



THEME

Activité Pratique N° 5 : Projet Spring Angular JWT , Digital Banking

Réalisé Par :

Manal Namir

Encadré Par :

Mohamed YOUSSEFI

Ce rapport contient 7 parties :

- ✓ Dans la première partie, nous avons introduit le projet, en expliquant les parties frontend et backend de l'application.
- ✓ La deuxième partie a abordé l'installation des dépendances nécessaires et a fourni un diagramme de classes qui illustre les relations entre les différentes entités du projet, ainsi qu'une explication de l'architecture globale de l'application.
- ✓ La troisième partie a mis l'accent sur la couche DAO (Data Access Object), qui permet la gestion des opérations de persistance des données. Nous avons détaillé les différentes interfaces et classes implémentées pour accéder aux données de manière efficace et sécurisée.
- ✓ La quatrième partie a couvert les couches services, DTOs et RestController. Les services assurent la logique métier de l'application, les DTOs facilitent le transfert des données entre les différentes couches et les RestControllers exposent les points d'entrée de l'API REST.
- ✓ La cinquième partie a présenté le client Angular de l'application, en mettant en évidence les fonctionnalités offertes aux utilisateurs, telles que la gestion des clients, des comptes bancaires et des opérations financières.
- ✓ La sixième partie a regroupé les différentes interfaces du projet, en les décrivant brièvement et en soulignant leur rôle et leur interaction dans l'ensemble de l'application.
- ✓ Et finalement on a une conclusion générale.

Introduction :

Parties 1 et 2 présente la partie backend d'une application web de gestion des comptes bancaires appelée "Digital-Banking-master". L'application est développée en utilisant le framework Spring Boot.

Parties 3 et 4 présente la partie frontend de l'application, développée en utilisant le framework Angular.

L'application propose différentes fonctionnalités pour gérer les opérations bancaires en ligne, y compris la consultation des comptes, la gestion des clients et les opérations financières telles que le débit, le crédit et le transfert.

Concernant le projet :

1. L'installation :

Partie Backend :

SDK : 1.8 Oracle OpenJDK version 18.0.2

Java : 17

Type : Maven

Les dépendances utilisées :

- ✓ Spring Web : pour créer des applications Web, y compris RESTful, à l'aide de Spring MVC. Utilise Apache Tomcat comme conteneur intégré par défaut.
- ✓ Spring DATA JPA : Pour conserver les données dans 'SQL stores' avec Java Persistance API à l'aide de Spring Data et Hibernate. C'est un module qui facilite le ORM basé sur JPA. Il est utilisé avec les bases de données relationnelles.
- ✓ MySQL : MySQL est l'un des systèmes de bases de données relationnelles les plus populaires et il est souvent utilisé dans les applications Spring Boot.
- ✓

- ✓ Lombok : Un outil de bibliothèque Java qui génère du code pour minimiser le code ‘boilerplate’. La bibliothèque remplace le code ‘boilerplate’ par des annotations faciles à utiliser (Exemples : @NoArgsConstructor, @Getter, @Setter...).
- ✓ Spring-boot-devtools : une dépendance Spring Boot qui fournit des outils de développement pratiques pour le développement de vos applications Spring Boot. Cette dépendance a été utilisée dans cette activité pour :
- ✓ Le mécanisme de rechargement automatique des classes en mode développement;
- ✓ Le serveur embarqué qui permet de redémarrer l’application automatiquement lorsqu’un changement est détecté.

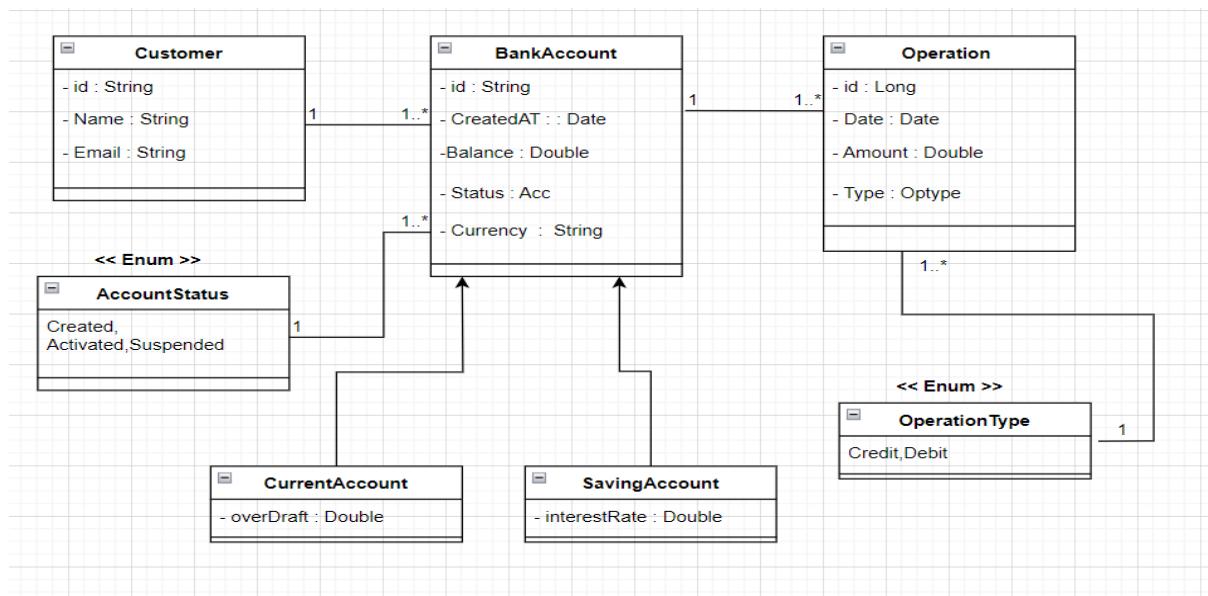
Partie Backend :

Angular CLI version 16.0.1

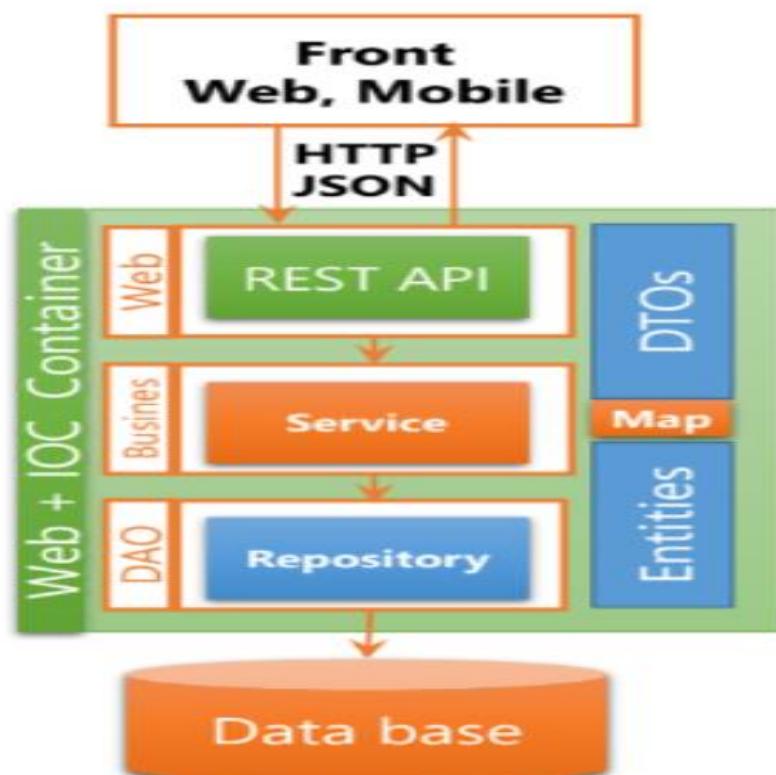
Installation :

Bootstrap & Bootstrap-icons : Une bibliothèque open-source de développement front-end pour la conception de sites et d'applications web. Elle fournit des styles CSS prédéfinis, des composants JavaScript et des icônes pour faciliter la création d'interfaces utilisateur esthétiques et responsives (npm i bootstrap bootstrap-icon).

2. Diagramme de classe :



3. Architecture de l'application :

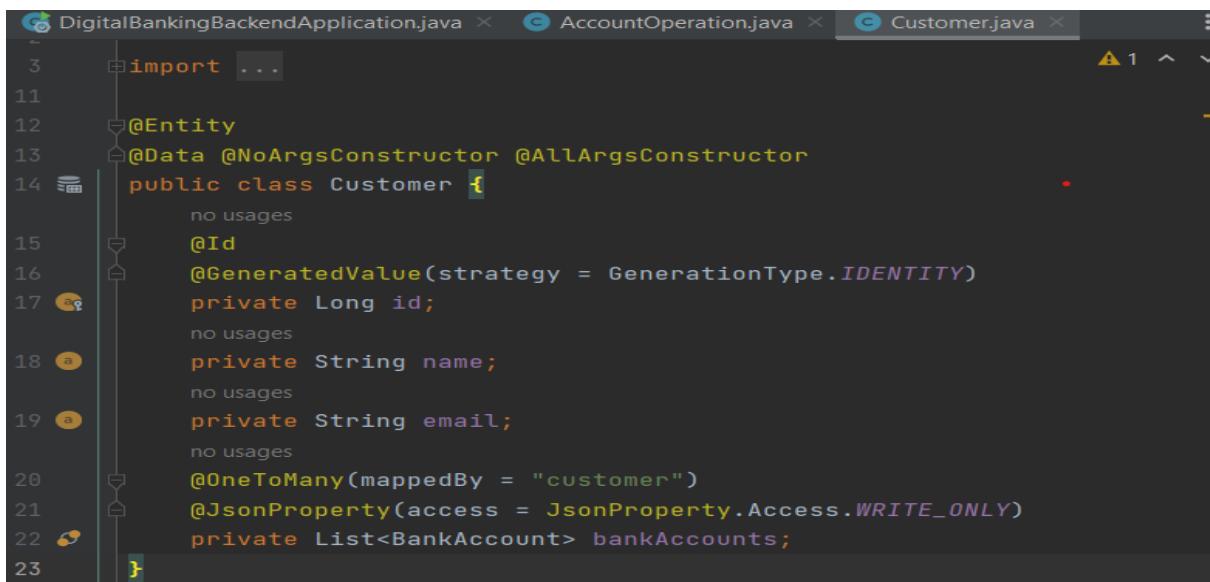


Partie 1 : Couche DAO

1. On a créé les entités JPA :

Les entités JPA représentent les objets métier de l'application et sont mappées aux tables de la base de données. Les entités de l'application Digital Banking sont les suivantes :

- Customer :



```
DigitalBankingBackendApplication.java × AccountOperation.java × Customer.java ×
3 import ...
11
12 @Entity
13 @Data @NoArgsConstructor @AllArgsConstructor
14 public class Customer {
15     no usages
16     @Id
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     private Long id;
19     no usages
20     private String name;
21     no usages
22     private String email;
23     no usages
24     @OneToMany(mappedBy = "customer")
25     @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
26     private List<BankAccount> bankAccounts;
```

- BankAccount :

```

BankAccount.java × Customer.java × BankAccount.java × BankAccountNotFound.java ×
import java.util.Date;
import java.util.List;
2 inheritors
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "TYPE", length = 4)
@Data @NoArgsConstructor @AllArgsConstructor
public abstract class BankAccount {
    no usages
    @Id
    private String id;
    no usages
    private double balance;
    no usages
    private Date createdAt;
    no usages
    @Enumerated(EnumType.STRING)
    private AccountStatus status;
    no usages
    @ManyToOne
    private Customer customer;
    no usages
    @OneToMany(mappedBy = "bankAccount", fetch = FetchType.LAZY)
    private List<AccountOperation> accountOperations;
}

}

```

➤ Saving Account :

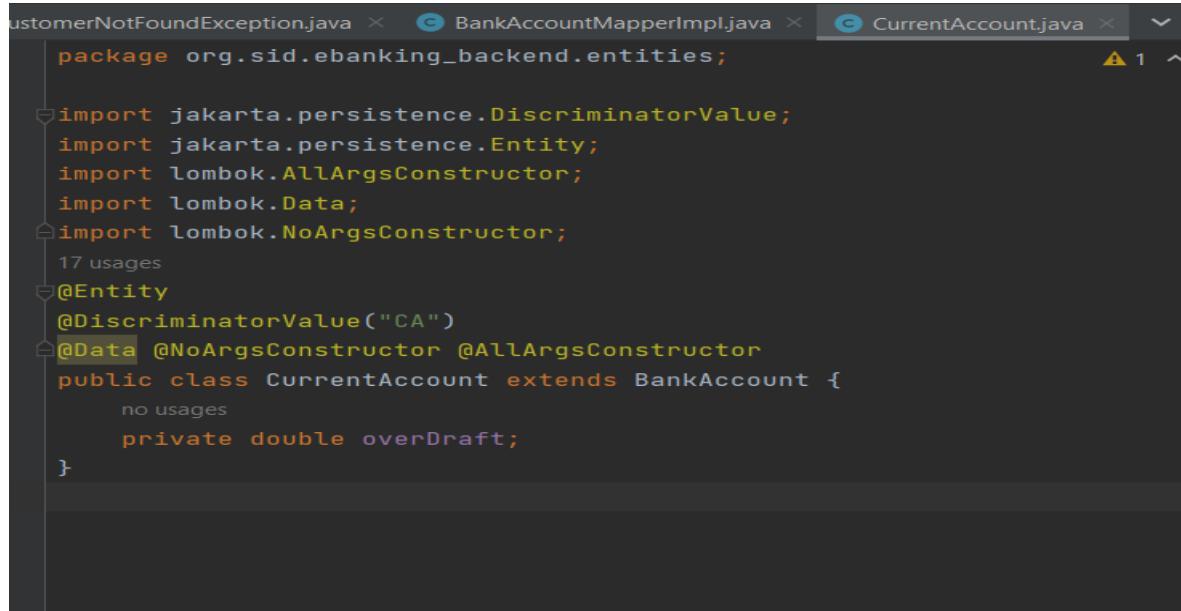
```

× BankAccount.java × SavingAccount.java × BankAccountNotFound.java ×
package org.sid.ebanking_backend.entities;

import jakarta.persistence.DiscriminatorValue;
import jakarta.persistence.Entity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
20 usages
@Entity
@DiscriminatorValue("SA")
@Data @NoArgsConstructor @AllArgsConstructor
public class SavingAccount extends BankAccount{
    no usages
    private double interestRate;
}

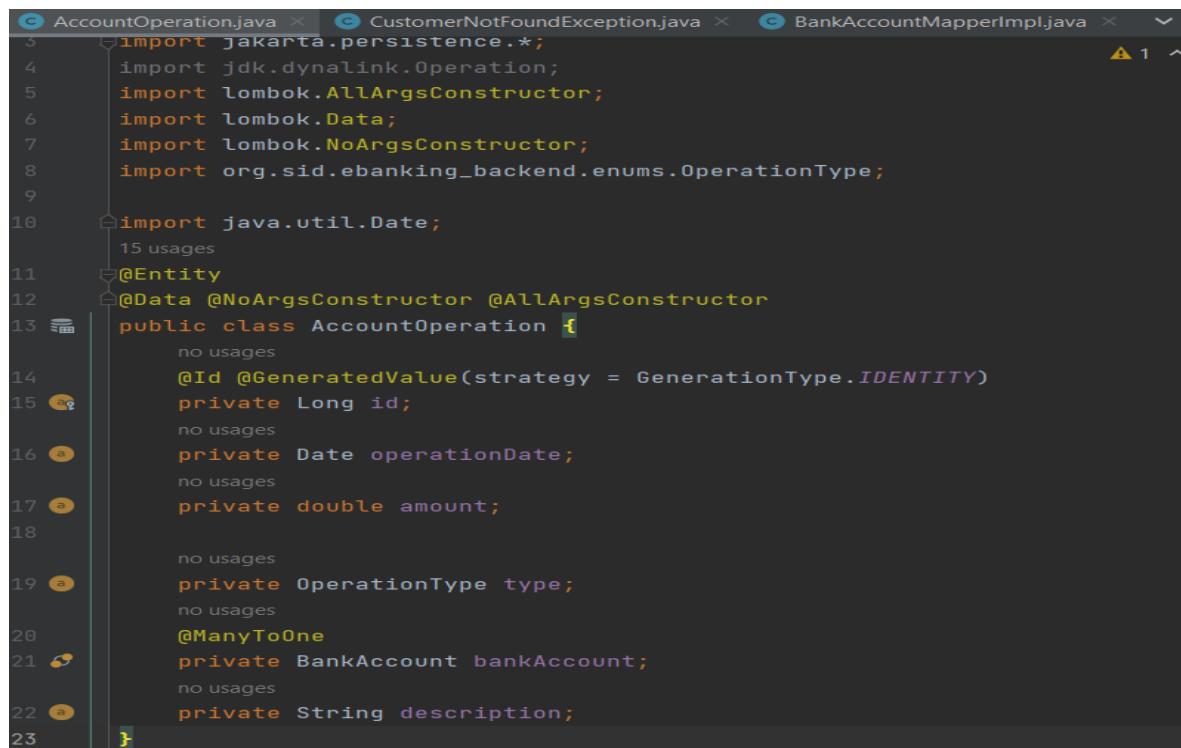
```

➤ CurrentAccount :



```
CustomerNotFoundException.java ×  BankAccountMapperImpl.java ×  CurrentAccount.java ×  ▾ 1 ^  
package org.sid.ebanking_backend.entities;  
  
import jakarta.persistence.DiscriminatorValue;  
import jakarta.persistence.Entity;  
import lombok.AllArgsConstructorConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructorConstructor;  
17 usages  
@Entity  
@DiscriminatorValue("CA")  
@Data @NoArgsConstructorConstructor @AllArgsConstructorConstructor  
public class CurrentAccount extends BankAccount {  
    no usages  
    private double overDraft;  
}
```

➤ AccountOperation :

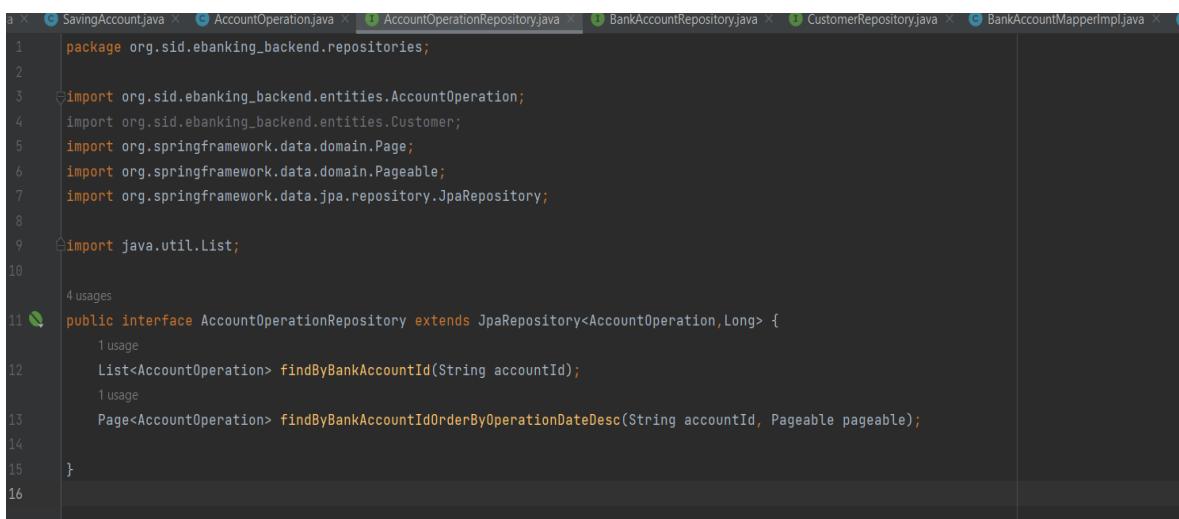


```
AccountOperation.java ×  CustomerNotFoundException.java ×  BankAccountMapperImpl.java ×  ▾ 1 ^  
import jakarta.persistence.*;  
import jdk.dynalink.Operation;  
import lombok.AllArgsConstructorConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructorConstructor;  
import org.sid.ebanking_backend.enums.OperationType;  
  
import java.util.Date;  
15 usages  
@Entity  
@Data @NoArgsConstructorConstructor @AllArgsConstructorConstructor  
public class AccountOperation {  
    no usages  
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    no usages  
    private Date operationDate;  
    no usages  
    private double amount;  
  
    no usages  
    private OperationType type;  
    no usages  
    @ManyToOne  
    private BankAccount bankAccount;  
    no usages  
    private String description;
```

2. On a créé les interfaces JPA Repository basées sur Spring Data :

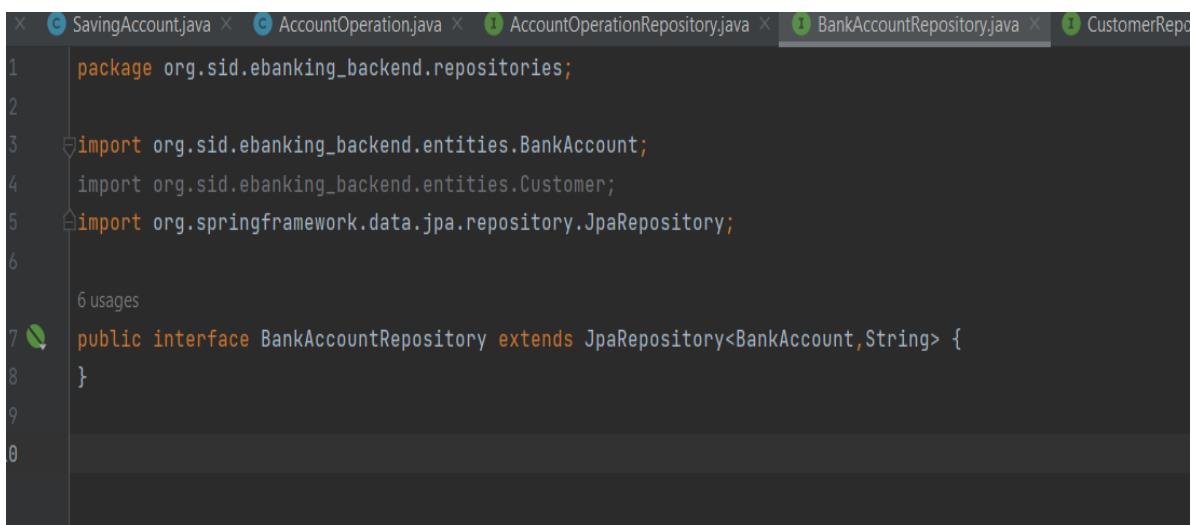
Cette couche gère l'accès aux données et comprend les interfaces JPA Repository basées sur Spring Data. Ces interfaces fournissent les opérations de base pour interagir avec les entités persistantes.

➤ AccountOperationRepository :



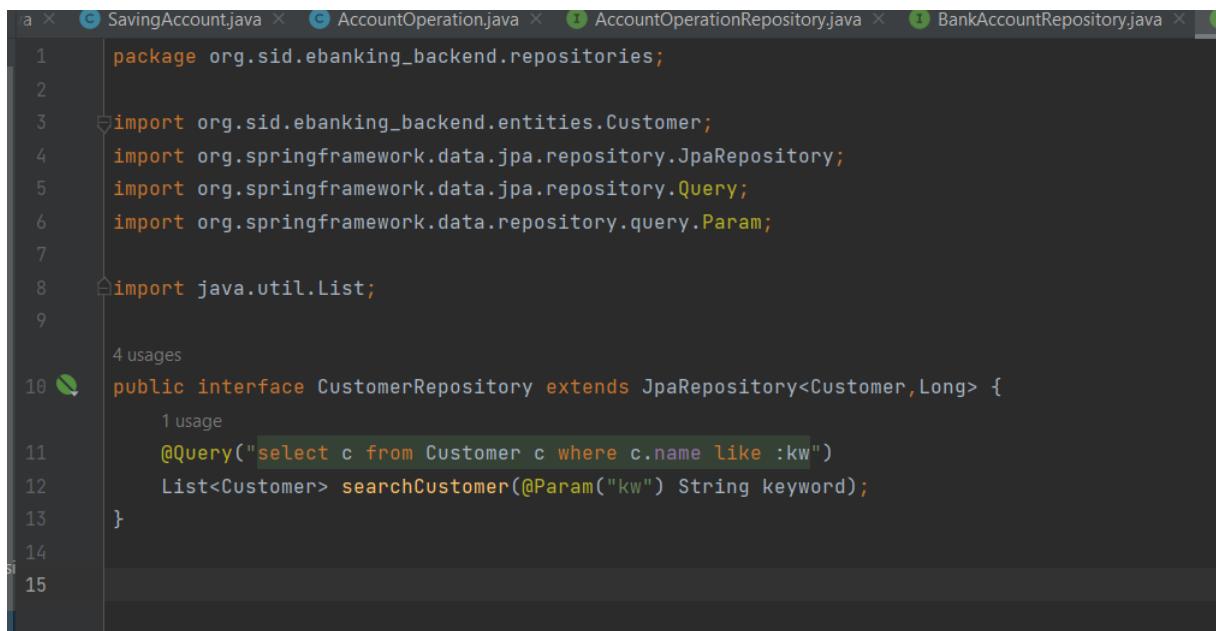
```
1 package org.sid.ebanking_backend.repositories;
2
3 import org.sid.ebanking_backend.entities.AccountOperation;
4 import org.sid.ebanking_backend.entities.Customer;
5 import org.springframework.data.domain.Page;
6 import org.springframework.data.domain.Pageable;
7 import org.springframework.data.jpa.repository.JpaRepository;
8
9 import java.util.List;
10
11 public interface AccountOperationRepository extends JpaRepository<AccountOperation, Long> {
12     List<AccountOperation> findByBankAccountId(String accountId);
13     Page<AccountOperation> findByBankAccountIdOrderByOperationDateDesc(String accountId, Pageable pageable);
14 }
15
16 }
```

➤ BankAccountRepository :



```
1 package org.sid.ebanking_backend.repositories;
2
3 import org.sid.ebanking_backend.entities.BankAccount;
4 import org.sid.ebanking_backend.entities.Customer;
5 import org.springframework.data.jpa.repository.JpaRepository;
6
7 public interface BankAccountRepository extends JpaRepository<BankAccount, String> {
```

➤ CustomerRepository :



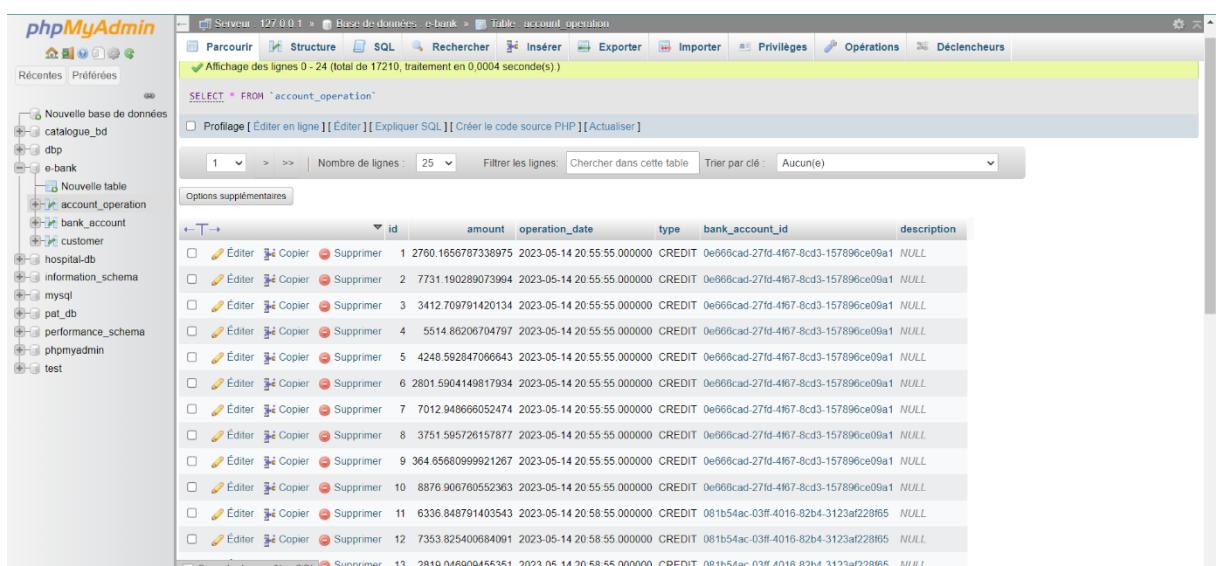
```

1 package org.sid.ebanking_backend.repositories;
2
3 import org.sid.ebanking_backend.entities.Customer;
4 import org.springframework.data.jpa.repository.JpaRepository;
5 import org.springframework.data.jpa.repository.Query;
6 import org.springframework.data.repository.query.Param;
7
8 import java.util.List;
9
10 public interface CustomerRepository extends JpaRepository<Customer, Long> {
11     @Query("select c from Customer c where c.name like :kw")
12     List<Customer> searchCustomer(@Param("kw") String keyword);
13 }
14
15

```

3. On les 3 tables :

➤ Accountoperation:



	Éditer	Copier	Supprimer	id	amount	operation_date	type	bank_account_id	description
<input type="checkbox"/>	Éditer	Copier	Supprimer	1	2760.165677336975	2023-05-14 20:55:55.000000	CREDIT	0e666cad-27fd-4f67-8cd3-157896ce09a1	NULL
<input type="checkbox"/>	Éditer	Copier	Supprimer	2	7731.190289073994	2023-05-14 20:55:55.000000	CREDIT	0e666cad-27fd-4f67-8cd3-157896ce09a1	NULL
<input type="checkbox"/>	Éditer	Copier	Supprimer	3	3412.709791420134	2023-05-14 20:55:55.000000	CREDIT	0e666cad-27fd-4f67-8cd3-157896ce09a1	NULL
<input type="checkbox"/>	Éditer	Copier	Supprimer	4	5514.80206704797	2023-05-14 20:55:55.000000	CREDIT	0e666cad-27fd-4f67-8cd3-157896ce09a1	NULL
<input type="checkbox"/>	Éditer	Copier	Supprimer	5	4248.592847066643	2023-05-14 20:55:55.000000	CREDIT	0e666cad-27fd-4f67-8cd3-157896ce09a1	NULL
<input type="checkbox"/>	Éditer	Copier	Supprimer	6	2801.5904149817934	2023-05-14 20:55:55.000000	CREDIT	0e666cad-27fd-4f67-8cd3-157896ce09a1	NULL
<input type="checkbox"/>	Éditer	Copier	Supprimer	7	7012.948666052474	2023-05-14 20:55:55.000000	CREDIT	0e666cad-27fd-4f67-8cd3-157896ce09a1	NULL
<input type="checkbox"/>	Éditer	Copier	Supprimer	8	3751.595726157877	2023-05-14 20:55:55.000000	CREDIT	0e666cad-27fd-4f67-8cd3-157896ce09a1	NULL
<input type="checkbox"/>	Éditer	Copier	Supprimer	9	364.65680999921267	2023-05-14 20:55:55.000000	CREDIT	0e666cad-27fd-4f67-8cd3-157896ce09a1	NULL
<input type="checkbox"/>	Éditer	Copier	Supprimer	10	8876.906760552363	2023-05-14 20:55:55.000000	CREDIT	0e666cad-27fd-4f67-8cd3-157896ce09a1	NULL
<input type="checkbox"/>	Éditer	Copier	Supprimer	11	6336.848791403543	2023-05-14 20:58:55.000000	CREDIT	081b54ac-03ff-4016-82b4-3123af228f65	NULL
<input type="checkbox"/>	Éditer	Copier	Supprimer	12	7353.825400684091	2023-05-14 20:58:55.000000	CREDIT	081b54ac-03ff-4016-82b4-3123af228f65	NULL
<input type="checkbox"/>	Console de requêtes SQL	Supprimer		13	2819.046909455351	2023-05-14 20:58:55.000000	CREDIT	081b54ac-03ff-4016-82b4-3123af228f65	NULL

➤ Bank-account :

phpMyAdmin

Parcourir Structure SQL Rechercher Insérer Exporter Importer Priviléges Opérations Déclencheurs

Affichage des lignes 0 - 24 (total de 258, traitement en 0,0007 seconde(s))

```
SELECT * FROM `bank_account`
```

Profilage [Éditer en ligne] [Éditer] [Expliquer SQL] [Créer le code source PHP] [Actualiser]

1 > >> Tout afficher Nombre de lignes : 25 Filtrer les lignes Chercher dans cette table Trier par clé : Aucun(e)

Options supplémentaires

	type	id	balance	created_at	status	over_draft	interest_rate	customer_id
<input type="checkbox"/>	SA	0014626e-1231-4a5c-9de7-f6d4b124bf05	1465049.7024269234	2023-06-23 15:40:12.000000	NULL	NULL	5.5	20
<input type="checkbox"/>	CA	00419bb7-8179-41e3-aa2d-1090ce666d530	4028963.078419007	2023-05-15 21:44:55.000000	NULL	9000	NULL	9
<input type="checkbox"/>	SA	007e8a0d-213f-4c1-b90a-82fbe96a785f	2577738.378424148	2023-06-23 15:35:51.000000	NULL	NULL	5.5	5
<input type="checkbox"/>	CA	009a9908-0f03-4ab9-944e-835ee9e3864b	3764740.02270692958	2023-05-15 21:44:55.000000	NULL	9000	NULL	4
<input type="checkbox"/>	SA	01884796-e0ed-4118-bf3b-87194c1652a5	710940.7087227578	2023-06-23 18:19:59.000000	NULL	NULL	5.5	8
<input type="checkbox"/>	CA	02563505-2a8-4aff-bee9-08615af8a64f	3944865.8375442554	2023-05-15 21:44:55.000000	NULL	9000	NULL	1
<input type="checkbox"/>	CA	03e8c7af-ea13-4bd1-96b3-1080bd0fd42a	846741.04867585762	2023-06-23 18:19:59.000000	NULL	9000	NULL	12
<input type="checkbox"/>	SA	03ee081f-fe54-4f87-b7be-7ba13e0d0d24	4315781.04816322	2023-05-15 21:44:55.000000	NULL	NULL	5.5	2
<input type="checkbox"/>	CA	05c3ea13-5c46-4e15-83be-2e42ce08ad19	607152.7137750192	2023-06-23 18:19:59.000000	NULL	9000	NULL	17
<input type="checkbox"/>	CA	062f7b07-12a6-4297-91dd-16d65f5481448	3232448.728399962	2023-05-23 14:22:53.000000	NULL	9000	NULL	15
<input type="checkbox"/>	SA	06ba12a8-1b0b-4731-86bf-d9f99decea	3765063.6568421167	2023-05-15 21:44:55.000000	NULL	NULL	5.5	11
<input type="checkbox"/>	CA	07da620c-f76f-43f4-a5c5-09632bb14035	722776.2117758008	2023-06-23 18:19:59.000000	NULL	9000	NULL	25
Console de requêtes SQL								

➤ Customer :

phpMyAdmin

Parcourir Structure SQL Rechercher Insérer Exporter Importer Priviléges Opérations Déclencheurs

Options supplémentaires

	id	email	name
<input type="checkbox"/>	1	Manal@gmail	Manal
<input type="checkbox"/>	2	Imane@gmail	Imane
<input type="checkbox"/>	3	Othmane@gmail	Othmane
<input type="checkbox"/>	4	Nabila@gmail	Nabila
<input type="checkbox"/>	5	Manal@gmail	Manal
<input type="checkbox"/>	6	Imane@gmail	Imane
<input type="checkbox"/>	7	Othmane@gmail	Othmane
<input type="checkbox"/>	8	Nabila@gmail	Nabila
<input type="checkbox"/>	9	Hassan@gmail.com	Hassan
<input type="checkbox"/>	10	Imane@gmail.com	Imane
<input type="checkbox"/>	11	Mohamed@gmail.com	Mohamed
<input type="checkbox"/>	12	Manal@gmail.com	Manal
<input type="checkbox"/>	13	Imane@gmail.com	Imane
<input type="checkbox"/>	14	Nabila@gmail.com	Nabila
<input type="checkbox"/>	15	Othmane@gmail.com	Othmane
<input type="checkbox"/>	16	Leila@gmail.com	Leila

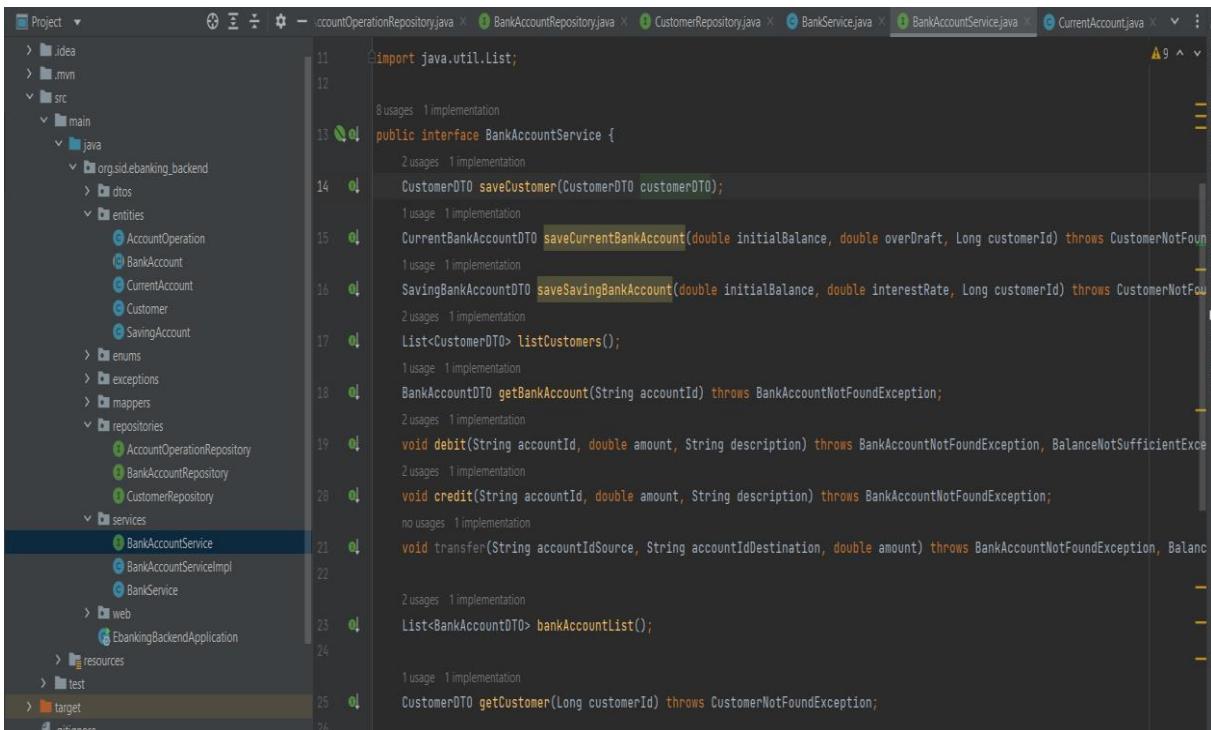
Partie 2 : Couche service, DTOs et RestController

1. On a créé la couche service, DTOs et RestController :

➤ La couche service :

Cette couche contient la logique métier de l'application et offre des services pour effectuer des opérations sur les comptes bancaires. Elle utilise les interfaces DAO pour accéder aux données et les mappers pour convertir les entités en DTO (Data Transfer Object) et vice versa.

On a le dossier services qui contient trois fichiers (BankAccountService , BankAccountServiceImpl, BankService) j'ai pris comme exemple le fichier BankAccountService :



```

import java.util.List;

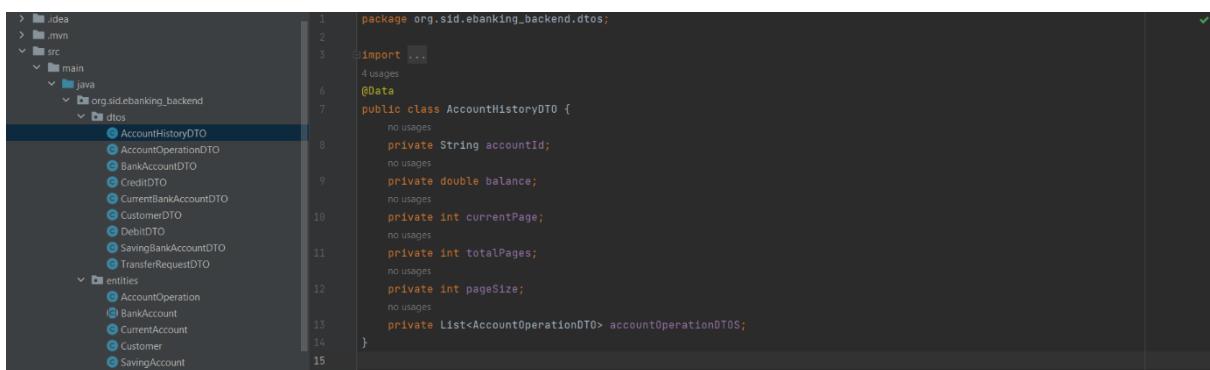
public interface BankAccountService {
    CustomerDTO saveCustomer(CustomerDTO customerDTO);
    CurrentBankAccountDTO saveCurrentBankAccount(double initialBalance, double overDraft, Long customerId) throws CustomerNotFoundException;
    SavingBankAccountDTO saveSavingBankAccount(double initialBalance, double interestRate, Long customerId) throws CustomerNotFoundException;
    List<CustomerDTO> listCustomers();
    BankAccountDTO getBankAccount(String accountId) throws BankAccountNotFoundException;
    void debit(String accountId, double amount, String description) throws BankAccountNotFoundException, BalanceNotSufficientException;
    void credit(String accountId, double amount, String description) throws BankAccountNotFoundException;
    void transfer(String accountIdSource, String accountIdDestination, double amount) throws BankAccountNotFoundException, BalanceNotSufficientException;
    List<BankAccountDTO> bankAccountList();
    CustomerDTO getCustomer(Long customerId) throws CustomerNotFoundException;
}

```

➤ La couche DTOs :

Les DTOs sont des objets utilisés pour transférer des données entre les différentes couches de l'application. Ils permettent de définir les informations à envoyer ou à recevoir lors des appels API.

On a le dossier Dtos qui contient 9 fichiers (CreditDTO, DebitDTO,..), j'ai pris comme exemple le fichier AccountHistoryDTO :



```

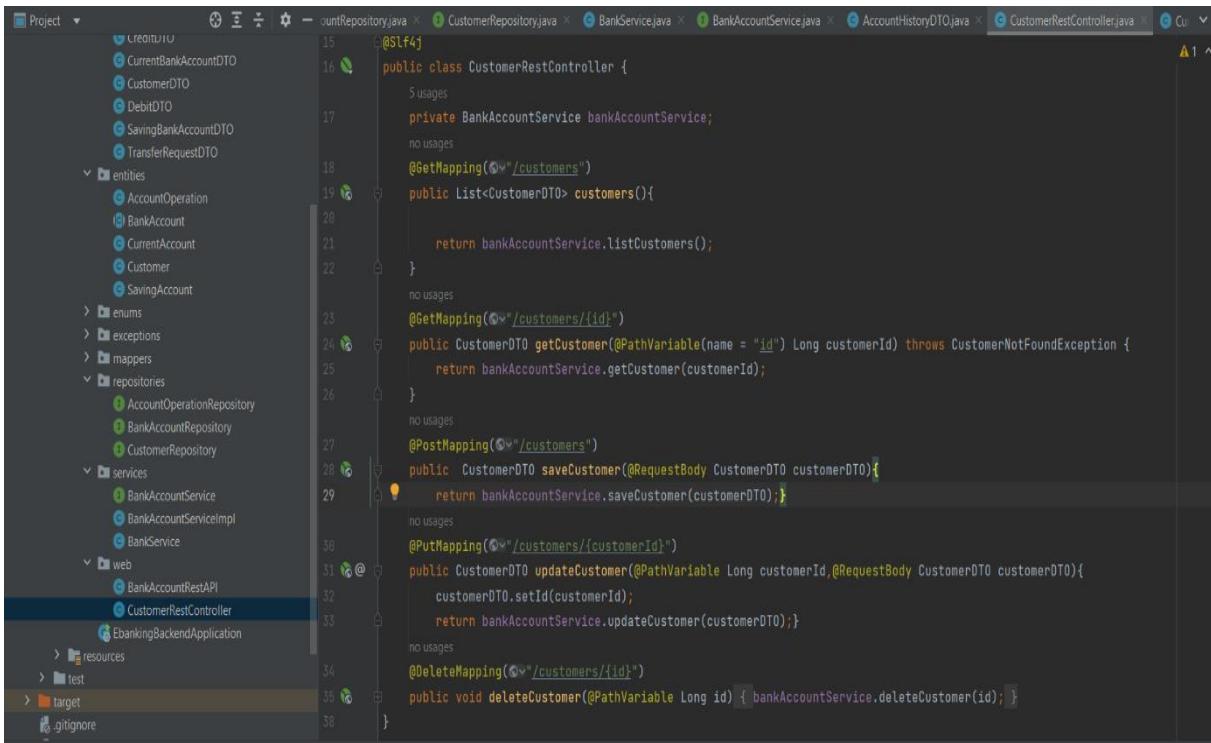
1 package org.sid.ebanking_backend.dtos;
2
3 import ...
4 usages
5
6 @Data
7 public class AccountHistoryDTO {
8     no usages
9     private String accountId;
10    no usages
11    private double balance;
12    no usages
13    private int currentPage;
14    no usages
15    private int totalPages;
16    no usages
17    private int pageSize;
18    no usages
19    private List<AccountOperationDTO> accountOperationDTOS;
20 }

```

➤ La couche RestController :

Les RestControllers sont des composants qui exposent les services de l'application via des API REST. Ils reçoivent les requêtes HTTP, appellent les services appropriés et renvoient les réponses aux clients.

On a le dossier web qui contient deux fichiers (BankAccountRestAPI , CustomerRestController) j'ai pris comme exemple le fichier CustomerRestController :



```

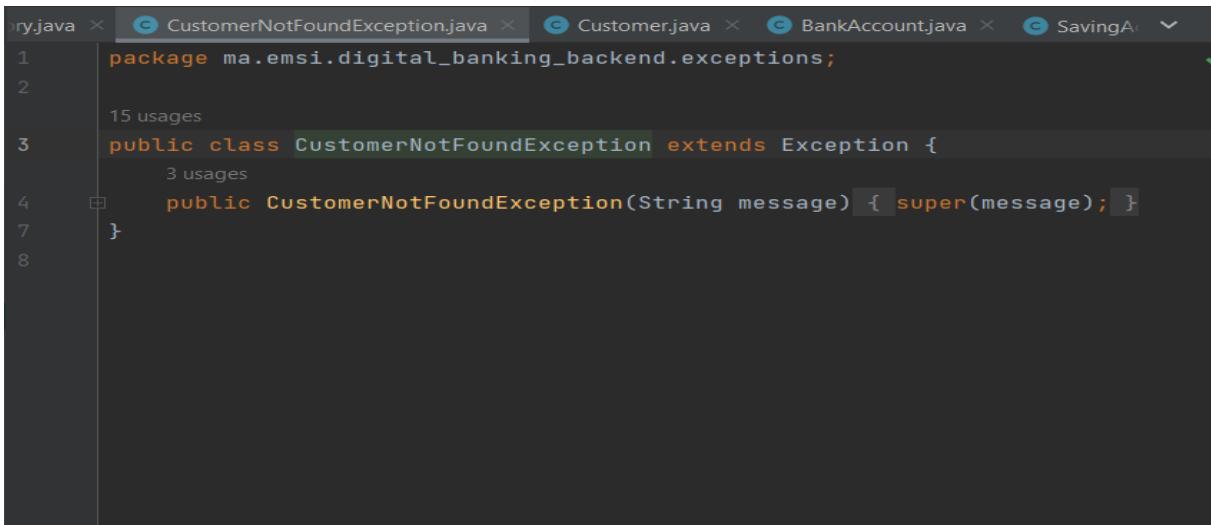
15  * @Slf4j
16  public class CustomerRestController {
17      private BankAccountService bankAccountService;
18
19      @GetMapping("/customers")
20      public List<CustomerDTO> customers(){
21
22          return bankAccountService.listCustomers();
23      }
24
25      @GetMapping("/{id}")
26      public CustomerDTO getCustomer(@PathVariable(name = "id") Long customerId) throws CustomerNotFoundException {
27
28          return bankAccountService.getCustomer(customerId);
29      }
30
31      @PostMapping("/customers")
32      public CustomerDTO saveCustomer(@RequestBody CustomerDTO customerDTO){
33
34          return bankAccountService.saveCustomer(customerDTO);
35      }
36
37      @PutMapping("/customers/{customerId}")
38      public CustomerDTO updateCustomer(@PathVariable Long customerId, @RequestBody CustomerDTO customerDTO){
        customerDTO.setId(customerId);
        return bankAccountService.updateCustomer(customerDTO);
39
40      }
41
42      @DeleteMapping("/customers/{id}")
43      public void deleteCustomer(@PathVariable Long id) { bankAccountService.deleteCustomer(id); }
44  }

```

2. On a créé les exceptions métier :

Ces exceptions sont utilisées pour gérer les erreurs spécifiques à l'application. Dans l'application Digital Banking, nous avons les exceptions suivantes :

- CustomerNotFoundException :

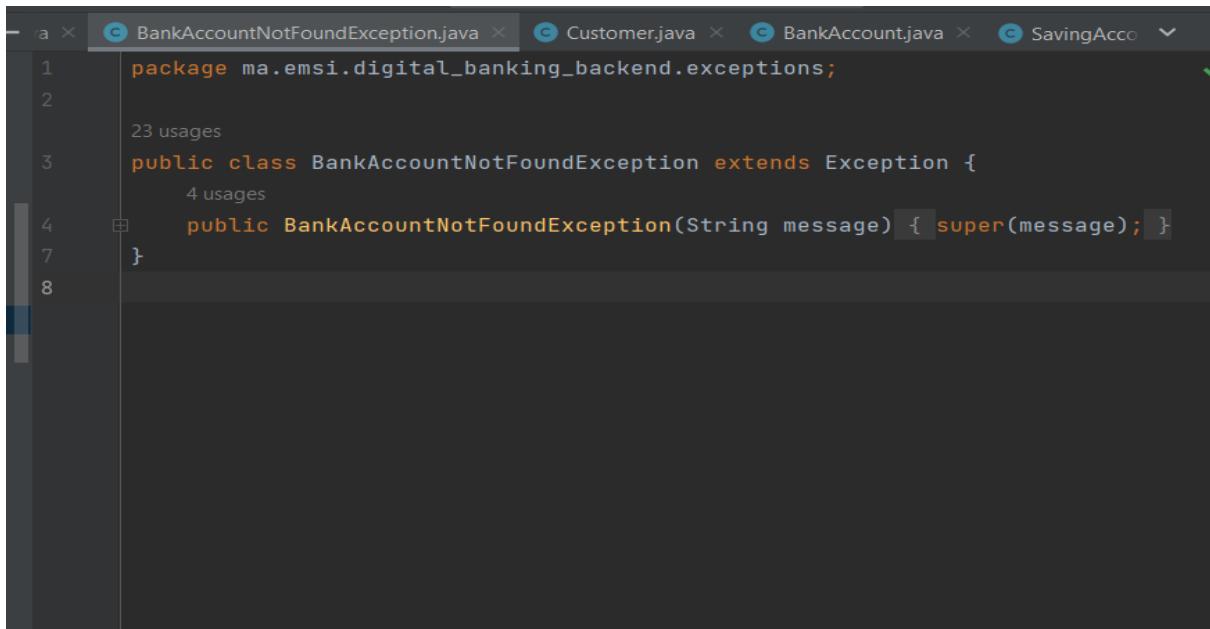


```

1 package ma.emsi.digital_backend.exceptions;
2
3 public class CustomerNotFoundException extends Exception {
4     public CustomerNotFoundException(String message) { super(message); }
5 }

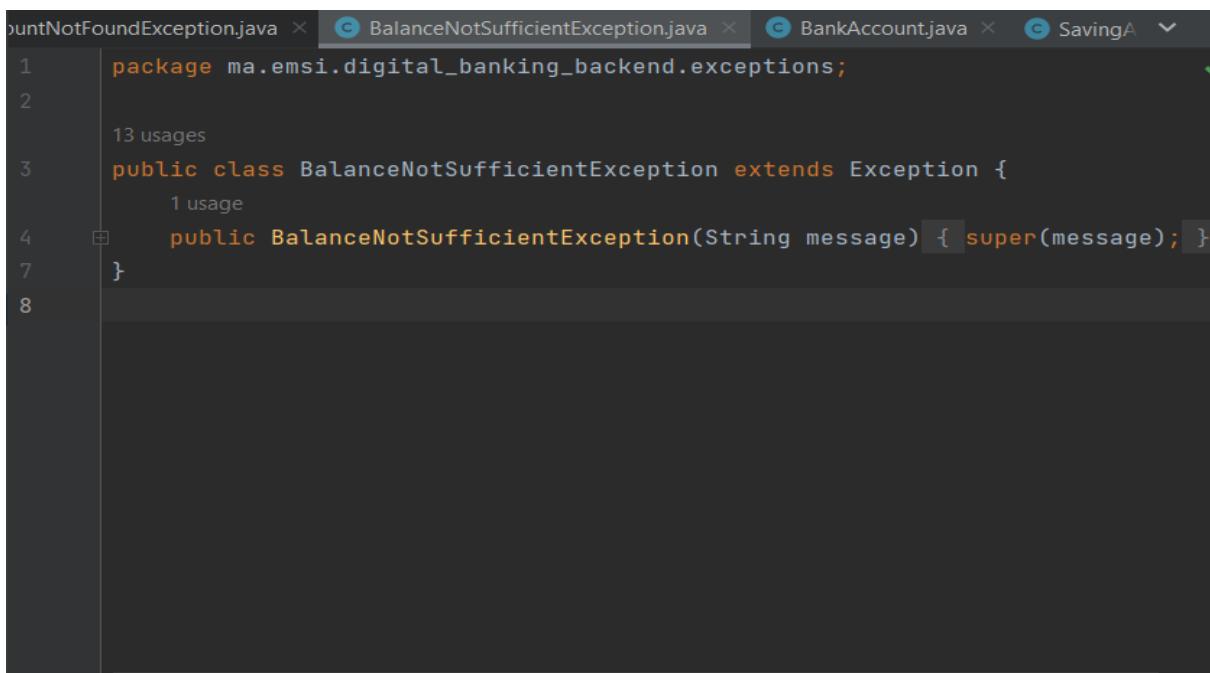
```

➤ BankAccountNotFoundException :



```
1 package ma.emsi.digital_backend.exceptions;
2
3     23 usages
4     public class BankAccountNotFoundException extends Exception {
5         4 usages
6         public BankAccountNotFoundException(String message) { super(message); }
7     }
8
```

➤ BalanceNotSufficientException :



```
1 package ma.emsi.digital_backend.exceptions;
2
3     13 usages
4     public class BalanceNotSufficientException extends Exception {
5         1 usage
6         public BalanceNotSufficientException(String message) { super(message); }
7     }
8
```

3. Les annotations:

- Les entités et les interfaces JPA :

@Entity JPA Entities :

Les entités JPA de l'application sont annotées avec `@Entity` pour les identifier comme entités persistantes. La classe abstraite `BankAccount` utilise l'annotation

`@Inheritance(strategy = InheritanceType.JOINED)` pour spécifier une stratégie d'héritage de type "JOINED".

Cela signifie que les sous-classes `CurrentAccount` et `SavingAccount` auront leurs propres tables distinctes, mais seront liées à la table de la classe `BankAccount` par une clé étrangère.

@Repository :

Les interfaces DAO de l'application utilisent l'annotation `@Repository` pour indiquer à Spring qu'il s'agit de composants de persistance qui gèrent l'accès aux données.

Ces interfaces étendent les interfaces JPA Repository de Spring Data, telles que `JpaRepository` ou `CrudRepository`, et fournissent des méthodes pour effectuer des opérations de base sur les entités.

- Les couches (dtos, mappers, services, web) :

@Service :

Les classes de service de l'application utilisent l'annotation `@Service` pour indiquer qu'elles contiennent la logique métier de l'application.

Ces classes utilisent les interfaces DAO pour accéder aux données et effectuer des opérations sur les entités. Elles peuvent également utiliser des mappers pour convertir les entités en DTOs et vice versa.

@RestController :

Les RestControllers de l'application utilisent l'annotation **@RestController** pour indiquer qu'ils sont des composants qui exposent les services de l'application via des API REST.

Les méthodes de ces classes sont annotées avec des annotations telles que **@GetMapping**, **@PostMapping**, **@PutMapping**, etc., pour spécifier les points de terminaison des API et les opérations HTTP correspondantes.

@ExceptionHandler :

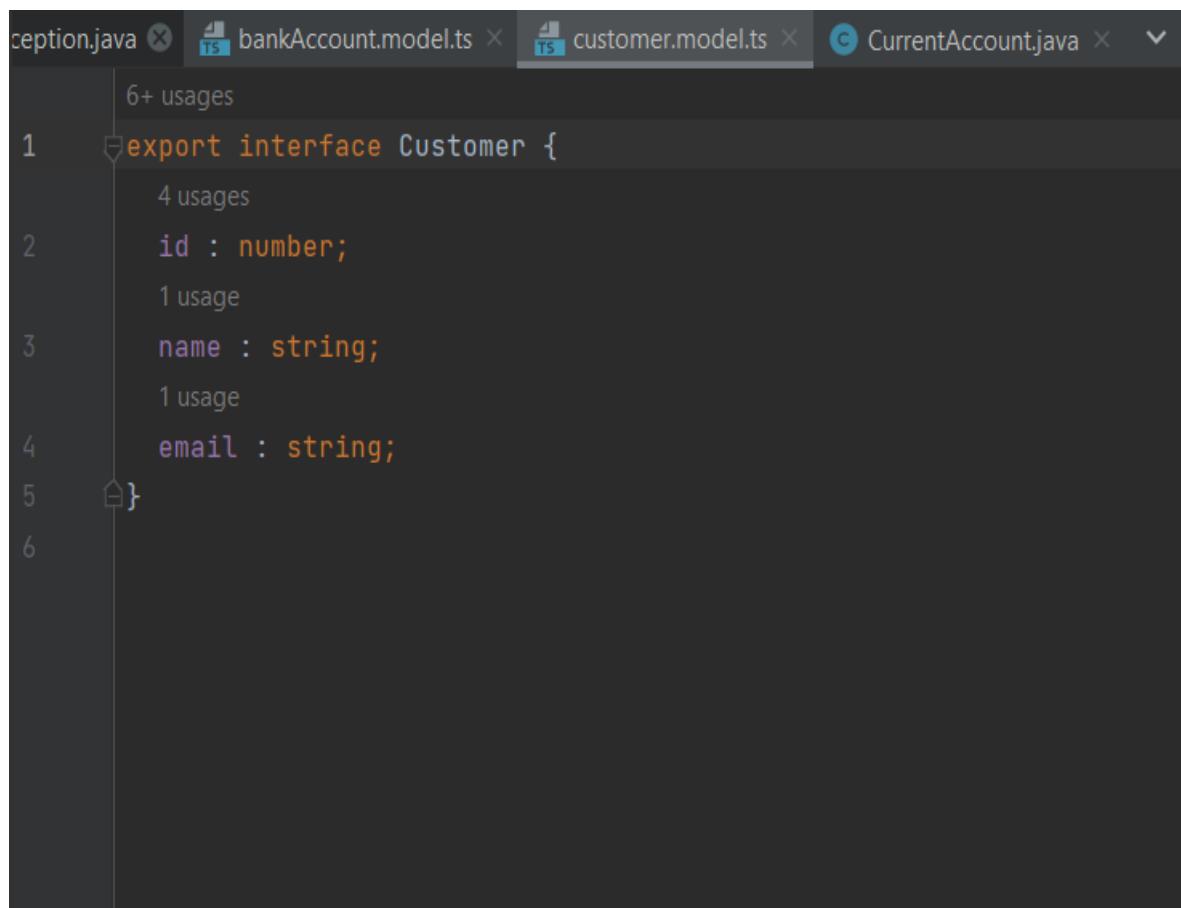
Les RestControllers utilisent également l'annotation **@ExceptionHandler** pour gérer les exceptions spécifiques à l'application. Ces méthodes sont annotées avec des types d'exception spécifiques, tels que **@ExceptionHandler(CustomerNotFoundException.class)**, et renvoient les réponses d'erreur appropriées aux clients.

Partie 3 et 4 : Client Angular

1. On a créé les modèles :

Les modèles font référence aux structures de données utilisées pour représenter et manipuler les données dans l'application. Les modèles sont généralement des classes TypeScript qui définissent la structure des objets de données et les fonctionnalités associées.

➤ Customer :



A screenshot of a code editor showing a TypeScript file named 'customer.model.ts'. The file contains the following code:

```
1 export interface Customer {
2     id : number;
3     name : string;
4     email : string;
5 }
```

The code editor interface shows tabs for 'ception.java', 'bankAccount.model.ts', 'customer.model.ts', and 'CurrentAccount.java'. A tooltip '6+ usages' is visible above the code area. The code is syntax-highlighted with orange for keywords like 'export', 'interface', and 'function', and blue for variable names like 'Customer', 'id', 'name', and 'email'.

➤ BankAccount :

```

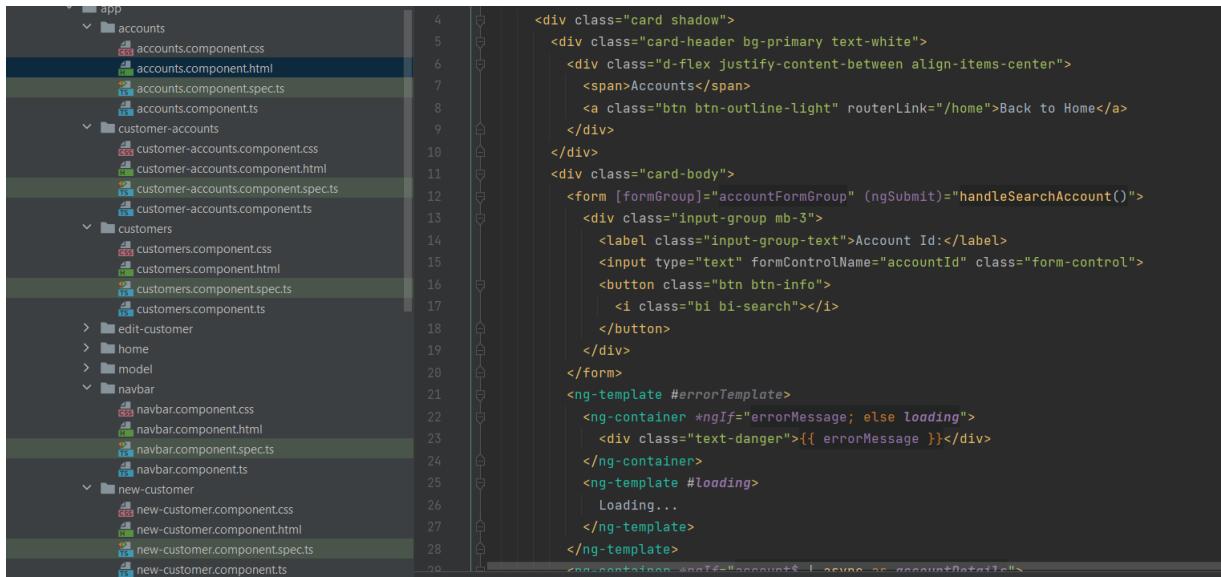
Exception.java ×  bankAccount.model.ts ×  customer.model.ts ×  CurrentAccount.java ×
1      5+ usages
2      ↘export interface AccountDetails {
3          3 usages
4              accountId: string;
5              balance: number;
6              currentPage: number;
7              totalPages: number;
8              pageSize: number;
9              1 usage
10             accountOperationDTOS: AccountOperation[];
11
12             1 usage
13             ↘export interface AccountOperation {
14                 1 usage
15                     id: number;
16                     operationDate: Date;
17                     amount: number;
18                     type: string;
19                     1 usage
20                     description: string;

```

2. On a créé les composants :

Les composants sont les éléments de base pour construire une application. Un composant est une combinaison de code HTML, CSS et TypeScript qui définit la logique, l'apparence et le comportement d'une partie spécifique de l'interface utilisateur de l'application.

- Accounts : Responsable de l'affichage des opérations bancaires d'un compte précis ainsi que d'effectuer des nouvelles opérations (crédit, débit, transfert).
- customer-accounts : Responsable de l'affichage des comptes bancaires d'un client.
- Customers : Responsable de l'affichage de la liste des clients existant, supprimer ou rechercher un client à partir de son nom.
- Navbar : Responsable de l'affichage et de la logique de la barre de navigation.
- new-customer : Responsable de l'affichage du formulaire d'ajout d'un nouveau client.



The screenshot shows a code editor with a dark theme. On the left is a file tree for an Angular project:

```

app
  accounts
    accounts.component.css
    accounts.component.html
    accounts.component.spec.ts
    accounts.component.ts
  customer-accounts
    customer-accounts.component.css
    customer-accounts.component.html
    customer-accounts.component.spec.ts
    customer-accounts.component.ts
  customers
    customers.component.css
    customers.component.html
    customers.component.spec.ts
    customers.component.ts
  edit-customer
  home
  model
  navbar
    navbar.component.css
    navbar.component.html
    navbar.component.spec.ts
    navbar.component.ts
  new-customer
    new-customer.component.css
    new-customer.component.html
    new-customer.component.spec.ts
    new-customer.component.ts
  
```

The right pane displays the content of the selected file, `accounts.component.html`:

```

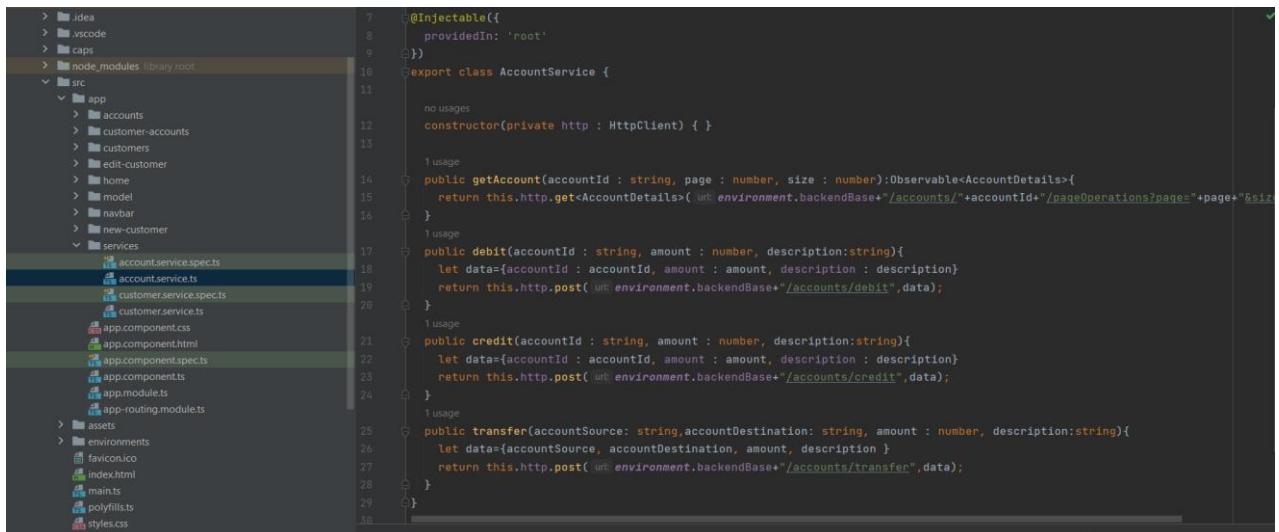

<div class="card-header bg-primary text-white">
    <div class="d-flex justify-content-between align-items-center">
      <span>Accounts</span>
      <a class="btn btn-outline-light" routerLink="/home">Back to Home</a>
    </div>
  </div>
  <div class="card-body">
    <form [formGroup]="accountFormGroup" (ngSubmit)="handleSearchAccount()">
      <div class="input-group mb-3">
        <label class="input-group-text">Account Id:</label>
        <input type="text" formControlName="accountId" class="form-control" />
        <button class="btn btn-info">
          <i class="bi bi-search"></i>
        </button>
      </div>
    </form>
    <ng-template #errorTemplate>
      <ng-container *ngIf="errorMessage; else loading">
        <div class="text-danger">{{ errorMessage }}</div>
      </ng-container>
      <ng-template #loading>
        Loading...
      </ng-template>
    </ng-template>
  </div>


```

3. On a créé les services :

Les services sont des classes qui fournissent des fonctionnalités réutilisables et partagées au sein de l'application. Ils sont utilisés pour séparer la logique métier et les fonctionnalités communes des composants afin de favoriser la modularité, la réutilisabilité et la maintenabilité du code (account, customer) .

- Accounts : Concerne les fonctionnalités métier des comptes bancaires.



The screenshot shows a code editor with a dark theme. On the left is a file tree for the `src` directory, specifically the `services` folder:

```

src
  services
    account.service.specs
    account.service.ts
    customer.service.specs
    customer.service.ts
    app.component.css
    app.component.html
    app.component.spec.ts
    app.components.ts
    app.module.ts
    app-routing.module.ts
  assets
  environments
    favicon.ico
    index.html
    main.ts
    polyfills.ts
    styles.css
  
```

The right pane displays the content of the selected file, `account.service.ts`:

```

@Injectable({
  providedIn: 'root'
})
export class AccountService {
  no usages
  constructor(private http : HttpClient) { }

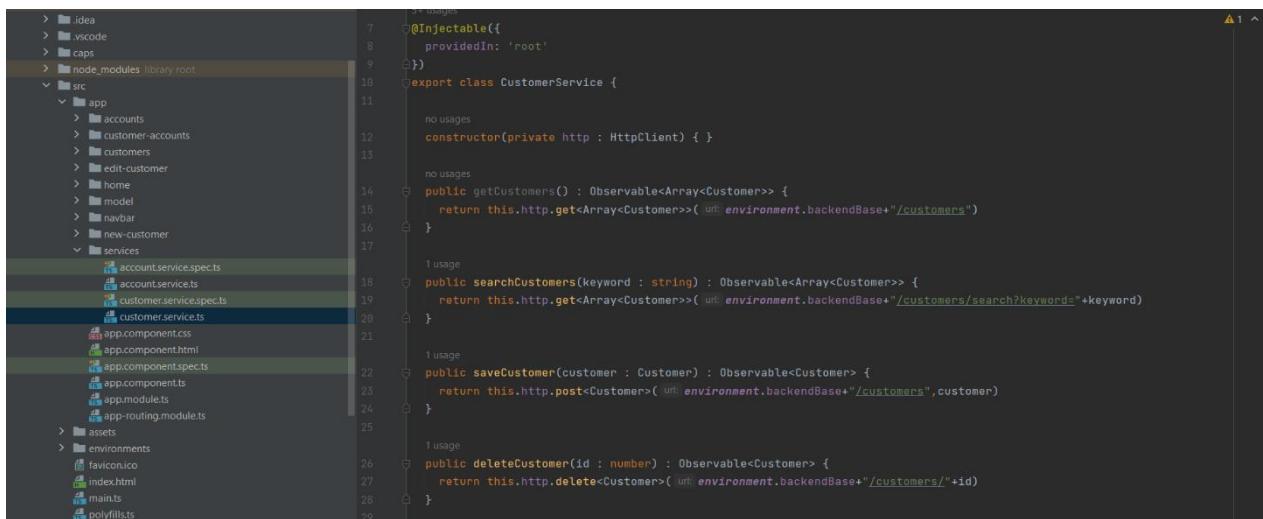
  1 usage
  public getAccount(accountId : string, page : number, size : number):Observable<AccountDetails>{
    return this.http.get<AccountDetails>(`${environment.backendBase}+${accounts}/+${accountId}+${page}+${size}`);
  }

  1 usage
  public debit(accountId : string, amount : number, description:string){
    let data={accountId : accountId, amount : amount, description : description}
    return this.http.post(` ${environment.backendBase}+${accounts}/debit`,data);
  }

  1 usage
  public credit(accountId : string, amount : number, description:string){
    let data={accountId : accountId, amount : amount, description : description}
    return this.http.post(` ${environment.backendBase}+${accounts}/credit`,data);
  }

  1 usage
  public transfer(accountSource: string,accountDestination: string, amount : number, description:string){
    let data={accountSource, accountDestination, amount, description }
    return this.http.post(` ${environment.backendBase}+${accounts}/transfer`,data);
  }
}
  
```

➤ Customer : Concerne les fonctionnalités métier des clients.



```

<!-- usage -->
@ Injectable({
  providedIn: 'root'
})
export class CustomerService {

  constructor(private http : HttpClient) { }

  no usages
  public getCustomers() : Observable<Array<Customer>> {
    return this.http.get<Array<Customer>>(` ${environment.backendBase}/customers`)
  }

  1 usage
  public searchCustomers(keyword : string) : Observable<Array<Customer>> {
    return this.http.get<Array<Customer>>(` ${environment.backendBase}/customers/search?keyword=${keyword}`)
  }

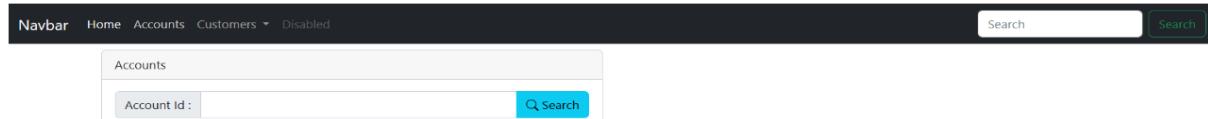
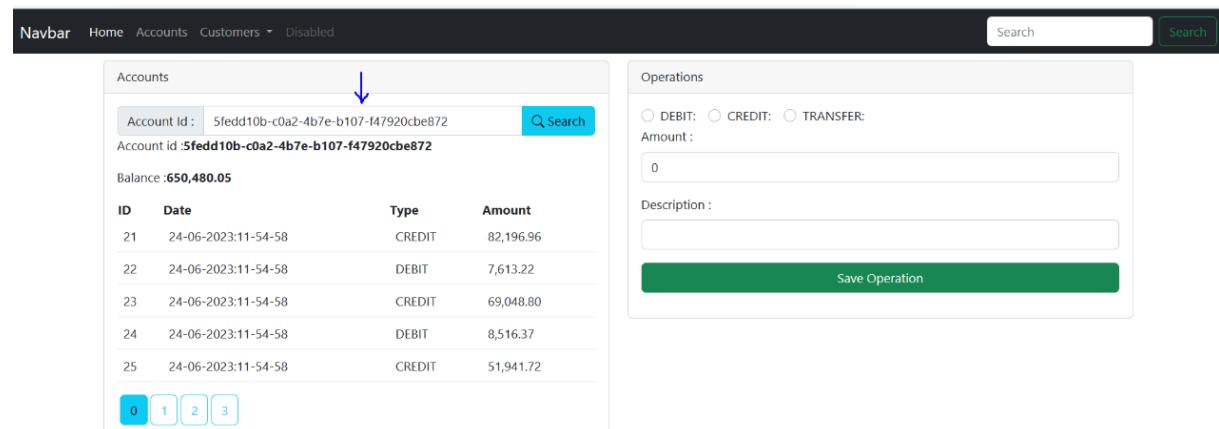
  1 usage
  public saveCustomer(customer : Customer) : Observable<Customer> {
    return this.http.post<Customer>(` ${environment.backendBase}/customers`,customer)
  }

  1 usage
  public deleteCustomer(id : number) : Observable<Customer> {
    return this.http.delete<Customer>(` ${environment.backendBase}/customers/${id}`)
  }
}

```

Les interfaces du projet :

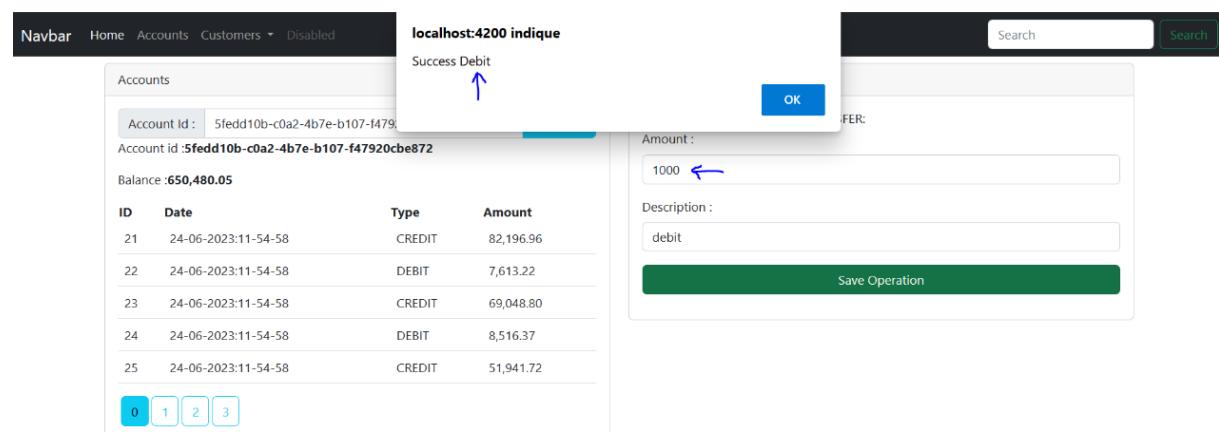
La page accounts nous donne la possibilité de chercher un client grâce à son ID et de visualiser toutes les opérations faites par ce client :

ID	Date	Type	Amount
21	24-06-2023:11:54:58	CREDIT	82,196.96
22	24-06-2023:11:54:58	DEBIT	7,613.22
23	24-06-2023:11:54:58	CREDIT	69,048.80
24	24-06-2023:11:54:58	DEBIT	8,516.37
25	24-06-2023:11:54:58	CREDIT	51,941.72

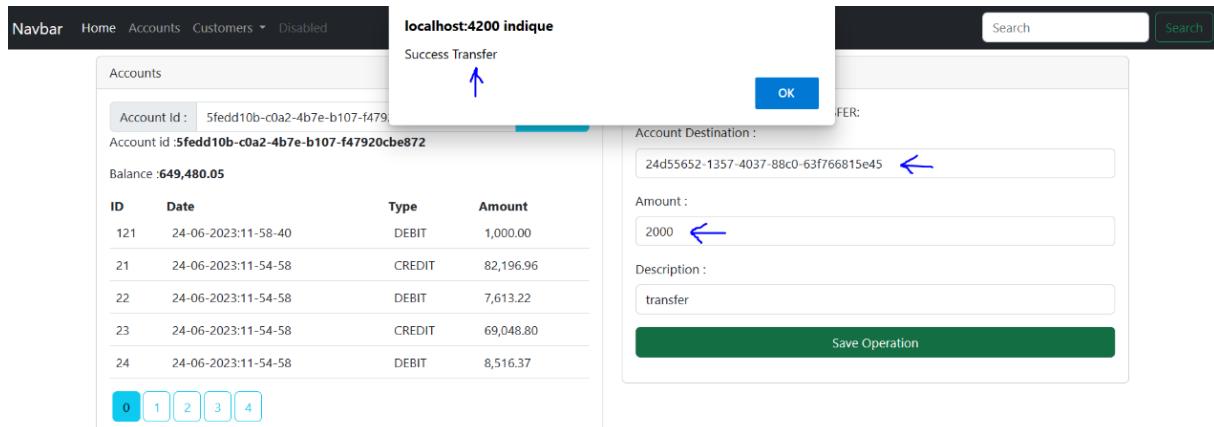
La possibilité d'effectuer une transfer,debit,credit est aussi possible

DEBIT :



ID	Date	Type	Amount
21	24-06-2023:11:54:58	CREDIT	82,196.96
22	24-06-2023:11:54:58	DEBIT	7,613.22
23	24-06-2023:11:54:58	CREDIT	69,048.80
24	24-06-2023:11:54:58	DEBIT	8,516.37
25	24-06-2023:11:54:58	CREDIT	51,941.72

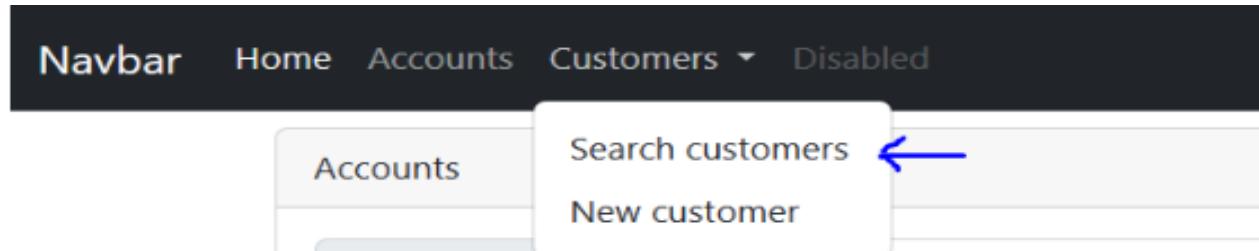
TRANSFER :



localhost:4200 indique Success Transfer

ID	Date	Type	Amount
121	24-06-2023:11:58:40	DEBIT	1,000.00
21	24-06-2023:11:54:58	CREDIT	82,196.96
22	24-06-2023:11:54:58	DEBIT	7,613.22
23	24-06-2023:11:54:58	CREDIT	69,048.80
24	24-06-2023:11:54:58	DEBIT	8,516.37

La page customers nous donne la possibilité de visualiser tout les clients :



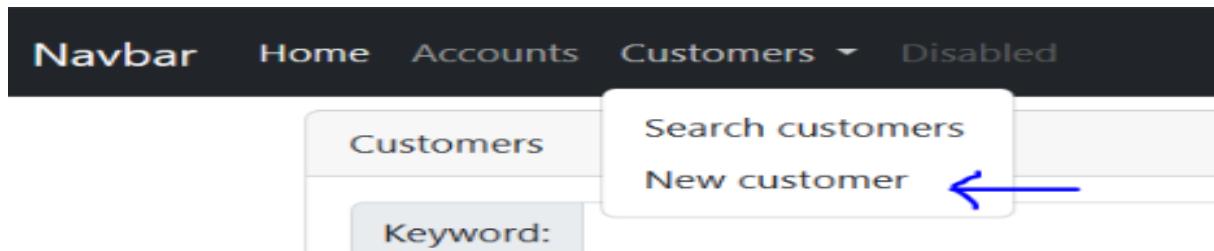
Navbar Home Accounts Customers ▾ Disabled

Accounts

Search customers ←

New customer

La page customers nous donne la possibilité de visualiser tout les clients :



Navbar Home Accounts Customers ▾ Disabled

Customers

Keyword:

Search customers

New customer ←

Conclusion :

En conclusion, l'architecture backend de l'application Digital Banking master, basée sur Spring Boot, suit une approche en couches qui favorise une meilleure séparation des responsabilités

Les couches DAO, service et RestController collaborent étroitement pour assurer la gestion des comptes bancaires et la manipulation des données.

Les entités JPA représentent les objets métier et sont mappées aux tables de la base de données, tandis que les DTOs sont utilisés pour transférer les données entre les différentes couches de l'application.

L'utilisation de Spring Boot simplifie le développement, la configuration et la gestion de l'application. De plus, les annotations telles que @Entity, @Repository, @Service et @RestController permettent de définir clairement le rôle des différents composants de l'application.

Grâce à cette architecture, il est possible d'étendre l'application Digital Banking avec de nouvelles fonctionnalités tout en maintenant un code clair et modulaire.

L'architecture frontend développée avec le framework Angular.

L'interface utilisateur offre aux utilisateurs la possibilité de consulter les clients enregistrés, d'ajouter de nouveaux clients, d'accéder aux détails des comptes bancaires, de visualiser les opérations effectuées et d'effectuer de nouvelles opérations financières telles que le débit, le crédit et le transfert.

L'application offre une expérience conviviale pour la gestion efficace et sécurisée des opérations bancaires en ligne.