects, each with fields for id, createdAt, balance, currency, and type. The first two are 'CURRENT\_ACCOUNT' and the last two are 'SAVING\_ACCOUNT'.

```
1 [
2   {
3     "id": "5b50d567-e894-4600-bf36-34048ecce307",
4     "createdAt": "2024-07-09T15:01:59.113+00:00",
5     "balance": 20572.05846181154,
6     "currency": "MAD",
7     "type": "CURRENT_ACCOUNT"
8   },
9   {
10    "id": "b426ec4f-26cd-4a8f-b6c9-d46134c67dd7",
11    "createdAt": "2024-07-09T15:01:59.425+00:00",
12    "balance": 37689.70581113184,
13    "currency": "MAD",
14    "type": "CURRENT_ACCOUNT"
15  },
16  {
17    "id": "20b83da0-053c-4e7a-a01c-363ee1e9bbce",
18    "createdAt": "2024-07-09T15:01:59.429+00:00",
19    "balance": 59607.6493189276,
20    "currency": "MAD",
21    "type": "SAVING_ACCOUNT"
22  },
23  {
24    "id": "04fdf8e6-aafd-47e5-aa45-343b8673eb95",
25    "createdAt": "2024-07-09T15:01:59.432+00:00",
26    "balance": 38473.061224572506,
27    "currency": "MAD",
28    "type": "SAVING_ACCOUNT"
29  },
30 ]
```

J'accède à l'URL suivante : <http://localhost:8081/bankAccounts/5b50d567-e894-4600-bf36-34048ecce307>, je peux récupérer les détails d'un compte bancaire spécifique, identifié par l'ID



J'utilise pour tester mes API. Il offre une interface conviviale qui me permet d'envoyer des requêtes HTTP vers des endpoints spécifiques, de gérer les différentes méthodes de requêtes comme GET, POST, PUT, DELETE, et de vérifier les réponses reçues.



J'intègre la dépendance Spring Boot OpenAPI Doc

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <version>1.8.0</version>
</dependency>
```

J'ajoute la dépendance spring-boot-starter-data-rest à mon projet Maven

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

Je modifie le repository en ajoutant l'annotation `@RepositoryRestResource`.

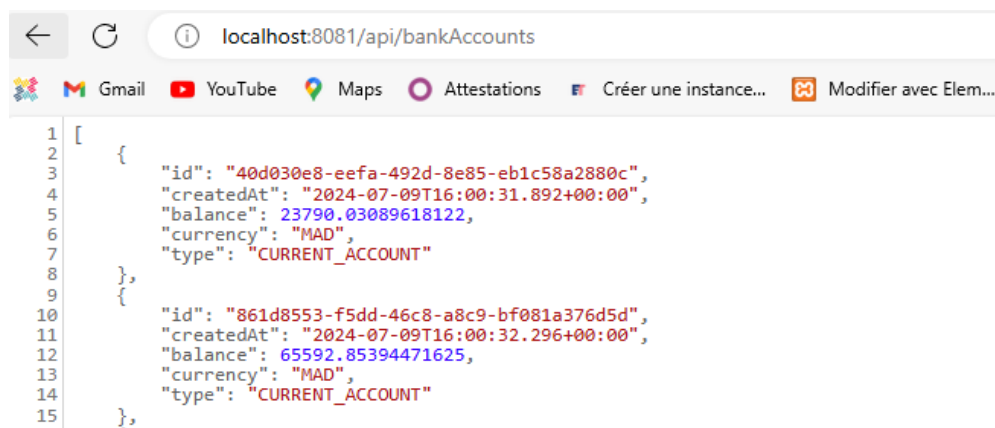
```
@RepositoryRestResource
public interface BankAccountRepository extends JpaRepository<BankAccount,String>
{
}
```

J'ajuste le contrôleur en ajoutant `@RequestMapping("/api")`.

```
@RestController
@RequestMapping("/api")
public class AccountRestController {
}
```

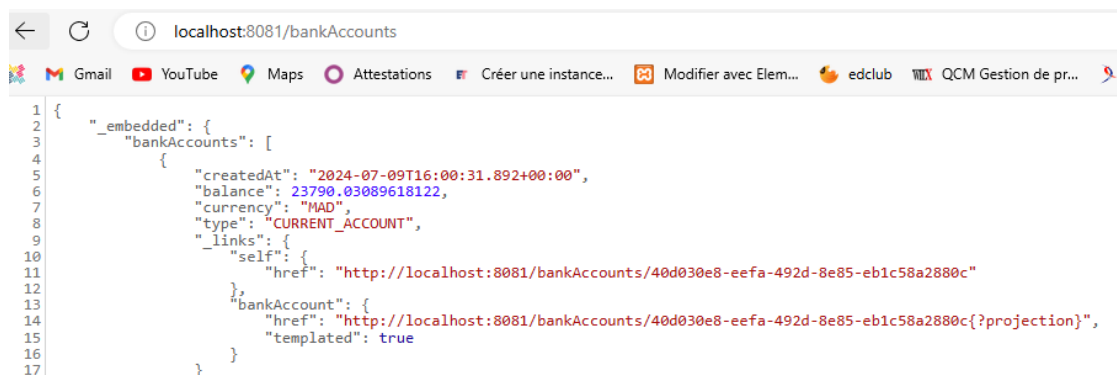
Lorsque j'ajoute `/api` à l'URL de base de mon application, cela crée deux services web RESTful distincts. Chacun de ces services expose des fonctionnalités ou des ressources différentes de manière structurée et standardisée

- RestController :



```
1 [
2   {
3     "id": "40d030e8-eefa-492d-8e85-eb1c58a2880c",
4     "createdAt": "2024-07-09T16:00:31.892+00:00",
5     "balance": 23790.03089618122,
6     "currency": "MAD",
7     "type": "CURRENT_ACCOUNT"
8   },
9   {
10    "id": "861d8553-f5dd-46c8-a8c9-bf081a376d5d",
11    "createdAt": "2024-07-09T16:00:32.296+00:00",
12    "balance": 65592.85394471625,
13    "currency": "MAD",
14    "type": "CURRENT_ACCOUNT"
15  },
16 ]
```

- Spring data rest



```
1 {
2   "_embedded": {
3     "bankAccounts": [
4       {
5         "createdAt": "2024-07-09T16:00:31.892+00:00",
6         "balance": 23790.03089618122,
7         "currency": "MAD",
8         "type": "CURRENT_ACCOUNT",
9         "_links": {
10          "self": {
11            "href": "http://localhost:8081/bankAccounts/40d030e8-eefa-492d-8e85-eb1c58a2880c"
12          },
13          "bankAccount": {
14            "href": "http://localhost:8081/bankAccounts/40d030e8-eefa-492d-8e85-eb1c58a2880c{?projection}",
15            "templated": true
16          }
17        }
18      }
19    ]
20  }
```

En utilisant l'URL <http://localhost:8081/bankAccounts?page=0&size=2>, je peux consulter les comptes bancaires avec une pagination intégrée. Cela permet de récupérer les deux premiers comptes de la liste, où `page=0` correspond à la première page et `size=2` indique que chaque page contient deux éléments.

```
1 {
2   "_embedded": {
3     "bankAccounts": [
4       {
5         "createdAt": "2024-07-09T16:00:31.892+00:00",
6         "balance": 23790.03089618122,
7         "currency": "MAD",
8         "type": "CURRENT_ACCOUNT",
9         "_links": {
10          "self": {
11            "href": "http://localhost:8081/bankAccounts/40d030e8-eefa-492d-8e85-eb1c58a2880c"
12          },
13          "bankAccount": {
14            "href": "http://localhost:8081/bankAccounts/40d030e8-eefa-492d-8e85-eb1c58a2880c{?projection}",
15            "templated": true
16          }
17        }
18      },
19      {
20        "createdAt": "2024-07-09T16:00:32.296+00:00",
21        "balance": 65592.85394471625,
22        "currency": "MAD",
23        "type": "CURRENT_ACCOUNT",
24        "_links": {
25          "self": {
26            "href": "http://localhost:8081/bankAccounts/861d8553-f5dd-46c8-a8c9-bf081a376d5d"
27          },
28          "bankAccount": {
29            "href": "http://localhost:8081/bankAccounts/861d8553-f5dd-46c8-a8c9-bf081a376d5d{?projection}",
30            "templated": true
31          }
32        }
33      }
34    ]
35  }
36 }
```

Je modifie le `BankRepository` existant en ajoutant une méthode pour rechercher des comptes bancaires par type.

```
@RepositoryRestResource
public interface BankAccountRepository extends JpaRepository<BankAccount,String>
{
    no usages
    List<BankAccount> findByType(AccountType type);
}
```

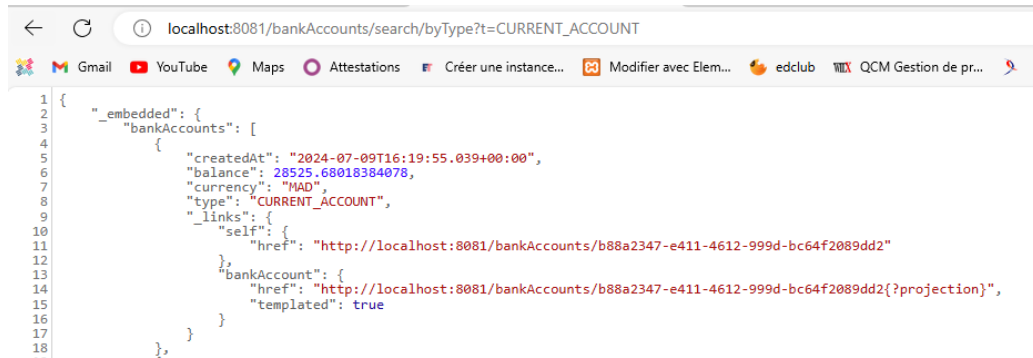
En utilisant l'URL [http://localhost:8081/bankAccounts/search/findByType?type=CURRENT\\_ACCOUNT](http://localhost:8081/bankAccounts/search/findByType?type=CURRENT_ACCOUNT), je peux rechercher des comptes bancaires de type "CURRENT\_ACCOUNT".

```
1 {
2   "_embedded": {
3     "bankAccounts": [
4       {
5         "createdAt": "2024-07-09T16:16:11.577+00:00",
6         "balance": 21107.198966517364,
7         "currency": "MAD",
8         "type": "CURRENT_ACCOUNT",
9         "_links": {
10          "self": {
11            "href": "http://localhost:8081/bankAccounts/aaa8bfe2-8cb2-4b82-b654-2d8c92ce5c9f"
12          },
13          "bankAccount": {
14            "href": "http://localhost:8081/bankAccounts/aaa8bfe2-8cb2-4b82-b654-2d8c92ce5c9f{?projection}",
15            "templated": true
16          }
17        }
18      },
19      {
20        "createdAt": "2024-07-09T16:16:11.834+00:00",
21        "balance": 38018.04378149235,
22        "currency": "MAD"
23      }
24    ]
25  }
26 }
```



J'ajuste la méthode `findByType` dans le repository pour spécifier le chemin `/byType` et utiliser le paramètre `type` de type `AccountType`.

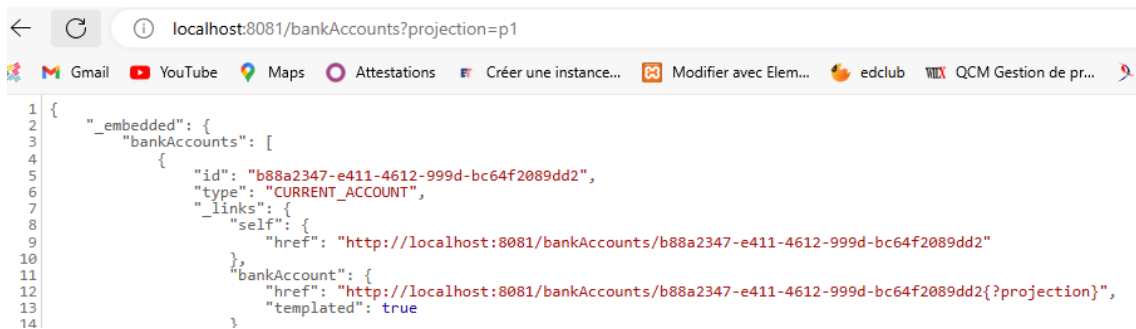
```
@RepositoryRestResource
public interface BankAccountRepository extends JpaRepository<BankAccount,String>
{
    no usages
    @RestResource(path = "/byType")
    List<BankAccount> findByType(@Param("t") AccountType type);
}
```



```
1 {
2   "_embedded": {
3     "bankAccounts": [
4       {
5         "createdAt": "2024-07-09T16:19:55.039+00:00",
6         "balance": 28525.68018384078,
7         "currency": "MAD",
8         "type": "CURRENT_ACCOUNT",
9         "_links": {
10          "self": {
11            "href": "http://localhost:8081/bankAccounts/b88a2347-e411-4612-999d-bc64f2089dd2"
12          },
13          "bankAccount": {
14            "href": "http://localhost:8081/bankAccounts/b88a2347-e411-4612-999d-bc64f2089dd2{?projection}",
15            "templated": true
16          }
17        }
18      }
19    ]
20  }
```

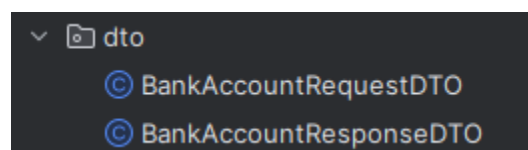
Je peux utiliser des projections pour récupérer des comptes bancaires avec seulement certains attributs spécifiques plutôt que tous les détails.

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



```
1 {
2   "_embedded": {
3     "bankAccounts": [
4       {
5         "id": "b88a2347-e411-4612-999d-bc64f2089dd2",
6         "type": "CURRENT_ACCOUNT",
7         "_links": {
8          "self": {
9            "href": "http://localhost:8081/bankAccounts/b88a2347-e411-4612-999d-bc64f2089dd2"
10          },
11          "bankAccount": {
12            "href": "http://localhost:8081/bankAccounts/b88a2347-e411-4612-999d-bc64f2089dd2{?projection}",
13            "templated": true
14          }
15        }
16      }
17    ]
18  }
```

Ensuite, j'ai créé une couche DAO



```
dto
├── BankAccountRequestDTO
└── BankAccountResponseDTO
```

J'ai créé BankAccountRequestDTO pour transférer les données nécessaires lors de la création ou de la mise à jour des comptes bancaires. Elle contient des champs tels que balance, currency, et type, permettant de spécifier les détails du compte.

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

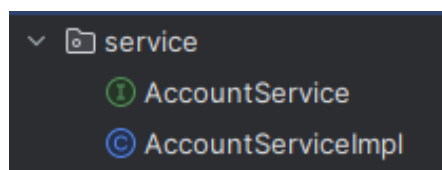
    private double balance;
    private String currency;
    private AccountType type;
}
```

J'ai créé BankAccountResponseDTO pour répondre aux requêtes concernant les comptes bancaires. Cette classe inclut des informations telles que : id, createdAt, balance, currency, et type, fournissant une vue complète des détails d'un compte après une opération comme la création, la récupération ou la mise à jour.

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

    private String id;
    private Date createdAt;
    private double balance;
    private String currency;
    private AccountType type;
}
```

Ensuite, j'ai créé un service



Ensuite, je crée ce service AccountService avec une méthode addAccount qui prend en charge l'ajout de nouveaux comptes bancaires à partir d'une requête représentée par BankAccountRequestDTO.

```
public interface AccountService
{
    1 usage 1 implementation
    public BankAccountResponseDTO addAccount(BankAccountRequestDTO bankAccountRequestDTO);
}
```

Ensuite, j'implémente le service AccountService pour gérer l'ajout de nouveaux comptes bancaires à partir des données fournies dans BankAccountRequestDTO.

```
@Service
@Transactional
public class AccountServiceImpl implements AccountService
{
    @Autowired
    private BankAccountRepository bankAccountRepository;
    1 usage
    @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())
            .createdAt(saveBankAccount.getCreatedAt())
            .balance(saveBankAccount.getBalance())
            .type(saveBankAccount.getType())
            .currency(saveBankAccount.getCurrency())
            .build();
        return bankAccountResponseDTO;
    }
}
```

Je crée un package mappers où je développe la classe AccountMapper pour convertir les objets BankAccount en BankAccountResponseDTO, simplifiant ainsi le processus de transfert de données entre les entités et les DTOs dans mon application.

```
@Controller
public class AccountMapper
{
    no usages
    public BankAccountResponseDTO fromBankAccount(BankAccount bankAccount)
    {
        BankAccountResponseDTO bankAccountResponseDTO=new BankAccountResponseDTO();
        BeanUtils.copyProperties(bankAccount, bankAccountResponseDTO);
        return bankAccountResponseDTO;
    }
}
```

Ensuite, Je modifier le contrôleur pour inclure l'utilisation des mappers.

```
@RestController
@RequestMapping("/api")
public class AccountRestController {
    6 usages
    private BankAccountRepository bankAccountRepository;
    2 usages
    private AccountService accountService;
    no usages
    private AccountMapper accountMapper;
```

Et puis je modifie l'implémentation du service pour inclure l'utilisation des mappers.

```
@Service
@Transactional
public class AccountServiceImpl implements AccountService
{
    @Autowired
    private BankAccountRepository bankAccountRepository;
    1 usage
    private AccountMapper accountMapper;
    1 usage
    @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 = accountMapper.fromBankAccount(saveBankAccount);
        return bankAccountResponseDTO;
    }
}
```

## 2. Micro Service avec web service GRAPHQL :

J'intègre la dépendance GraphQL à mon projet en ajoutant les bibliothèques et les dépendances nécessaires. Cela me permet d'activer la gestion des requêtes et des réponses via GraphQL dans mon application.

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

Je crée et définis un schéma GraphQL en spécifiant les types de données, les champs et les résolveurs requis pour mon API. Ce schéma détermine comment les données peuvent être interrogées, mises à jour et manipulées via GraphQL dans mon application.

```
resources
└─ graphql
    └─ schema.graphql

type Query {
  accountList: [BankAccount]
}

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

Je crée un contrôleur web nommé BankAccountGraphQLController qui utilise un service pour interagir avec les données. Ce contrôleur agit comme une interface entre le client et la couche de données, facilitant ainsi les requêtes et les réponses via GraphQL dans mon application.

```
@Controller
public class BankAccountGraphQLController {

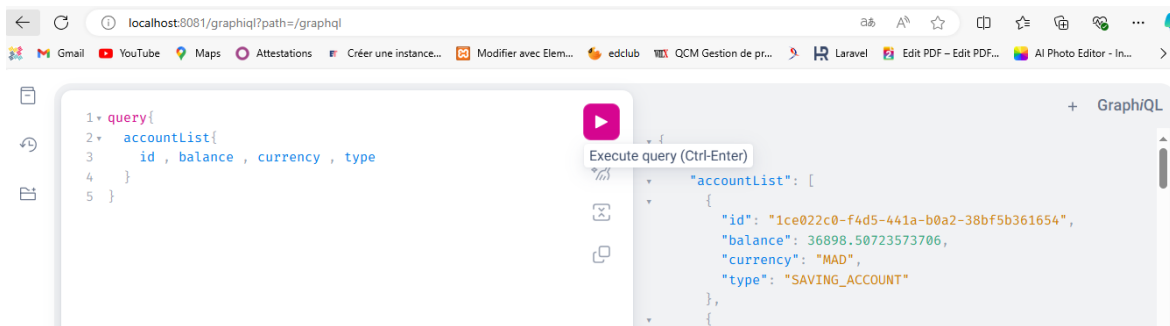
    @Autowired
    private BankAccountRepository bankAccountRepository;

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

J'ajoute activation d'interface GraphQL dans application Spring Boot en application.properties

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

J'ai utilisé l'interface GraphQL pour exécuter une requête GraphQL demandant les champs id, balance, currency et type pour chaque élément de la liste de comptes (accountList) disponibles sur mon serveur local



J'ajoute une méthode dans BankAccountGraphQLController pour récupérer un compte bancaire spécifique par son identifiant

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

J'ajoute en schéma graphql un champ nommé bankAccountById qui prend un argument id de type String. Ce champ permettra de récupérer les détails d'un compte bancaire spécifique en utilisant son identifiant.

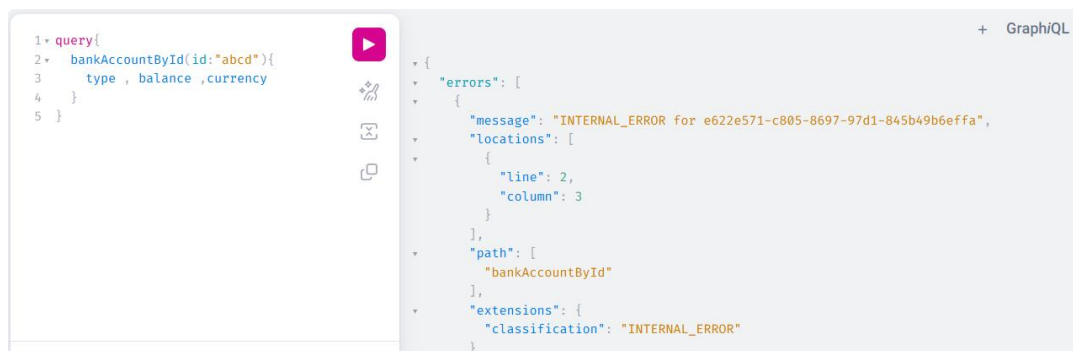
```
type Query {
    accountList: [BankAccount] ,
    bankAccountById(id:String) : BankAccount
}

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

J'ai utilisé l'interface GraphiQL pour exécuter une requête GraphQL qui récupère le type, le solde et la devise d'un compte bancaire spécifique identifié par un ID

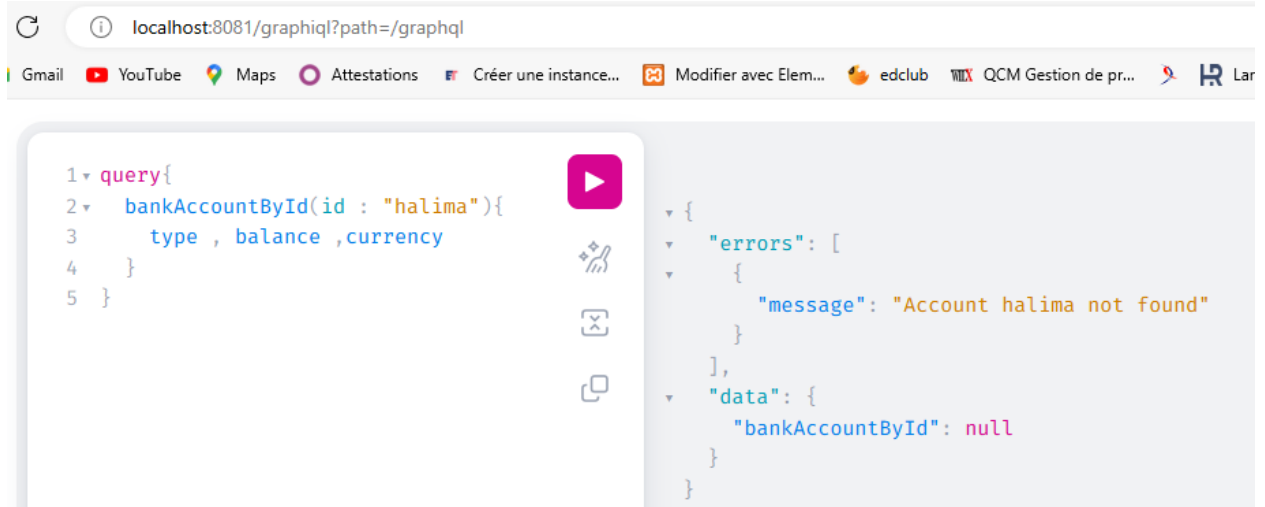


Si le compte recherché par son identifiant n'existe pas, alors un message d'erreur est affiché.



Et après je crée handler exception nommé CustomDataFetcherExceptionHandler qui personnalise la gestion des exceptions pour les requêtes GraphQL dans une application Spring Boot

```
@Component
public class CustomDataFetcherExceptionHandler extends DataFetcherExceptionHandlerAdapter {
    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;
            }
        };
    }
}
```



J'ajoute une méthode `addAccount` en `BankAccountGraphQLController` indiquant qu'elle gère une mutation GraphQL pour créer et sauvegarder un nouveau compte bancaire.

```
@Autowired
private AccountService accountService;

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

Je modifie schéma GraphQL en ajoutant une mutation `addAccount` pour créer un nouveau compte bancaire, utilisant un input `BankAccountDTO` pour spécifier les détails du compte.

```
type Query {
  accountList: [BankAccount] ,
  bankAccountById(id:String) : BankAccount
}

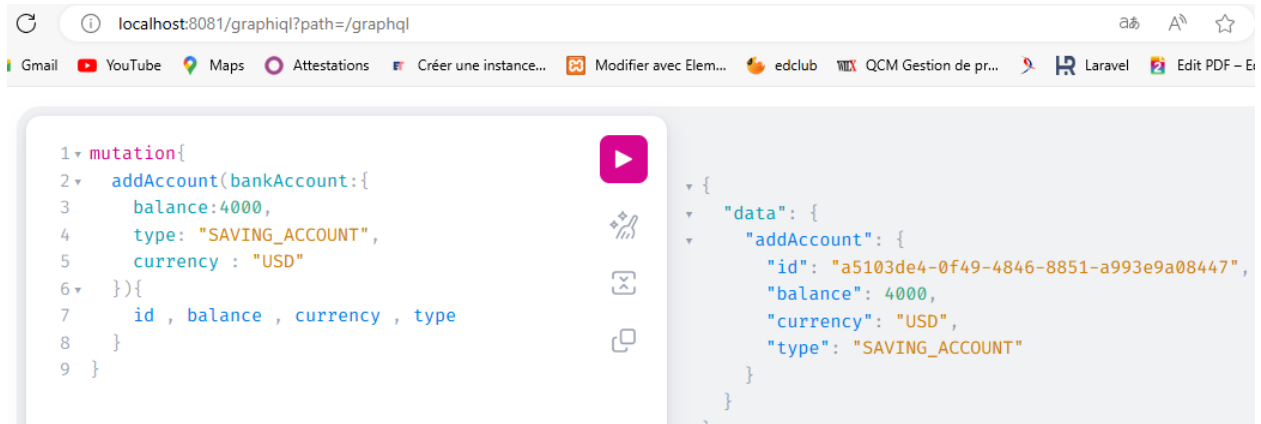
input BankAccountDTO {
  balance: Float,
  currency: String,
  type: String
}

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

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



J'ajoute un nouveau compte à l'aide de GraphQL en utilisant une requête de mutation GraphQL, Cette requête spécifie les détails du compte à créer



Pour la modification, j'ai étendu l'interface `AccountService` en y ajoutant une méthode `updateAccount`.

```
public interface AccountService
{
    2 usages 1 implementation
    BankAccountResponseDTO addAccount(BankAccountRequestDTO bankAccountRequestDTO);
    1 usage 1 implementation
    BankAccountResponseDTO updateAccount(String id ,BankAccountRequestDTO bankAccountRequestDTO);
}
```

Ensuite, je détaille l'implémentation de la méthode `updateAccount` dans le service `AccountServiceImpl`. Cette méthode vérifie l'existence du compte à mettre à jour, puis applique les modifications spécifiées.

```
@Override
public BankAccountResponseDTO updateAccount(String id, BankAccountRequestDTO bankAccountDTO) {
    Optional<BankAccount> existingBankAccountOptional = bankAccountRepository.findById(id);
    if (!existingBankAccountOptional.isPresent()) {
        throw new ResourceNotFoundException("BankAccount not found with ID: " + id);
    }
    BankAccount existingBankAccount = existingBankAccountOptional.get();
    existingBankAccount.setBalance(bankAccountDTO.getBalance());
    existingBankAccount.setType(bankAccountDTO.getType());
    existingBankAccount.setCurrency(bankAccountDTO.getCurrency());
    BankAccount updatedBankAccount = bankAccountRepository.save(existingBankAccount);
    return accountMapper.fromBankAccount(updatedBankAccount);
}
```

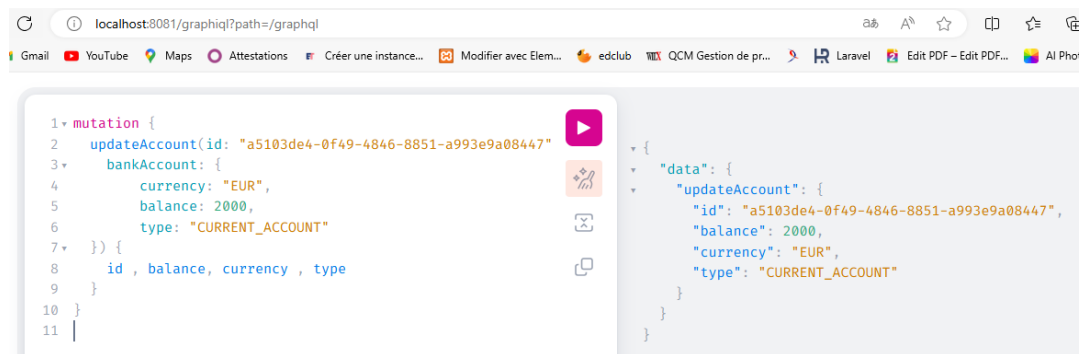
J'ajoute une méthode pour mettre à jour un compte bancaire existant via une mutation GraphQL dans `BankAccountGraphQLController`

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

J'ai modifié le schema dans GraphQL pour inclure une mutation GraphQL nommée `updateAccount`, permettant de mettre à jour un compte bancaire existant en utilisant son identifiant et les détails spécifiés dans `BankAccountDTO`.

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

Je teste la modification en utilisant l'interface GraphQL pour exécuter la mutation GraphQL `updateAccount`



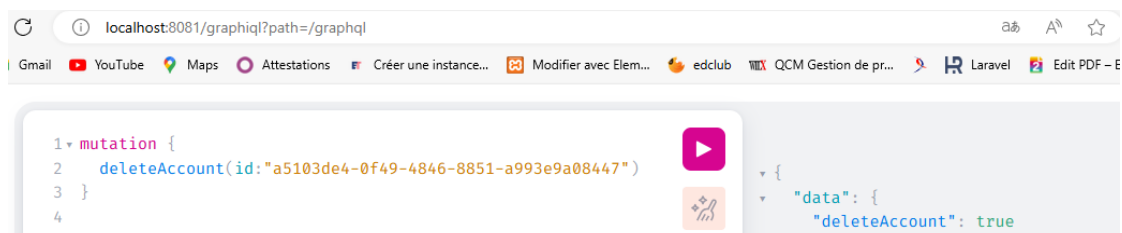
Pour la suppression, j'ajoute une méthode de mutation GraphQL dans `BankAccountGraphQLController` pour supprimer un compte bancaire existant en utilisant son identifiant.

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

Ensuite, j'ajoute une opération de mutation GraphQL `deleteAccount` dans le schéma GraphQL.

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

Je teste la suppression en utilisant l'interface GraphQL



Après cela, je crée l'entité Customer avec une relation OneToMany vers l'entité BankAccount. Cette relation permet à un client d'avoir plusieurs comptes bancaires.

```
@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;
}
```

Ensuite, j'ajoute une variable dans l'entité BankAccount avec une relation ManyToOne vers Customer. Cette relation permet à chaque compte bancaire d'être associé à un seul client

```
@Entity
@Data @NoArgsConstructor @AllArgsConstructor @Builder
public class BankAccount
{
    @Id
    private String id;
    private Date createdAt;
    private Double balance;
    private String currency;
    @Enumerated(EnumType.STRING)
    private AccountType type;
    @ManyToOne
    private Customer customer;
}
```

Je crée un repository pour l'entité Customer afin de gérer les opérations de base de données liées aux clients

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

Je créer liste de clients et pour chaque client je créer ensemble de comptes

```
@Bean
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()*90000)
                    .createdAt(new Date())
                    .currency("MAD")
                    .customer(customer)
                    .build();
                bankAccountRepository.save(bankAccount);
            }
        });
    };
};
```

Ensuite, je vérifie la présence des enregistrements dans la base de données en accédant à la console H2.

SELECT \* FROM BANK\_ACCOUNT;

BALANCE	CREATED_AT	CUSTOMER_ID	CURRENCY	ID	TYPE
72323.84679241215	2024-07-09 14:24:11.264	1	MAD	809cdde1-01b3-4e2c-b7c1-af505326b025	SAVING_ACCOUNT
16606.48785849643	2024-07-09 14:24:11.309	1	MAD	80f7fd13-0bc7-42fb-a442-02105b2d65e5	SAVING_ACCOUNT
85734.85560971653	2024-07-09 14:24:11.315	1	MAD	2a30e643-c081-4ad1-b565-477539d73f3c	CURRENT_ACCOUNT
63744.91354764527	2024-07-09 14:24:11.32	1	MAD	eeb9b3af-65bc-4adf-8b43-d7f5e19bcfff	SAVING_ACCOUNT
64907.913295768616	2024-07-09 14:24:11.326	1	MAD	3d89e33d-6dd5-4be3-a43d-a0adc08acdfd	SAVING_ACCOUNT
87254.78150778238	2024-07-09 14:24:11.332	1	MAD	acaa5609-7f41-4e7d-882c-98858903283e	SAVING_ACCOUNT
66890.76027235176	2024-07-09 14:24:11.338	1	MAD	d9298873-a6e2-4535-9666-c8b2dd7d27a0	CURRENT_ACCOUNT
34091.79366662585	2024-07-09 14:24:11.343	1	MAD	25eecbae-8e85-4f8f-8f4e-489b3e1a2d26	SAVING_ACCOUNT
88643.34336480749	2024-07-09 14:24:11.348	1	MAD	b341b583-6388-40af-80a1-1132e89afd74	SAVING_ACCOUNT
51691.91779801895	2024-07-09 14:24:11.353	1	MAD	9c9d8cc9-f6b8-43f8-aa38-71e654e8400c	CURRENT_ACCOUNT
97820.65836638422	2024-07-09 14:24:11.358	2	MAD	ea5efe61-3a81-4c47-8c2e-74678399e488	SAVING_ACCOUNT
47580.18172260162	2024-07-09 14:24:11.365	2	MAD	e42f8e55-b8d0-4a29-ae94-07f7bdb48f48	SAVING_ACCOUNT

En modifiant le schéma GraphQL, j'ajoute le nouveau type Customer et je mets à jour le type BankAccount en lui ajoutant l'attribut customer. Cela permet de représenter les relations entre les clients et les comptes bancaires dans l'API GraphQL,

```

type Customer{
  id :ID ,
  name :String ,
  bankAccounts : [BankAccount]
}
type BankAccount {
  id: String,
  createdAt: Float,
  balance: Float,
  currency: String,
  type: String ,
  customer : Customer
}

```

localhost:8081/graphql?path=/graphql

ail YouTube Maps Attestations Créer une instance... Modifier avec Elem... edclub QCM Gestion de pr... Laravel Edit PDF - E

```

1 query{
2   accountList{
3     id , balance , customer{name}
4   }
5 }

```

```

{
  "data": {
    "accountList": [
      {
        "id": "1ce022c0-f4d5-441a-b0a2-38bf5b361654",
        "balance": 36898.50723573706,
        "customer": {
          "name": "Mohamed"
        }
      },
      {
        "id": "9068018e-5fd5-4024-943f-511253f4bb42",
        "balance": 34943.387278326576,
        "customer": {
          "name": "Mohamed"
        }
      }
    ]
  }
}

```

J'ajoute une méthode dans le contrôleur BankAccountGraphQLController qui permet de lister les clients. Cette méthode sera exposée via une requête GraphQL pour récupérer tous les clients

```

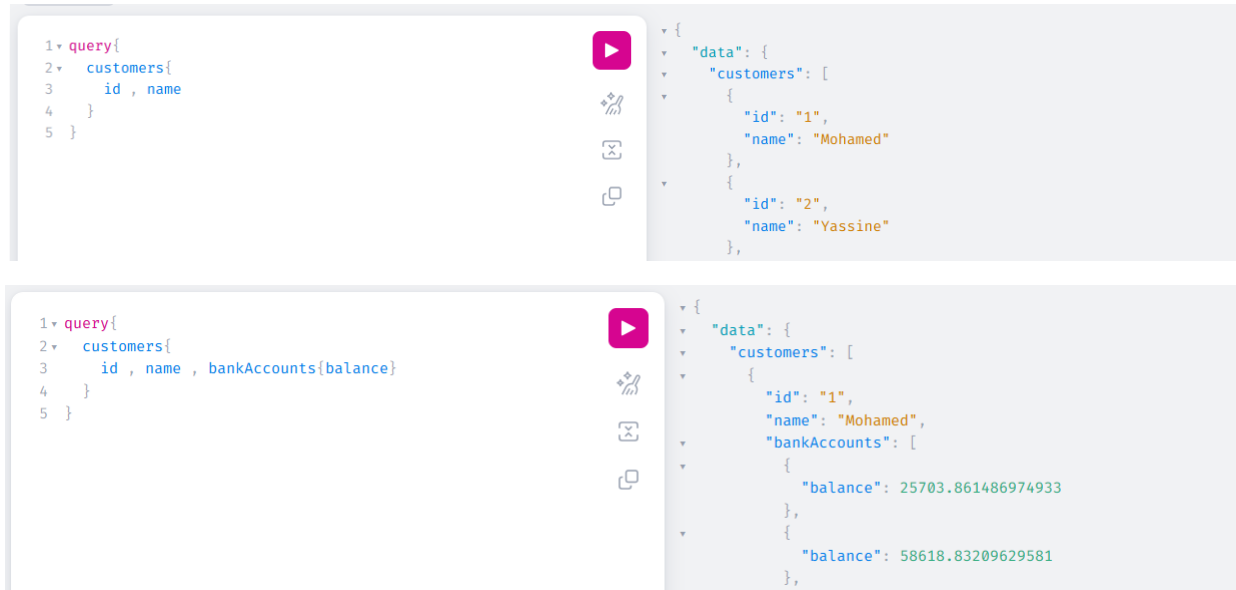
@Autowired
private CustomerRepository customerRepository;

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

```

Je modifie le schéma GraphQL en ajoutant une requête nommée customers qui permet de récupérer la liste des clients enregistrés dans le système.

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



The first screenshot shows a query to fetch a list of customers by their IDs and names. The query is: `query { customers { id, name } }`. The response is a JSON object: `{ "data": { "customers": [ { "id": "1", "name": "Mohamed" }, { "id": "2", "name": "Yassine" } ] } }`.

The second screenshot shows a query to fetch a list of customers, including their bank account balances. The query is: `query { customers { id, name, bankAccounts { balance } } }`. The response is a JSON object: `{ "data": { "customers": [ { "id": "1", "name": "Mohamed", "bankAccounts": [ { "balance": 25703.861486974933 } ] }, { "id": "2", "name": "Yassine", "bankAccounts": [ { "balance": 58618.83209629581 } ] } ] } }`.

Pour éviter la redondance dans la relation bidirectionnelle, j'ajoute l'annotation `@JsonProperty` dans les entités concernées.

```
@Entity @NoArgsConstructor @AllArgsConstructor @Data @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;  
}
```

Donc, lorsque je consulte maintenant la liste des comptes, il n'y a plus de redondances

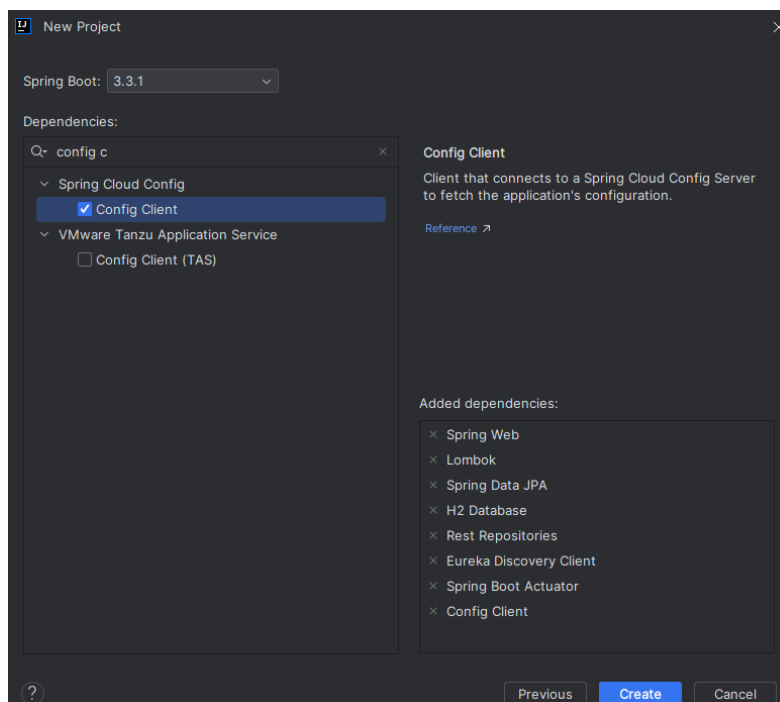
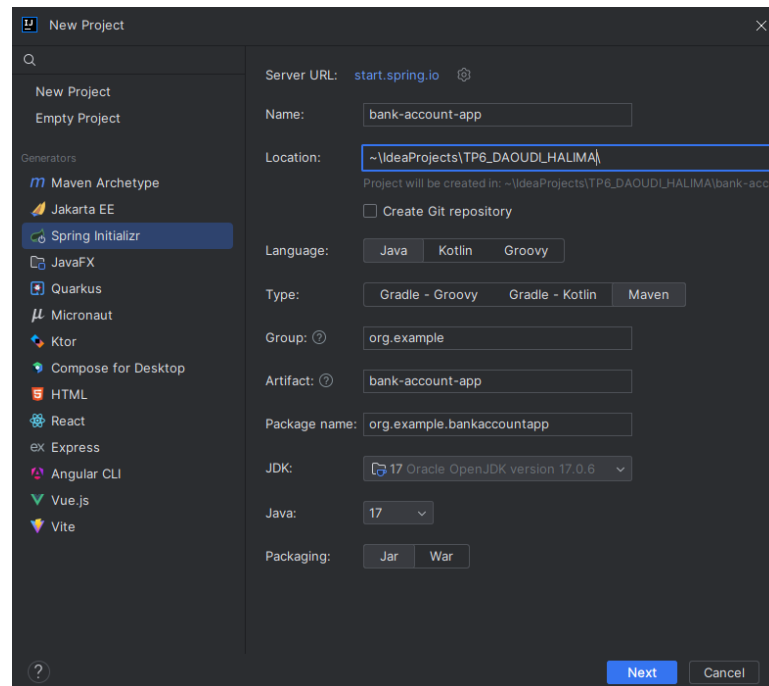
```
localhost:8081/api/bankAccounts

[
  {
    "id": "cc081d70-0d7c-4fe7-94e5-5279a380af95",
    "createdAt": "2024-07-09T13:52:38.886+00:00",
    "balance": 32184.54669358153,
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT",
    "customer": {
      "id": 1,
      "name": "Mohamed"
    }
  },
  {
    "id": "59ce5379-2e8d-4478-8707-c299cf439be8",
    "createdAt": "2024-07-09T13:52:38.940+00:00",
    "balance": 53290.93332205012,
    "currency": "MAD",
    "type": "SAVING_ACCOUNT",
    "customer": {
      "id": 1,
      "name": "Mohamed"
    }
  },
  {
    "id": "f065ea2a-56f8-4c16-8d59-7b38cee90008",
    "createdAt": "2024-07-09T13:52:38.948+00:00",
    "balance": 41506.85304870679,
    "currency": "MAD",
    "type": "SAVING_ACCOUNT",
    "customer": {
      "id": 1,
      "name": "Mohamed"
    }
  }
]
```

## Partie 2 :

Je développe une application basée sur des micro-services dédiée à la gestion des clients et des comptes bancaires. Chaque compte bancaire est associé à un client spécifique, assurant ainsi une relation directe entre les deux entités

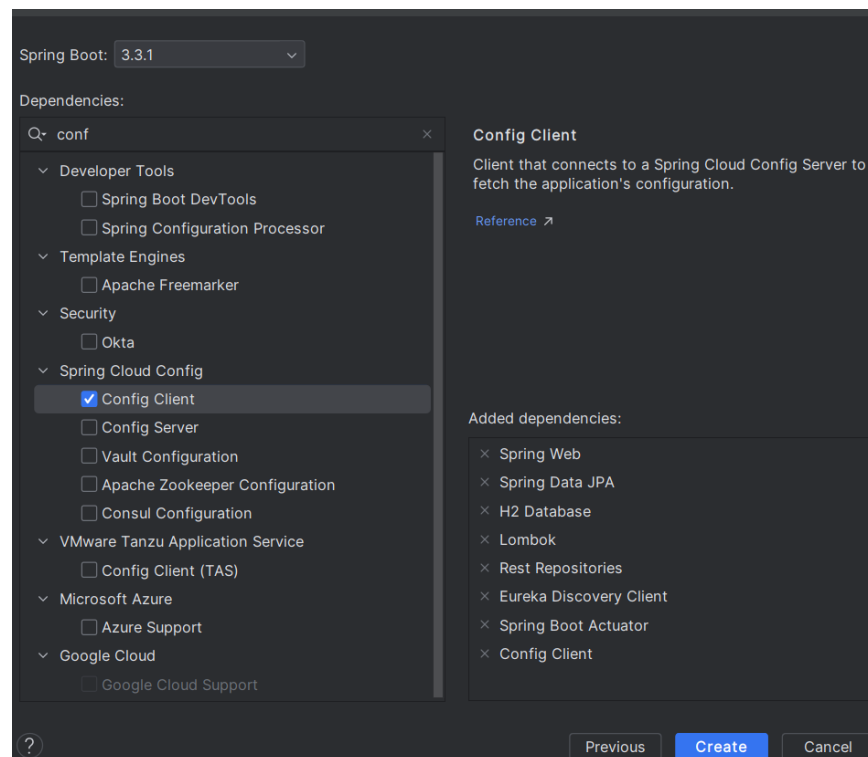
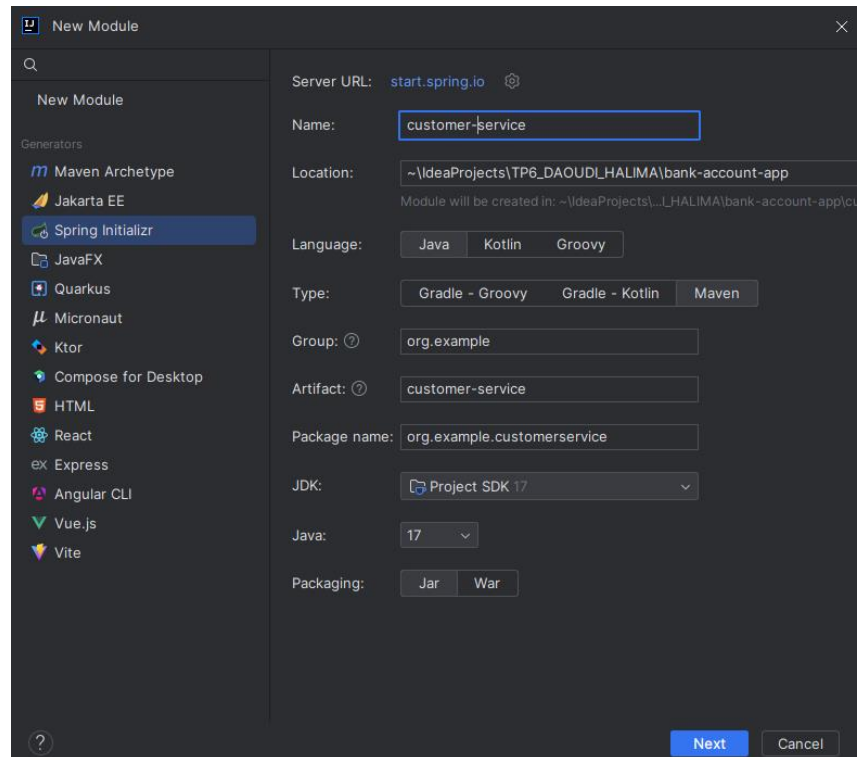
Pour commencer, je crée un projet Maven appelé **bank-account-app** pour maintenir la structure et la gestion des dépendances de manière organisée.





Ensuite, j'organise le projet en modules distincts, chacun représentant un micro-service autonome.

## 1.Customer-service :



## 2. account-service :

The 'New Module' dialog in IntelliJ IDEA is shown with the 'Spring Initializr' generator selected. The configuration for the 'account-service' module is as follows:

- Server URL: [start.spring.io](https://start.spring.io)
- Name: `account-service`
- Location: `~\IdeaProjects\TP6_DAOUDLHALIMA\bank-account-app`  
Module will be created in: `~\IdeaProjects\...DLHALIMA\bank-account-app\...`
- Language: `Java` (selected), `Kotlin`, `Groovy`
- Type: `Gradle - Groovy` (selected), `Gradle - Kotlin`, `Maven`
- Group: `org.example`
- Artifact: `account-service`
- Package name: `org.example.accountservice`
- JDK: `Project SDK 17`
- Java: `17`
- Packaging: `Jar` (selected), `War`

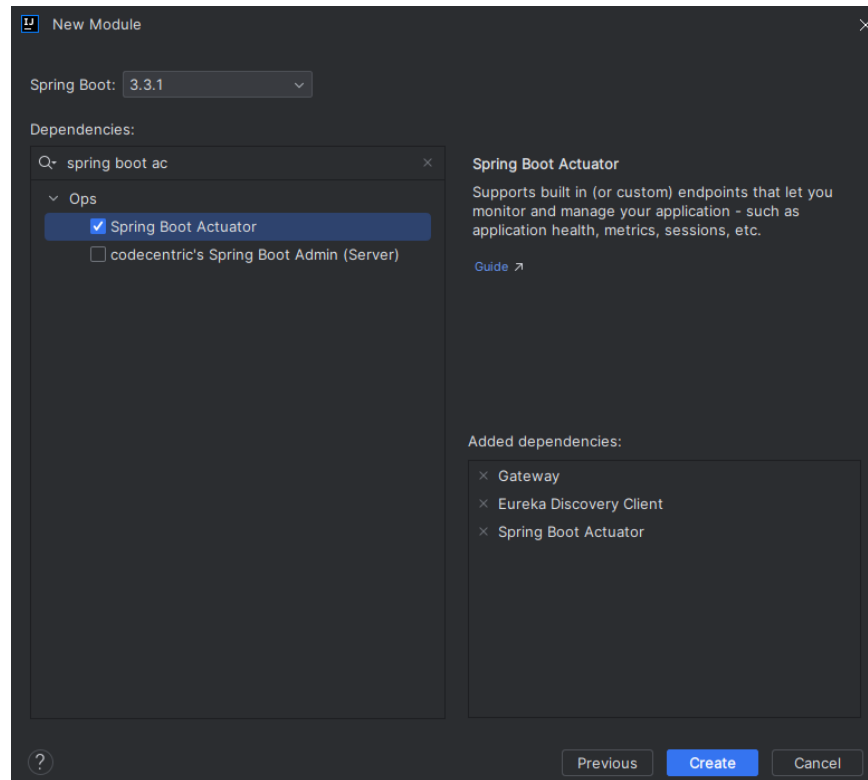
Buttons: `Next` (highlighted), `Cancel`

## 3. Gateway-service :

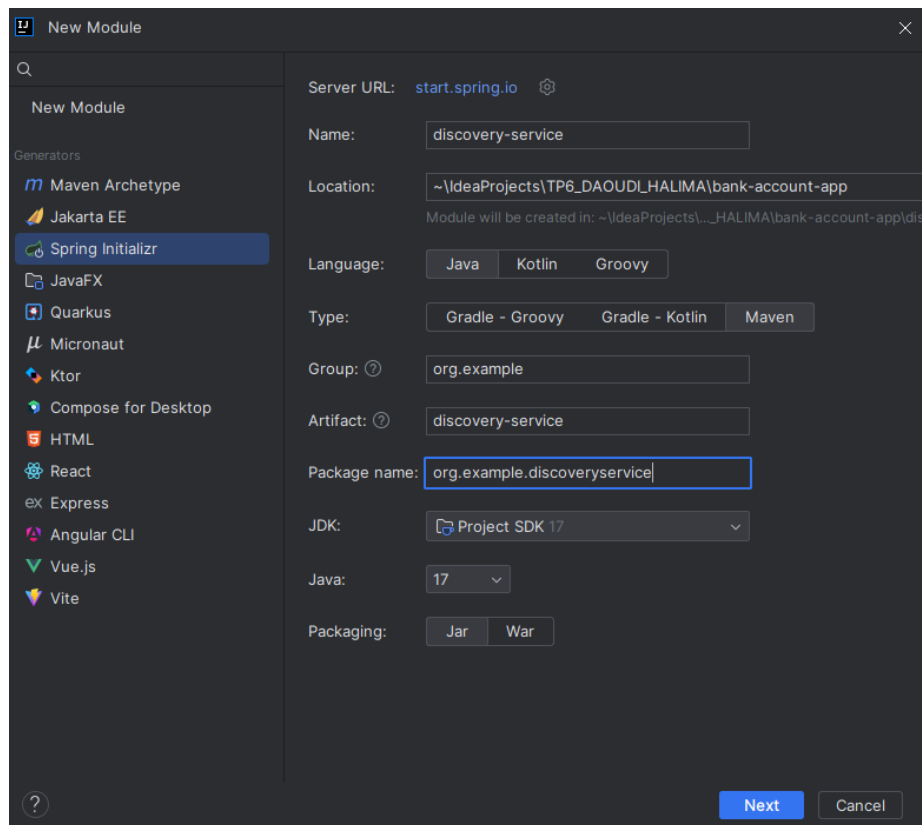
The 'New Module' dialog in IntelliJ IDEA is shown with the 'Spring Initializr' generator selected. The configuration for the 'gateway-service' module is as follows:

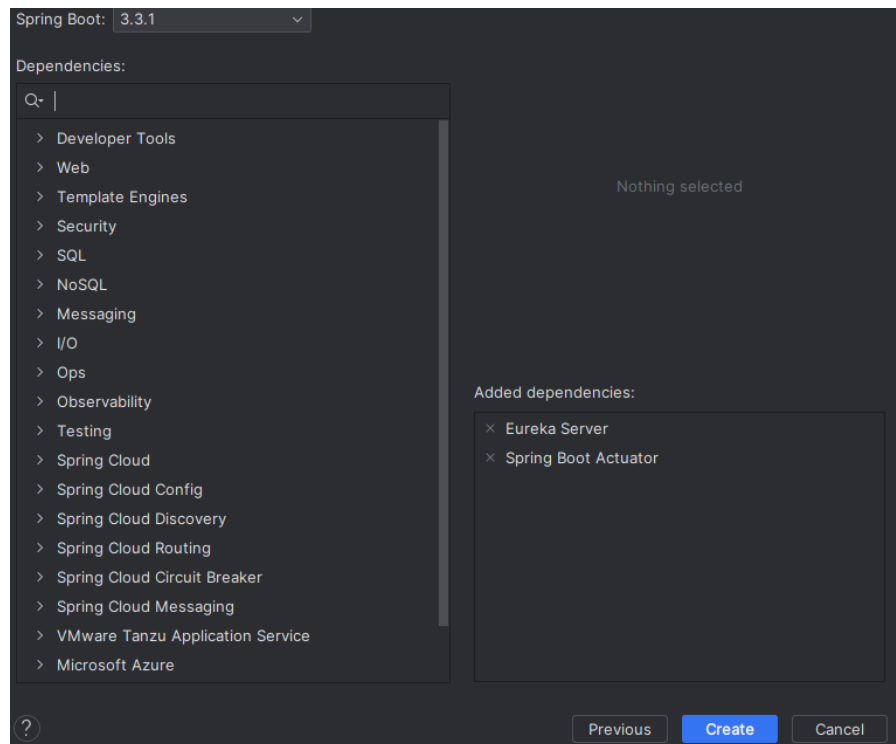
- Server URL: [start.spring.io](https://start.spring.io)
- Name: `gateway-service`
- Location: `~\IdeaProjects\TP6_DAOUDLHALIMA\bank-account-app`  
Module will be created in: `~\IdeaProjects\...DLHALIMA\bank-account-app\...`
- Language: `Java` (selected), `Kotlin`, `Groovy`
- Type: `Gradle - Groovy` (selected), `Gradle - Kotlin`, `Maven`
- Group: `org.example`
- Artifact: `gateway-service`
- Package name: `org.example.gatewayservice`
- JDK: `Project SDK 17`
- Java: `17`
- Packaging: `Jar` (selected), `War`

Buttons: `Next` (highlighted), `Cancel`

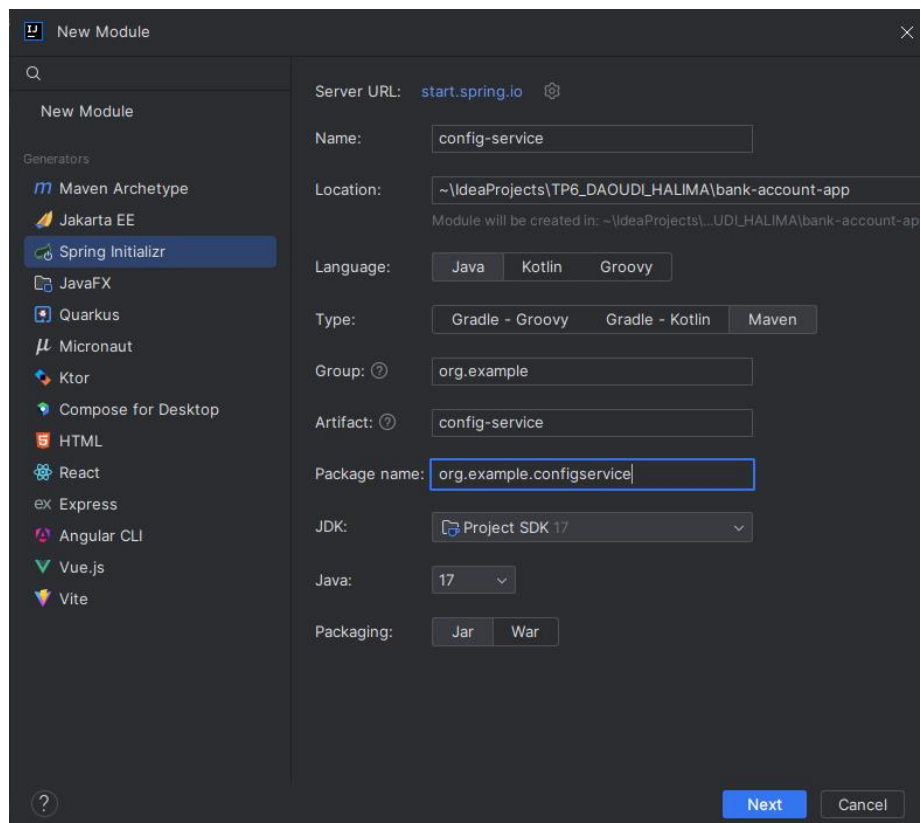


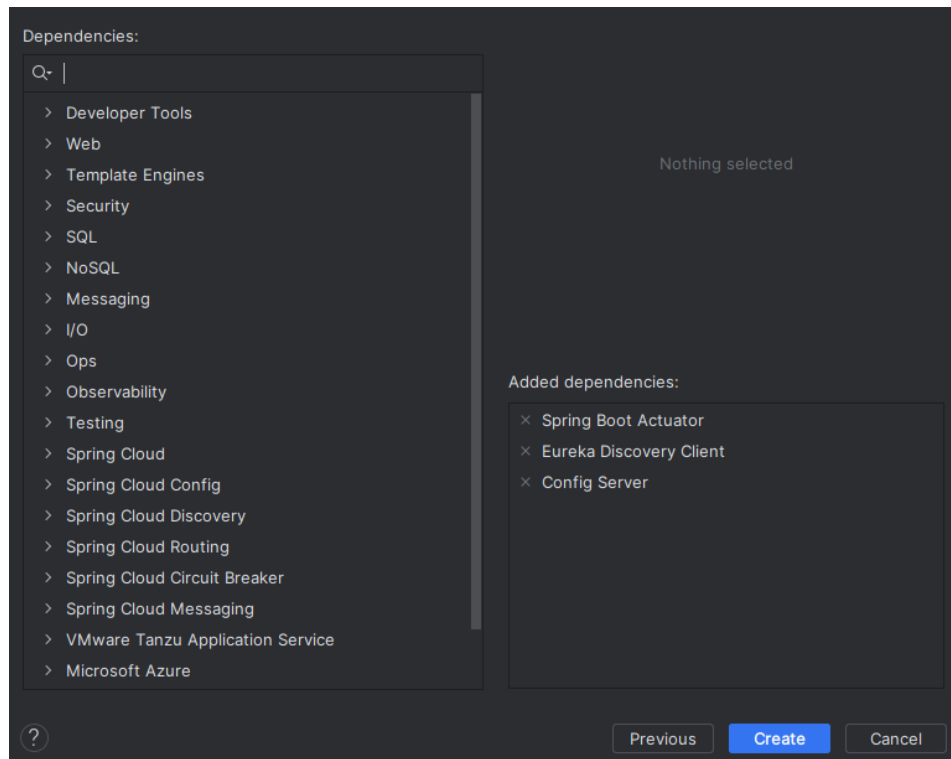
#### 4. discovery-service :



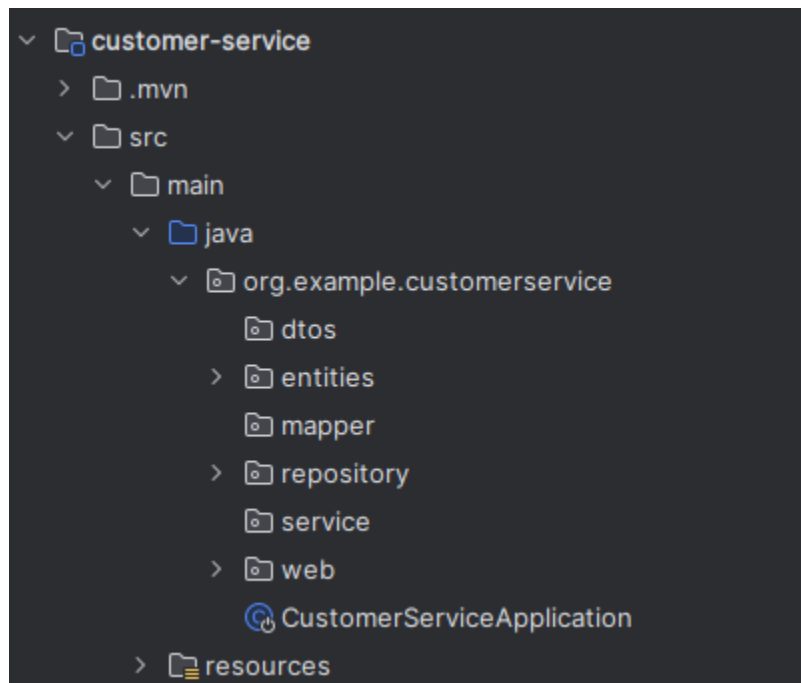


## 5. config-service :





Après avoir structuré le projet en modules distincts pour **customer-service**, je crée les packages suivants :



Dans le package `entities`, je crée la classe `Customer` pour représenter les données d'un client dans l'application.

```
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`, je crée l'interface `CustomerRepository` pour définir les méthodes d'accès aux données des clients.

```
package org.example.customerservice.repository;

import org.example.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, je crée la classe CustomerRestController pour implémenter les endpoints REST qui gèrent les opérations liées aux clients.

```
@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("/{id}")
    public Customer customerById(@PathVariable Long id){
        return customerRepository.findById(id).get();
    }

}
```

Dans la classe CustomerServiceApplication, je crée deux instances de la classe Customer pour représenter des clients dans l'application.

```
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);
    }
}
```

Je configure l'application customer-service en applications.properties avec un nom spécifique, un port défini, une base de données H2 en mémoire, et j'active la console H2 pour le développement.

```
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
```

Je consulte la base de données H2 via l'interface web pour vérifier la présence des deux clients que j'ai ajoutés.

Run Run Selected Auto complete Clear SQL statement:

SELECT \* FROM CUSTOMER

SELECT \* FROM CUSTOMER;

ID	EMAIL	FIRST_NAME	LAST_NAME
1	hibasalmi@gmail.com	Hiba	salmi
2	arabimohammed@gmail.com	Mohammed	arabi

(2 rows, 2 ms)

Edit

Je consulte l'URL <http://localhost:8081/customers/1> pour vérifier la présence du client avec l'identifiant 1 dans mon application.

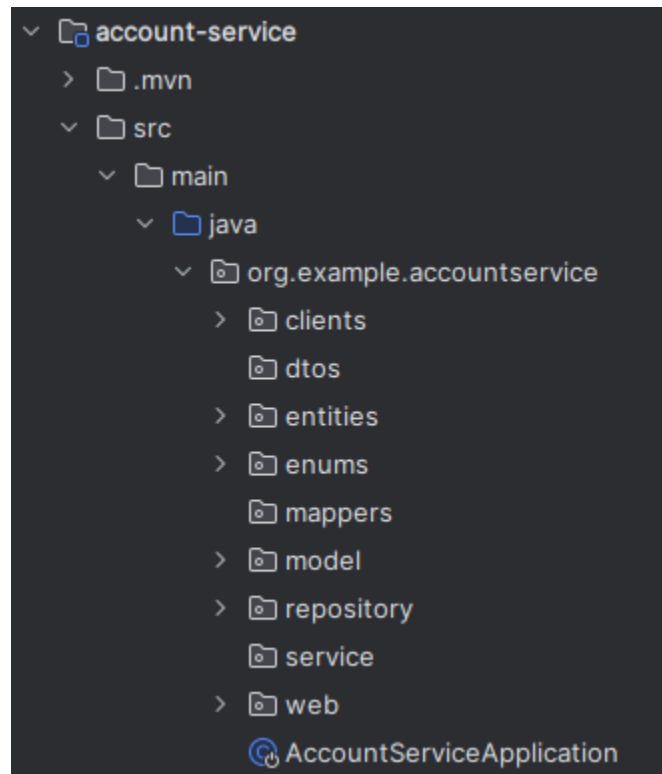
← ↻ ⓘ localhost:8081/customers/1

Gmail YouTube Maps Attestations Créer une instance...

```
1 {
2   "id": 1,
3   "firstName": "Hiba",
4   "lastName": "salmi",
5   "email": "hibasalmi@gmail.com"
6 }
```




Dans le micro-service account-service, je crée les packages suivants :



Dans le package entities, je crée la classe BankAccount pour représenter les données d'un compte bancaire dans l'application.

```
@Entity
@Getter @Setter @ToString @NoArgsConstructor @AllArgsConstructor @Builder
public class BankAccount {
    @Id
    private String accountId;
    private double balance;
    private LocalDate createdAt;
    private String currency;
    @Enumerated(EnumType.STRING)
    private AccountType type;
    @Transient
    private Customer customer;
    private Long customerId;
}
```


Dans le package enums, je crée l'énumération AccountType pour définir les différents types de comptes bancaires disponibles dans l'application.

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

Dans le package model, je crée la classe Customer pour représenter les données d'un client dans l'application.

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

Dans le package web, je crée la classe AccountRestController pour implémenter les endpoints REST qui gèrent les opérations liées aux comptes bancaires.

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

Dans la classe principale de l'application, je crée deux instances de la classe BankAccount :

```
@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(1))
                .build();
            BankAccount bankAccount2 = BankAccount.builder()
                .accountId(UUID.randomUUID().toString())
                .currency("MAD")
                .balance(100000)
                .createAt(LocalDate.now())
                .type(AccountType.SAVING_ACCOUNT)
                .customerId(Long.valueOf(2))
                .build();
            accountRepository.save(bankAccount1);
            accountRepository.save(bankAccount2);
        };
    }
}
```

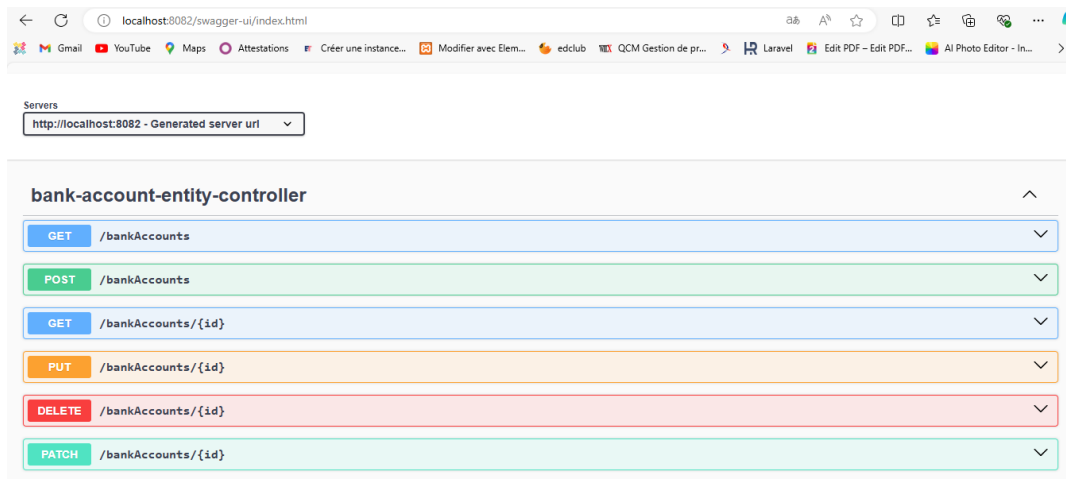
J'ai configuré le fichier application.properties de mon projet avec les paramètres spécifiques nécessaires.

```
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
```

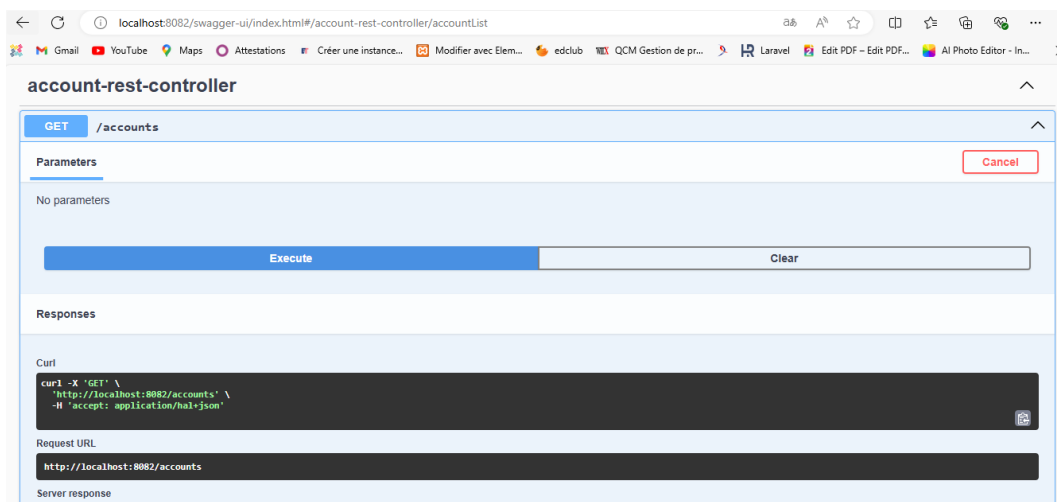
J'ai ajouté la dépendance Springdoc OpenAPI Starter WebMVC dans le fichier pom.xml de mon projet.

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

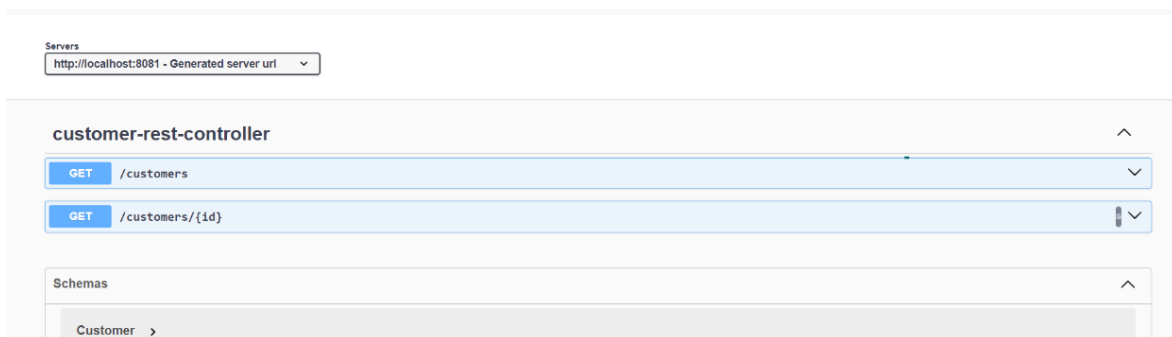
Je consulte <http://localhost:8082/swagger-ui/index.html> pour le microservice account-service



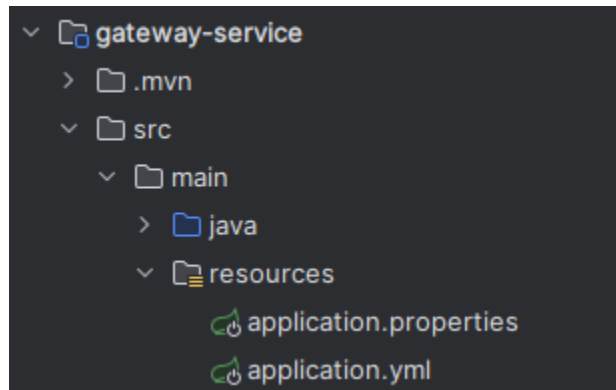
Je teste l'endpoint GET du contrôleur REST des comptes bancaires dans Swagger.



Je consulte <http://localhost:8081/swagger-ui.html> pour le service client (Customer service) et je teste les opérations GET pour récupérer la liste des clients (/customers) ainsi que les détails d'un client spécifique par son identifiant (/customers/{id}).



Dans le microservice Gateway, je crée le fichier `application.yml` pour configurer les paramètres spécifiques.



J'ai configuré un microservice Gateway pour diriger le trafic vers deux autres microservices : le Customer service sur `http://localhost:8081/` avec les chemins commençant par `/customers/**`, et Account service sur `http://localhost:8082/` avec les chemins commençant par `/accounts/**`. L'application est nommée `gateway-service` et elle utilise le port `8888`.

```
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 l'application du microservice Discovery (Discovery Service), j'ai configuré la classe `DiscoveryServiceApplication` avec une méthode `main` qui utilise `SpringApplication.run` pour démarrer l'application. Cela permet au microservice de démarrer et de se préparer à la découverte et à l'enregistrement des autres microservices dans l'environnement.

```
import ...

@SpringBootApplication
@EnableEurekaServer
public class DiscoveryServiceApplication {

    public static void main(String[] args) {

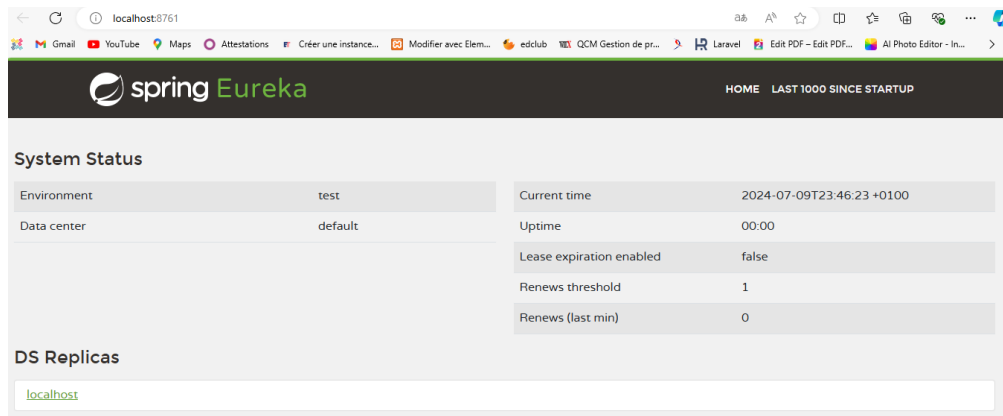
        SpringApplication.run(DiscoveryServiceApplication.class, args);
    }

}
```

J'ai configuré l'application du microservice Discovery (Discovery Service) avec les propriétés suivantes dans le fichier `application.properties` :

```
spring.application.name=discovery-service
server.port=8761
eureka.client.fetch-registry=false
eureka.client.register-with-eureka=false
```

Je consulte `http://localhost:8761` pour visualiser le tableau de bord Eureka et vérifier l'enregistrement des microservices.



The screenshot shows the Spring Eureka dashboard in a web browser. The browser's address bar displays `localhost:8761`. The dashboard has a dark header with the "spring Eureka" logo and navigation links for "HOME" and "LAST 1000 SINCE STARTUP".

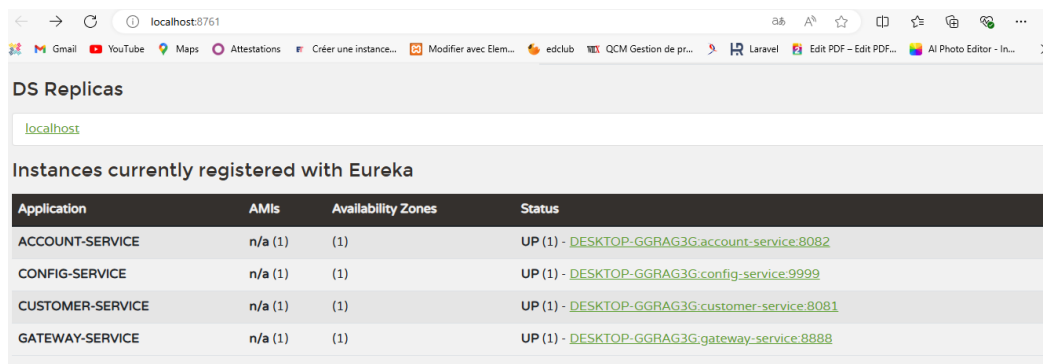
**System Status**

Environment	test	Current time	2024-07-09T23:46:23 +0100
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

**DS Replicas**

localhost
-----------

J'ajoute la propriété `discovery=true` pour activer la découverte des services dans le gateway.



DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACCOUNT-SERVICE	n/a (1)	(1)	UP (1) - <a href="http://DESKTOP-GGRAG3G:8082">DESKTOP-GGRAG3G:account-service:8082</a>
CONFIG-SERVICE	n/a (1)	(1)	UP (1) - <a href="http://DESKTOP-GGRAG3G:9999">DESKTOP-GGRAG3G:config-service:9999</a>
CUSTOMER-SERVICE	n/a (1)	(1)	UP (1) - <a href="http://DESKTOP-GGRAG3G:8081">DESKTOP-GGRAG3G:customer-service:8081</a>
GATEWAY-SERVICE	n/a (1)	(1)	UP (1) - <a href="http://DESKTOP-GGRAG3G:8888">DESKTOP-GGRAG3G:gateway-service:8888</a>

Dans le service client (Customer Service), j'ai modifié le fichier `application.properties` avec les configurations suivantes :

```
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
```

Dans le service des comptes (Account Service), j'ai modifié le fichier `application.properties` avec les configurations suivantes :

```
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
```

Dans le service Gateway (Gateway Service), j'ai modifié le fichier `application.properties` avec les configurations suivantes :

```
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 :

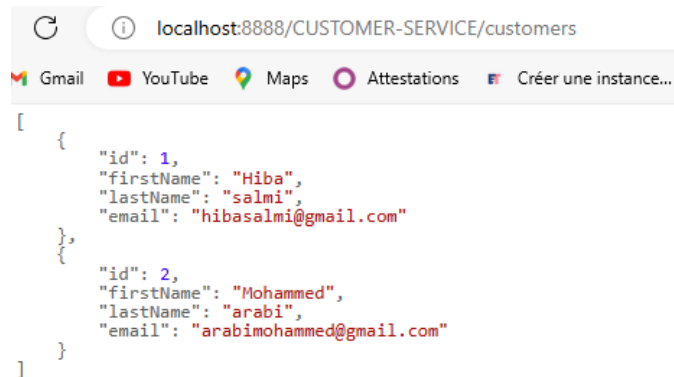
```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.ReactiveDiscoveryClient;
import org.springframework.cloud.gateway.discovery.DiscoveryClientRouteDefinitionLocator;
import org.springframework.cloud.gateway.discovery.DiscoveryLocatorProperties;
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);
    }
}
```

Je consulte <http://localhost:8888/CUSTOMER-SERVICE/customers> pour accéder à l'endpoint des clients via le service Gateway.



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

Dans le service des comptes (Account Service), j'ajoute la dépendance spring-cloud-starter-openfeign dans le fichier pom.xml.

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

Je crée le package clients dans le service des comptes (Account Service) pour définir les interfaces Feign Client.

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

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

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



Dans AccountRestController, j'ajoute l'utilisation de Feign Client pour interagir avec d'autres microservices via les interfaces définies dans le package clients.

```
@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;
}
```

J'ajoute la dépendance Spring Starter Circuit Breaker Resilience4j dans votre projet Account Service

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

Je modifie CustomerRestClient pour intégrer la gestion de circuit breaker avec Resilience4j, en utilisant des annotations comme @CircuitBreaker pour sécuriser les appels au client CustomerServiceClient.

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

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

    @GetMapping("/{customers}")
    @CircuitBreaker(name = "customerService", fallbackMethod = "getAllCustomers")
    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;
    }
}
```

J'ajuste les tests de l'application dans AccountServiceApplicationTests

```
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);
    });
};
```

localhost:8888/ACCOUNT-SERVICE/accounts

Gmail YouTube Maps Attestations Créer une instance...

```
[
  {
    "accountId": "439ad6ee-7349-4b7f-af3d-b7af240b755c",
    "balance": 4427.708965234443,
    "createAt": "2024-07-10",
    "currency": "MAD",
    "type": "CURRENT_ACCOUNT",
    "customer": {
      "id": 1,
      "firstName": "Hiba",
      "lastName": "salmi",
      "email": "hibasalmi@gmail.com"
    },
    "customerId": 1
  },
  {
    "accountId": "3c724163-0204-4459-a244-5eaf2d0d1b2",
    "balance": 737.7578338750967,
    "createAt": "2024-07-10",
    "currency": "MAD",
    "type": "SAVING_ACCOUNT",
    "customer": {
      "id": 1,
      "firstName": "Hiba",
      "lastName": "salmi",
      "email": "hibasalmi@gmail.com"
    },
    "customerId": 1
  }
]
```