

## DÉPARTEMENT MATHÉMATIQUES ET INFORMATIQUE

### Filière :

« Ingénierie informatique – Big Data et Cloud Computing »

### II-BDCC

### Rapport de projet fin de module

## Digital Banking

Réalisé par :

- HOUD Fatima-Ezzahra

Encadré par :

- Pr. YOUSFI Mohamed

**Année Universitaire : 2022-2023**



# Plan

I.	Introduction.....	4
1.	Contexte général .....	4
2.	Objectifs du projet.....	4
3.	Structure de projet.....	4
II.	Conception et architecture .....	6
1.	Architecture de projet .....	6
1.1.	Architecture MVC .....	6
1.2.	Base de données MySQL .....	8
2.	Diagramme de classe .....	9
III.	Réalisation et implementation.....	11
1.	Outils utilisés .....	11
2.	Code source.....	11
1.1.	Coté backend .....	11
1.2.	Coté frontend .....	26
3.	Interfaces.....	27
IV.	Conclusion .....	30

# I. Introduction

## 1. Contexte général

La gestion des comptes bancaires est une préoccupation centrale pour de nombreux individus et entreprises à travers le monde. Les avancées technologiques ont permis le développement d'applications innovantes qui offrent des solutions pratiques et efficaces pour la gestion des finances personnelles. Dans ce contexte, notre projet vise à créer une application de gestion de comptes bancaires répondant aux besoins des clients et offrant une expérience utilisateur optimale.

## 2. Objectifs du projet

L'objectif principal de ce projet est de concevoir et développer une application complète et fonctionnelle permettant aux clients de gérer facilement leurs comptes bancaires. L'application offrira des fonctionnalités telles que la création de comptes, la consultation des soldes, l'exécution d'opérations de débit et de crédit, ainsi que la visualisation des historiques de transactions. Nous souhaitons fournir aux utilisateurs une plateforme intuitive et conviviale pour effectuer leurs opérations financières en toute simplicité.

## 3. Structure de projet

Le projet est divisé en différentes parties qui couvrent les différents aspects du développement de l'application de gestion de comptes bancaires. Chaque partie est essentielle pour assurer le bon fonctionnement et la performance de l'application. Voici un aperçu des principales parties du projet :

- **Couche DAO** (Data Access Object) : Cette partie concerne la manipulation des données dans la base de données. Nous mettrons en place les entités JPA (Java Persistence API) pour représenter les différentes entités du système bancaire, ainsi que les interfaces JPA Repository pour effectuer les opérations de lecture et d'écriture des données.

- **Couche services**, DTO et mappers : Cette partie implémente la logique métier de l'application. Nous développerons des services qui gèrent les opérations bancaires et utilisent des objets DTO pour transférer les données entre les différentes couches de l'application. Les mappers seront utilisés pour convertir les entités en DTO et vice versa.
- **Couche Web** (RestController) : Dans cette partie, nous créerons les RestControllers qui serviront de points d'entrée pour les requêtes HTTP. Ils permettront aux clients de communiquer avec l'application et d'effectuer des opérations sur leurs comptes bancaires via des API REST.
- **Frontend Angular** : Le développement du frontend de l'application sera réalisé en utilisant Angular, un framework JavaScript populaire. Nous concevrons une interface utilisateur attrayante et ergonomique pour permettre aux clients d'accéder facilement aux fonctionnalités de gestion des comptes bancaires et d'interagir avec l'application de manière conviviale.
- **Sécurité** avec Spring Security et JWT : La sécurité est un élément crucial dans toute application bancaire. Nous mettrons en place un système de sécurité robuste en utilisant Spring Security et JWT (JSON Web Tokens) pour authentifier et autoriser les clients, protéger les données sensibles et garantir l'intégrité des opérations effectuées.

## II. Conception et architecture

### 1. Architecture de projet

#### 1.1. Architecture MVC

L'architecture de notre application de gestion de comptes bancaires repose sur le modèle MVC (Modèle-Vue-Contrôleur), combinant les frameworks Spring pour le backend et Angular pour le frontend. Nous avons choisi cette architecture car elle favorise la séparation des responsabilités, la modularité et la facilité de maintenance du code.

##### Modèle (Backend avec Spring) :

Utilisation de Spring Boot : Nous avons adopté Spring Boot pour faciliter le développement de l'application backend. Spring Boot offre une configuration simplifiée, une gestion transparente des dépendances et des fonctionnalités avancées pour le développement d'applications Java.

Modélisation des entités : Nous avons défini plusieurs entités clés telles que Customer (client), BankAccount (compte bancaire), SavingAccount (compte épargne), CurrentAccount (compte courant) et AccountOperation (opération de compte) en utilisant des classes Java avec des annotations JPA.

Repositories basés sur Spring Data JPA : Nous avons créé des interfaces JPA Repository basées sur Spring Data pour effectuer les opérations de lecture et d'écriture des données dans la base de données.

##### Vue (Frontend avec Angular) :

Utilisation d'Angular : Nous avons choisi Angular pour développer le frontend de l'application. Angular est un framework JavaScript basé sur TypeScript, offrant des fonctionnalités avancées pour la création d'interfaces utilisateur réactives et dynamiques.

Création de composants Angular : Nous avons conçu des composants réutilisables tels que CustomerComponent, BankAccountComponent, etc.,

pour représenter les différentes parties de l'interface utilisateur liées aux clients et aux comptes bancaires.

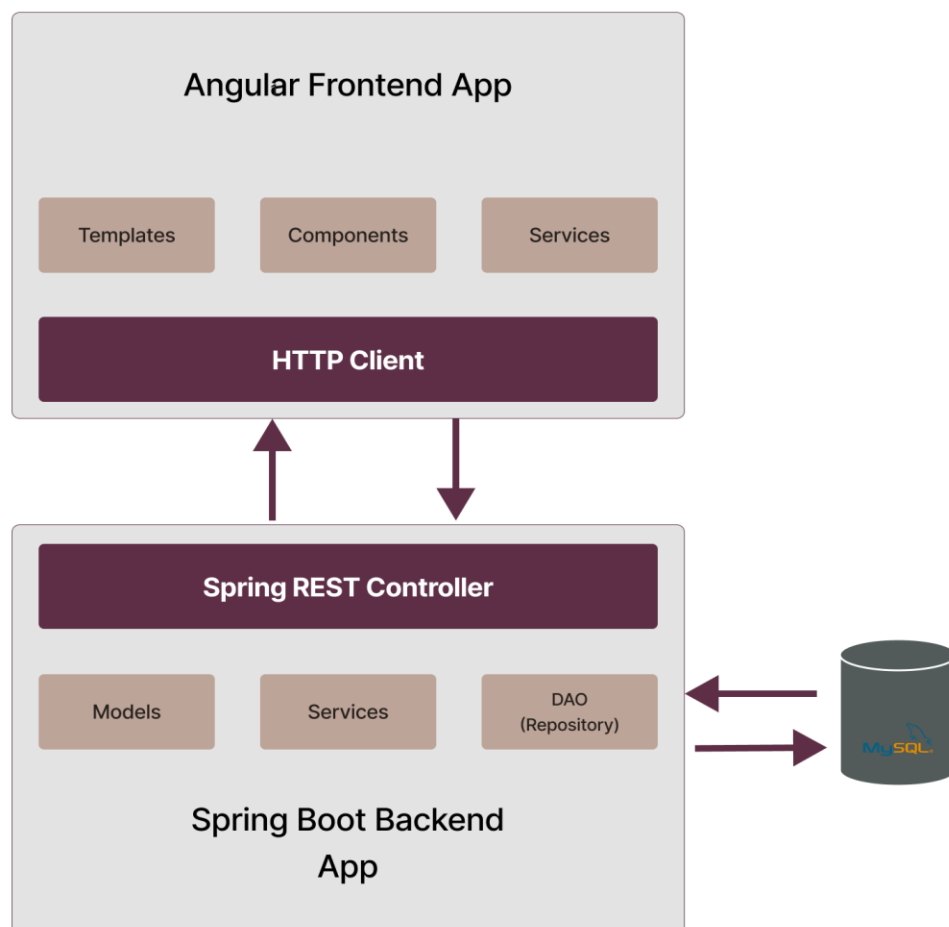
Gestion des interactions utilisateur : Angular nous permet de gérer les événements utilisateur, les formulaires et les appels HTTP pour interagir avec le backend et récupérer les données nécessaires.

### Contrôleur (Backend avec Spring) :

Utilisation des RestControllers : Nous avons mis en place des RestControllers basés sur Spring pour gérer les requêtes HTTP et les réponses. Ces RestControllers exposent des API REST qui servent de points d'entrée pour les opérations de gestion des comptes bancaires.

Implémentation des routes et des méthodes HTTP : Les RestControllers sont configurés pour gérer les différentes routes de l'API REST, telles que /customers, /bank-accounts, etc. Ils utilisent les méthodes HTTP appropriées (GET, POST, PUT, DELETE) pour effectuer les opérations correspondantes sur les données.

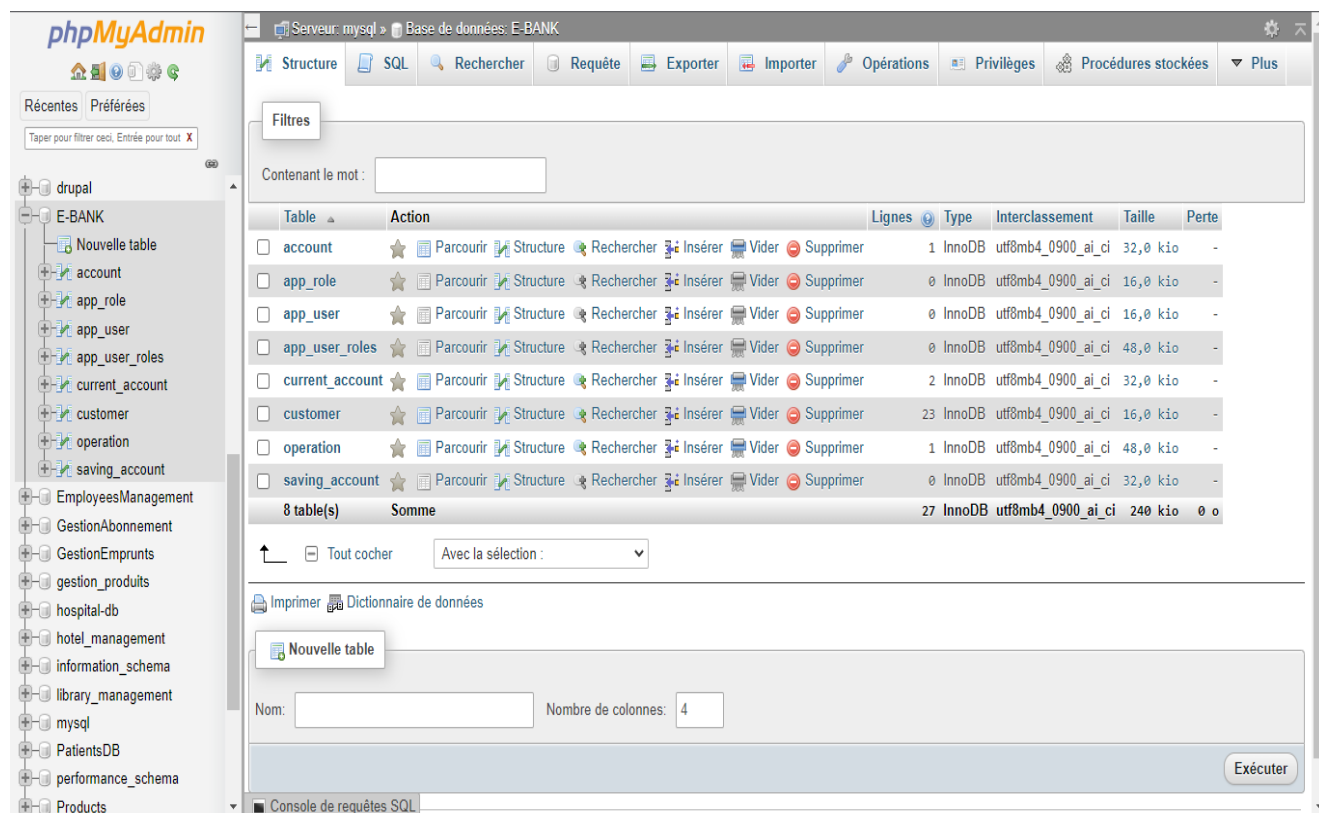
Section 1



## 1.2. Base de données MySQL

Pour la persistance des données, nous avons choisi d'utiliser une base de données relationnelle MySQL. MySQL est largement utilisé et offre une grande stabilité, une bonne performance et une prise en charge complète du langage SQL. Nous avons créé des tables correspondantes aux entités du modèle dans MySQL, permettant de stocker les informations des clients, des comptes bancaires et des opérations associées.

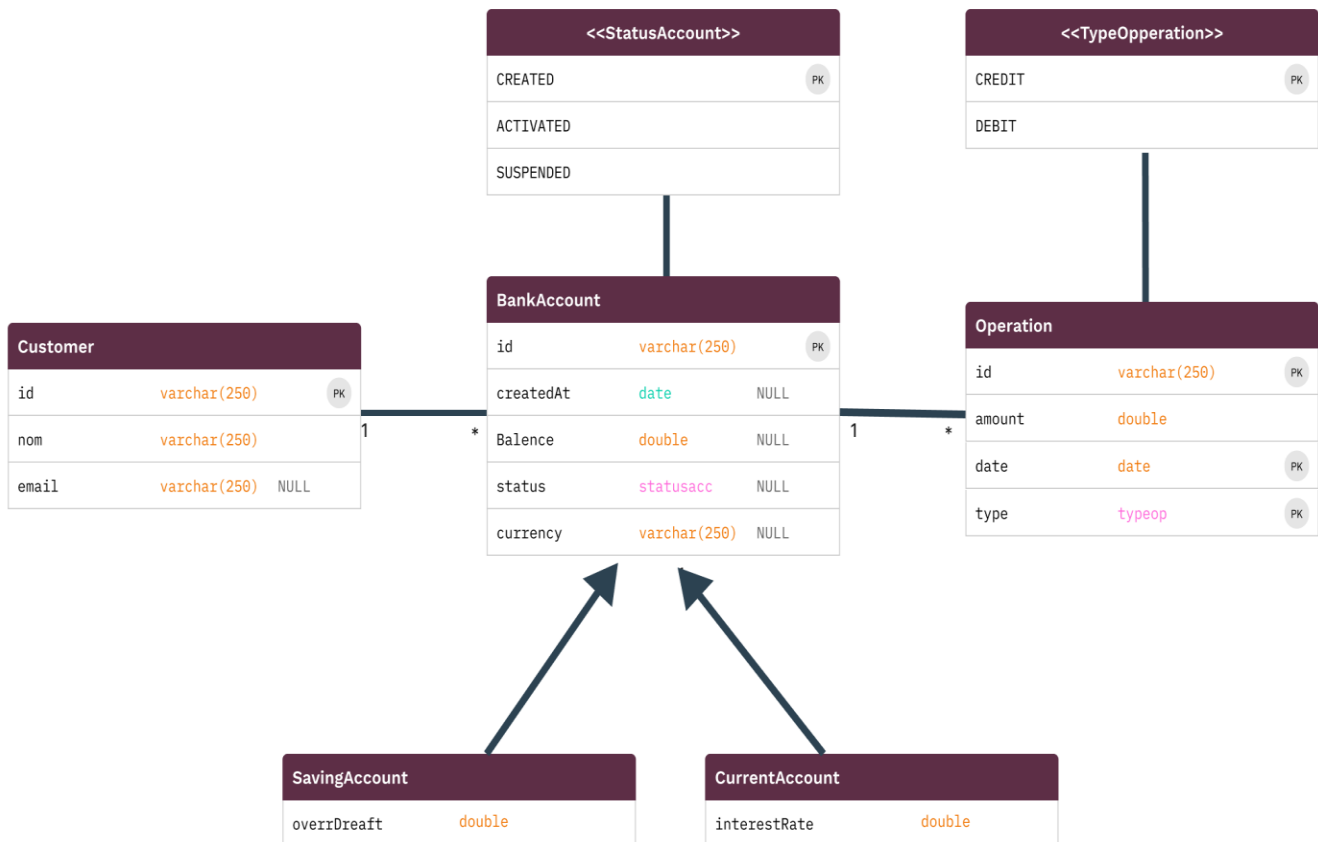
En combinant l'architecture MVC avec Spring et Angular, ainsi que l'utilisation de la base de données MySQL, nous avons réussi à mettre en place une structure solide et efficace pour notre application de gestion de comptes bancaires. Cette architecture nous permet de séparer les préoccupations, de faciliter la maintenance et de fournir une expérience utilisateur fluide et réactive.





## 2. Diagramme de classe

La partie du diagramme de classe de notre projet de gestion de comptes bancaires joue un rôle essentiel dans la représentation visuelle des entités et de leurs relations. Ce diagramme permet de visualiser la structure des classes de notre application et les interactions entre elles. Voici un exemple de paragraphe décrivant cette partie :



Le diagramme de classe de notre application de gestion de comptes bancaires représente les entités clés et leurs relations. Nous avons identifié les classes **Customer** (client), **BankAccount** (compte bancaire), **SavingAccount** (compte épargne), **CurrentAccount** (compte courant) et **AccountOperation** (opération de compte). La classe **Customer** représente les informations relatives aux clients, telles que leur nom, leur adresse et

leur numéro de téléphone. Les classes BankAccount, SavingAccount et CurrentAccount héritent de la classe Account et représentent respectivement les comptes bancaires, les comptes épargnes et les comptes courants. La classe AccountOperation enregistre les opérations effectuées sur les comptes, telles que les dépôts, les retraits et les virements. Les relations entre les classes sont définies à l'aide d'associations, telles que l'association "possède" entre Customer et BankAccount pour indiquer que chaque client peut posséder plusieurs comptes bancaires. Le diagramme de classe offre une représentation visuelle claire de la structure de notre application et facilite la compréhension des entités et de leurs interactions.

### III. Réalisation et implementation

La phase de réalisation et d'implémentation de notre projet de gestion de comptes bancaires a été une étape cruciale dans le développement de l'application. Cette phase nous a permis de concrétiser les concepts et les spécifications définis lors de la phase de conception, en mettant en place les différentes couches du système et en implémentant les fonctionnalités clés. Dans cette partie, nous décrirons en détail les technologies, les outils et les étapes que nous avons utilisés pour réaliser avec succès le projet.

#### 1. Outils utilisés

Pour la réalisation de notre application de gestion de comptes bancaires, nous avons utilisé un ensemble de technologies et d'outils adaptés aux différents aspects du projet. Nous avons opté pour Java comme langage de programmation principal, accompagné du framework Spring Boot pour la partie backend et d'Angular pour le frontend. Pour la gestion des données, nous avons choisi MySQL comme système de gestion de base de données relationnelle. De plus, nous avons utilisé des outils tels que Spring Data JPA, Hibernate et Visual Studio Code pour faciliter le développement et la gestion du code.

#### 2. Code source

##### 1.1. Coté backend

- Entité «AccountOperation »

```
- package ma.enset.digitalbanking.entities;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import ma.enset.digitalbanking.dtos.*;
import ma.enset.digitalbanking.entities.*;
import ma.enset.digitalbanking.enums.OperationType;
import ma.enset.digitalbanking.exceptions.*;
import ma.enset.digitalbanking.mappers.*;
import ma.enset.digitalbanking.repositories.*;
```

```

import java.util.Date;
@Entity
@Data @NoArgsConstructor @AllArgsConstructor
public class AccountOperation {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Date operationDate;
    private double amount;
    @Enumerated(EnumType.STRING)
    private OperationType type;
    @ManyToOne
    private BankAccount bankAccount;
    private String description;
}

```

## - Entité «BankAccount »

```

- package ma.enset.digitalbanking.entities;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import ma.enset.digitalbanking.dtos.*;
import ma.enset.digitalbanking.entities.*;
import ma.enset.digitalbanking.enums.AccountStatus;
import ma.enset.digitalbanking.enums.OperationType;
import ma.enset.digitalbanking.exceptions.*;
import ma.enset.digitalbanking.mappers.*;
import ma.enset.digitalbanking.repositories.*;
import java.util.Date;

import java.util.Date;
import java.util.List;
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@DiscriminatorColumn(name = "TYPE",length = 4)
@Data @NoArgsConstructor @AllArgsConstructor
public abstract class BankAccount {
    @Id
    private String id;
    private double balance;
    private Date createdAt;
    @Enumerated(EnumType.STRING)
    private AccountStatus status;
    @ManyToOne
    private Customer customer;

    @OneToMany(mappedBy = "bankAccount", fetch = FetchType.LAZY)
    private List<AccountOperation> accountOperations;
}

```

## - Entité «CurrentAccount »

```

- package ma.enset.digitalbanking.entities;

```

```

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import jakarta.persistence.*;

@Entity
@DiscriminatorValue("CA")
@Data @NoArgsConstructor @AllArgsConstructor
public class CurrentAccount extends BankAccount {
    private double overDraft;
}

```

#### - Entité «Customer »

```

- package ma.enset.digitalbanking.entities;
import com.fasterxml.jackson.annotation.JsonProperty;
import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.List;

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

```

#### - Entité «SavingAccount »

```

- package ma.enset.digitalbanking.entities;

import jakarta.persistence.DiscriminatorValue;
import jakarta.persistence.Entity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@DiscriminatorValue("SA")
@Data @NoArgsConstructor @AllArgsConstructor
public class SavingAccount extends BankAccount {
    private double interestRate;
}

```

- Dto «AccountHistory »

```
- package ma.enset.digitalbanking.dtos;

import lombok.Data;

import java.util.List;
@Data
public class AccountHistoryDTO {
    private String accountId;
    private double balance;
    private int currentPage;
    private int totalPages;
    private int pageSize;
    private List<AccountOperationDTO> accountOperationDTOS;
}
```

- Dto «AccountOperation »

```
- package ma.enset.digitalbanking.dtos;

import lombok.Data;
import ma.enset.digitalbanking.enums.*;
import java.util.Date;

@Data
public class AccountOperationDTO {
    private Long id;
    private Date operationDate;
    private double amount;
    private OperationType type;
    private String description;
}
```

- Dto «BankAccount »

```
- package ma.enset.digitalbanking.dtos;

import lombok.Data;

@Data
public class BankAccountDTO {
    private String type;
}
```

- Dto «Credit »

```
- package ma.enset.digitalbanking.dtos;

import lombok.Data;
```

```

@Data
public class CreditDTO {
    private String accountId;
    private double amount;
    private String description;
}

```

#### - Dto «CurrentBankAccount »

```

- package ma.enset.digitalbanking.dtos;

import lombok.Data;
import ma.enset.digitalbanking.enums.AccountStatus;
import java.util.Date;

@Data
public class CurrentBankAccountDTO extends BankAccountDTO {
    private String id;
    private double balance;
    private Date createdAt;
    private AccountStatus status;
    private CustomerDTO customerDTO;
    private double overDraft;
}

```

#### - Dto «Customer »

```

- package ma.enset.digitalbanking.dtos;

import lombok.Data;

@Data
public class CustomerDTO {
    private Long id;
    private String name;
    private String email;
}

```

#### - Dto «Debit »

```

- package ma.enset.digitalbanking.dtos;

import lombok.Data;

@Data
public class DebitDTO {
    private String accountId;
    private double amount;
    private String description;
}

```

#### - Dto «SavingBankAccount »

```

- package ma.enset.digitalbanking.dtos;

import lombok.Data;
import ma.enset.digitalbanking.enums.*;
import java.util.Date;

```

```

@Data
public class SavingBankAccountDTO extends BankAccountDTO {
    private String id;
    private double balance;
    private Date createdAt;
    private AccountStatus status;
    private CustomerDTO customerDTO;
    private double interestRate;
}

```

- Enum «AccountStatus »

```

- package ma.enset.digitalbanking.enums;

public enum AccountStatus {
    CREATED, ACTIVATED, SUSPENDED
}

```

- Enum «OperationType »

```

- package ma.enset.digitalbanking.enums;

public enum OperationType {
    DEBIT, CREDIT
}

```

- Mapper «BankAccountMapper »

```

- package ma.enset.digitalbanking.mappers;

import ma.enset.digitalbanking.dtos.*;
import ma.enset.digitalbanking.entities.*;
import org.springframework.beans.BeanUtils;
import org.springframework.stereotype.Service;
@Service
public class BankAccountMapperImpl {
    public CustomerDTO fromCustomer(Customer customer){
        CustomerDTO customerDTO=new CustomerDTO();
        BeanUtils.copyProperties(customer,customerDTO);
        return customerDTO;
    }
    public Customer fromCustomerDTO(CustomerDTO customerDTO){
        Customer customer=new Customer();
        BeanUtils.copyProperties(customerDTO,customer);
        return customer;
    }

    public SavingBankAccountDTO fromSavingBankAccount(SavingAccount
savingAccount){
        SavingBankAccountDTO savingBankAccountDTO=new
SavingBankAccountDTO();
        BeanUtils.copyProperties(savingAccount,savingBankAccountDTO);

        savingBankAccountDTO.setCustomerDTO(fromCustomer(savingAccount.getCustom
er()));
}

```



```

savingBankAccountDTO.setType(savingAccount.getClass().getSimpleName());
return savingBankAccountDTO;
}

public SavingAccount fromSavingBankAccountDTO(SavingBankAccountDTO
savingBankAccountDTO) {
    SavingAccount savingAccount=new SavingAccount();
    BeanUtils.copyProperties(savingBankAccountDTO,savingAccount);

    savingAccount.setCustomer(fromCustomerDTO(savingBankAccountDTO.getCustom
erDTO()));
    return savingAccount;
}

public CurrentBankAccountDTO fromCurrentBankAccount(CurrentAccount
currentAccount) {
    CurrentBankAccountDTO currentBankAccountDTO=new
CurrentBankAccountDTO();
    BeanUtils.copyProperties(currentAccount,currentBankAccountDTO);

    currentBankAccountDTO.setCustomerDTO(fromCustomer(currentAccount.getCust
omer()));

    currentBankAccountDTO.setType(currentAccount.getClass().getSimpleName());
    ;
    return currentBankAccountDTO;
}

public CurrentAccount
fromCurrentBankAccountDTO(CurrentBankAccountDTO currentBankAccountDTO) {
    CurrentAccount currentAccount=new CurrentAccount();
    BeanUtils.copyProperties(currentBankAccountDTO,currentAccount);

    currentAccount.setCustomer(fromCustomerDTO(currentBankAccountDTO.getCust
omerDTO()));
    return currentAccount;
}

public AccountOperationDTO fromAccountOperation(AccountOperation
accountOperation) {
    AccountOperationDTO accountOperationDTO=new
AccountOperationDTO();
    BeanUtils.copyProperties(accountOperation,accountOperationDTO);
    return accountOperationDTO;
}
}

```

## - Repository «AccountOperation »

```

- package ma.enset.digitalbanking.repositories;

import ma.enset.digitalbanking.entities.AccountOperation;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;

```

```
import java.util.List;

public interface AccountOperationRepository extends
JpaRepository<AccountOperation, Long> {
    List<AccountOperation> findByBankAccountId(String accountId);
    Page<AccountOperation>
findByBankAccountIdOrderByOperationDateDesc(String accountId, Pageable
pageable);
}
```

#### - Repository «BankAccount »

```
- package ma.enset.digitalbanking.repositories;

import ma.enset.digitalbanking.entities.BankAccount;
import org.springframework.data.jpa.repository.JpaRepository;

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

#### - Repository «Customer »

```
- package ma.enset.digitalbanking.repositories;

import ma.enset.digitalbanking.entities.Customer;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import java.util.List;

public interface CustomerRepository extends JpaRepository<Customer, Long>
{
    @Query("select c from Customer c where c.name like :kw")
    List<Customer> searchCustomer(@Param("kw") String keyword);
}
```

#### - Service «BankAccount »

```
- package ma.enset.digitalbanking.services;

import ma.enset.digitalbanking.dtos.*;
import ma.enset.digitalbanking.exceptions.*;

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;

    CustomerDTO updateCustomer(CustomerDTO customerDTO);

    void deleteCustomer(Long customerId);

    List<AccountOperationDTO> accountHistory(String accountId);

    AccountHistoryDTO getAccountHistory(String accountId, int page, int
    size) throws BankAccountNotFoundException;

    List<CustomerDTO> searchCustomers(String keyword);
}

```

## - Service «BankAccountImpl »

```

- package ma.enset.digitalbanking.services;

import lombok.AllArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import ma.enset.digitalbanking.dtos.*;
import ma.enset.digitalbanking.entities.*;
import ma.enset.digitalbanking.enums.OperationType;
import ma.enset.digitalbanking.exceptions.*;
import ma.enset.digitalbanking.mappers.*;
import ma.enset.digitalbanking.repositories.*;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.Date;
import java.util.List;
import java.util.UUID;
import java.util.stream.Collectors;

@Service
@Transactional
@AllArgsConstructor
@Slf4j
public class BankAccountServiceImpl implements BankAccountService {
    private CustomerRepository customerRepository;
    private BankAccountRepository bankAccountRepository;
    private AccountOperationRepository accountOperationRepository;
}

```

```

private BankAccountMapperImpl dtoMapper;

@Override
public CustomerDTO saveCustomer(CustomerDTO customerDTO) {
    log.info("Saving new Customer");
    Customer customer=dtoMapper.fromCustomerDTO(customerDTO);
    Customer savedCustomer = customerRepository.save(customer);
    return dtoMapper.fromCustomer(savedCustomer);
}

@Override
public CurrentBankAccountDTO saveCurrentBankAccount(double
initialBalance, double overDraft, Long customerId) throws
CustomerNotFoundException {
    Customer
customer=customerRepository.findById(customerId).orElse(null);
    if(customer==null)
        throw new CustomerNotFoundException("Customer not found");
    CurrentAccount currentAccount=new CurrentAccount();
    currentAccount.setId(UUID.randomUUID().toString());
    currentAccount.setCreatedAt(new Date());
    currentAccount.setBalance(initialBalance);
    currentAccount.setOverDraft(overDraft);
    currentAccount.setCustomer(customer);
    CurrentAccount savedBankAccount =
bankAccountRepository.save(currentAccount);
    return dtoMapper.fromCurrentBankAccount(savedBankAccount);
}

@Override
public SavingBankAccountDTO saveSavingBankAccount(double
initialBalance, double interestRate, Long customerId) throws
CustomerNotFoundException {
    Customer
customer=customerRepository.findById(customerId).orElse(null);
    if(customer==null)
        throw new CustomerNotFoundException("Customer not found");
    SavingAccount savingAccount=new SavingAccount();
    savingAccount.setId(UUID.randomUUID().toString());
    savingAccount.setCreatedAt(new Date());
    savingAccount.setBalance(initialBalance);
    savingAccount.setInterestRate(interestRate);
    savingAccount.setCustomer(customer);
    SavingAccount savedBankAccount =
bankAccountRepository.save(savingAccount);
    return dtoMapper.fromSavingBankAccount(savedBankAccount);
}

@Override
public List<CustomerDTO> listCustomers() {
    List<Customer> customers = customerRepository.findAll();
    List<CustomerDTO> customerDTOS = customers.stream()
        .map(customer -> dtoMapper.fromCustomer(customer))
        .collect(Collectors.toList());
    /*
    List<CustomerDTO> customerDTOS=new ArrayList<>();

```

```

        for (Customer customer:customers){
            CustomerDTO customerDTO=dtoMapper.fromCustomer(customer);
            customerDTOS.add(customerDTO);
        }
        */
        return customerDTOS;
    }

    @Override
    public BankAccountDTO getBankAccount(String accountId) throws
    BankAccountNotFoundException {
        BankAccount
        bankAccount=bankAccountRepository.findById(accountId)
            .orElseThrow(()->new
            BankAccountNotFoundException("BankAccount not found"));
        if(bankAccount instanceof SavingAccount){
            SavingAccount savingAccount= (SavingAccount) bankAccount;
            return dtoMapper.fromSavingBankAccount(savingAccount);
        } else {
            CurrentAccount currentAccount= (CurrentAccount) bankAccount;
            return dtoMapper.fromCurrentBankAccount(currentAccount);
        }
    }

    @Override
    public void debit(String accountId, double amount, String
    description) throws BankAccountNotFoundException,
    BalanceNotSufficientException {
        BankAccount
        bankAccount=bankAccountRepository.findById(accountId)
            .orElseThrow(()->new
            BankAccountNotFoundException("BankAccount not found"));
        if(bankAccount.getBalance()<amount)
            throw new BalanceNotSufficientException("Balance not
            sufficient");
        AccountOperation accountOperation=new AccountOperation();
        accountOperation.setType(OperationType.DEBIT);
        accountOperation.setAmount(amount);
        accountOperation.setDescription(description);
        accountOperation.setOperationDate(new Date());
        accountOperation.setBankAccount(bankAccount);
        accountOperationRepository.save(accountOperation);
        bankAccount.setBalance(bankAccount.getBalance()-amount);
        bankAccountRepository.save(bankAccount);
    }

    @Override
    public void credit(String accountId, double amount, String
    description) throws BankAccountNotFoundException {
        BankAccount
        bankAccount=bankAccountRepository.findById(accountId)
            .orElseThrow(()->new
            BankAccountNotFoundException("BankAccount not found"));
        AccountOperation accountOperation=new AccountOperation();
        accountOperation.setType(OperationType.CREDIT);

```

```

        accountOperation.setAmount(amount);
        accountOperation.setDescription(description);
        accountOperation.setOperationDate(new Date());
        accountOperation.setBankAccount(bankAccount);
        accountOperationRepository.save(accountOperation);
        bankAccount.setBalance(bankAccount.getBalance()+amount);
        bankAccountRepository.save(bankAccount);
    }

    @Override
    public void transfer(String accountIdSource, String
accountIdDestination, double amount) throws
BankAccountNotFoundException, BalanceNotSufficientException {
        debit(accountIdSource, amount, "Transfer to
"+accountIdDestination);
        credit(accountIdDestination, amount, "Transfer from
"+accountIdSource);
    }

    @Override
    public List<BankAccountDTO> bankAccountList() {
        List<BankAccount> bankAccounts =
bankAccountRepository.findAll();
        List<BankAccountDTO> bankAccountDTOS =
bankAccounts.stream().map(bankAccount -> {
            if (bankAccount instanceof SavingAccount) {
                SavingAccount savingAccount = (SavingAccount)
bankAccount;
                return dtoMapper.fromSavingBankAccount(savingAccount);
            } else {
                CurrentAccount currentAccount = (CurrentAccount)
bankAccount;
                return dtoMapper.fromCurrentBankAccount(currentAccount);
            }
        }).collect(Collectors.toList());
        return bankAccountDTOS;
    }

    @Override
    public CustomerDTO getCustomer(Long customerId) throws
CustomerNotFoundException {
        Customer customer = customerRepository.findById(customerId)
            .orElseThrow(() -> new
CustomerNotFoundException("Customer Not found"));
        return dtoMapper.fromCustomer(customer);
    }

    @Override
    public CustomerDTO updateCustomer(CustomerDTO customerDTO) {
        log.info("Saving new Customer");
        Customer customer=dtoMapper.fromCustomerDTO(customerDTO);
        Customer savedCustomer = customerRepository.save(customer);
        return dtoMapper.fromCustomer(savedCustomer);
    }

    @Override
    public void deleteCustomer(Long customerId) {
        customerRepository.deleteById(customerId);
    }

```

```

        @Override
        public List<AccountOperationDTO> accountHistory(String accountId) {
            List<AccountOperation> accountOperations =
            accountOperationRepository.findByBankAccountId(accountId);
            return accountOperations.stream().map(op ->
            >dtoMapper.fromAccountOperation(op)).collect(Collectors.toList());
        }

        @Override
        public AccountHistoryDTO getAccountHistory(String accountId, int
        page, int size) throws BankAccountNotFoundException {
            BankAccount
            bankAccount=bankAccountRepository.findById(accountId).orElse(null);
            if(bankAccount==null) throw new
            BankAccountNotFoundException("Account not Found");
            Page<AccountOperation> accountOperations =
            accountOperationRepository.findByBankAccountIdOrderByOperationDateDesc(a
            ccountId, PageRequest.of(page, size));
            AccountHistoryDTO accountHistoryDTO=new AccountHistoryDTO();
            List<AccountOperationDTO> accountOperationDTOS =
            accountOperations.getContent().stream().map(op ->
            dtoMapper.fromAccountOperation(op)).collect(Collectors.toList());
            accountHistoryDTO.setAccountOperationDTOS(accountOperationDTOS);
            accountHistoryDTO.setAccountId(bankAccount.getId());
            accountHistoryDTO.setBalance(bankAccount.getBalance());
            accountHistoryDTO.setCurrentPage(page);
            accountHistoryDTO.setPageSize(size);

            accountHistoryDTO.setTotalPages(accountOperations.getTotalPages());
            return accountHistoryDTO;
        }

        @Override
        public List<CustomerDTO> searchCustomers(String keyword) {
            List<Customer>
            customers=customerRepository.searchCustomer(keyword);
            List<CustomerDTO> customerDTOS = customers.stream().map(cust ->
            dtoMapper.fromCustomer(cust)).collect(Collectors.toList());
            return customerDTOS;
        }
    }
}

```

## - Controller «CustomerRest »

```

- package ma.enset.digitalbanking.controllers;

import lombok.AllArgsConstructor;
import lombok.extern.slf4j.Slf4j;

import ma.enset.digitalbanking.dtos.CustomerDTO;
import ma.enset.digitalbanking.exceptions.CustomerNotFoundException;
import ma.enset.digitalbanking.services.BankAccountService;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController

```

```

@AllArgsConstructor
@Slf4j
@CrossOrigin("*")
public class CustomerRestController {
    private BankAccountService bankAccountService;
    @GetMapping("/customers")
    public List<CustomerDTO> customers() {
        return bankAccountService.listCustomers();
    }
    @GetMapping("/customers/search")
    public List<CustomerDTO> searchCustomers(@RequestParam(name =
"keyword",defaultValue = "") String keyword){
        return bankAccountService.searchCustomers("%"+keyword+"%");
    }
    @GetMapping("/customers/{id}")
    public CustomerDTO getCustomer(@PathVariable(name = "id") Long
customerId) throws CustomerNotFoundException {
        return bankAccountService.getCustomer(customerId);
    }
    @PostMapping("/customers")
    public CustomerDTO saveCustomer(@RequestBody CustomerDTO
customerDTO){
        return bankAccountService.saveCustomer(customerDTO);
    }
    @PutMapping("/customers/{customerId}")
    public CustomerDTO updateCustomer(@PathVariable Long customerId,
@RequestBody CustomerDTO customerDTO){
        customerDTO.setId(customerId);
        return bankAccountService.updateCustomer(customerDTO);
    }
    @DeleteMapping("/customers/{id}")
    public void deleteCustomer(@PathVariable Long id){
        bankAccountService.deleteCustomer(id);
    }
}

```

## - Controller «BankAccountI »

```

- package ma.enset.digitalbanking.controllers;

import ma.enset.digitalbanking.dtos.*;
import ma.enset.digitalbanking.dtos.BankAccountDTO;
import ma.enset.digitalbanking.dtos.CreditDTO;
import ma.enset.digitalbanking.dtos.DebitDTO;
import ma.enset.digitalbanking.exceptions.BalanceNotSufficientException;
import ma.enset.digitalbanking.exceptions.BankAccountNotFoundException;
import ma.enset.digitalbanking.services.BankAccountService;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@CrossOrigin("*")
public class BankAccountRestController {
    private BankAccountService bankAccountService;

```



```

    public BankAccountRestController(BankAccountService
bankAccountService) {
        this.bankAccountService = bankAccountService;
    }

    @GetMapping("/accounts/{accountId}")
    public BankAccountDTO getBankAccount(@PathVariable String accountId)
throws BankAccountNotFoundException {
        return bankAccountService.getBankAccount(accountId);
    }
    @GetMapping("/accounts")
    public List<BankAccountDTO> listAccounts() {
        return bankAccountService.bankAccountList();
    }
    @GetMapping("/accounts/{accountId}/operations")
    public List<AccountOperationDTO> getHistory(@PathVariable String
accountId) {
        return bankAccountService.accountHistory(accountId);
    }

    @GetMapping("/accounts/{accountId}/pageOperations")
    public AccountHistoryDTO getAccountHistory(
        @PathVariable String accountId,
        @RequestParam(name="page",defaultValue = "0") int page,
        @RequestParam(name="size",defaultValue = "5")int size)
throws BankAccountNotFoundException {
        return
bankAccountService.getAccountHistory(accountId,page,size);
    }
    @PostMapping("/accounts/debit")
    public DebitDTO debit(@RequestBody DebitDTO debitDTO) throws
BankAccountNotFoundException, BalanceNotSufficientException,
BalanceNotSufficientException {

this.bankAccountService.debit(debitDTO.getAccountId(),debitDTO.getAmount
(),debitDTO.getDescription());
        return debitDTO;
    }
    @PostMapping("/accounts/credit")
    public CreditDTO credit(@RequestBody CreditDTO creditDTO) throws
BankAccountNotFoundException {

this.bankAccountService.credit(creditDTO.getAccountId(),creditDTO.getAmo
unt(),creditDTO.getDescription());
        return creditDTO;
    }
    @PostMapping("/accounts/transfer")
    public void transfer(@RequestBody TransferRequestDTO
transferRequestDTO) throws BankAccountNotFoundException,
BalanceNotSufficientException {
        this.bankAccountService.transfer(
            transferRequestDTO.getAccountSource(),
            transferRequestDTO.getAccountDestination(),
            transferRequestDTO.getAmount());
    }
}

```

```
}
```

## - Properties

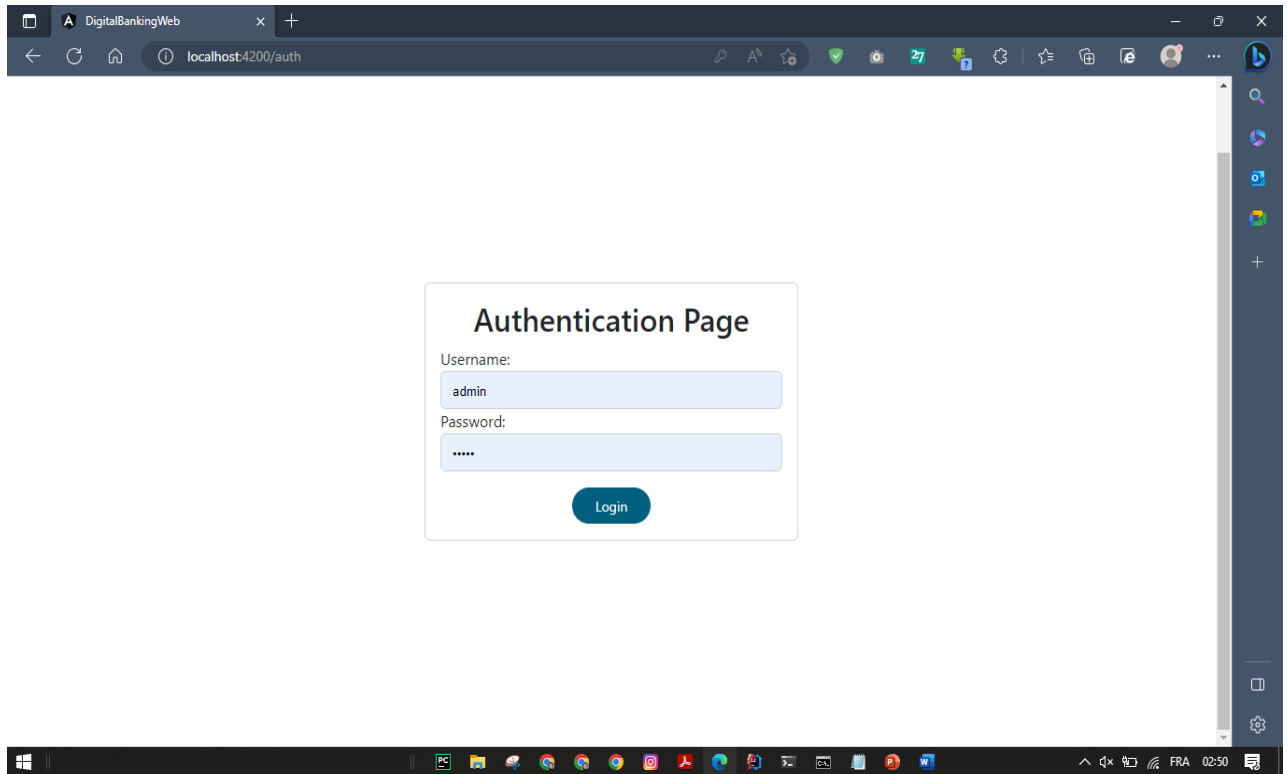
```
#spring.datasource.url=jdbc:h2:mem:E-BANK  
#spring.h2.console.enabled=true  
server.port=9999  
spring.datasource.url=jdbc:mysql://localhost:3306/E-  
BANK?createDatabaseIfNotExist=true  
spring.datasource.username=root  
spring.datasource.password=fatiza  
spring.jpa.hibernate.ddl-auto = update  
spring.jpa.properties.hibernate.dialect =  
org.hibernate.dialect.MariaDBDialect
```

## 1.2. Coté frontend

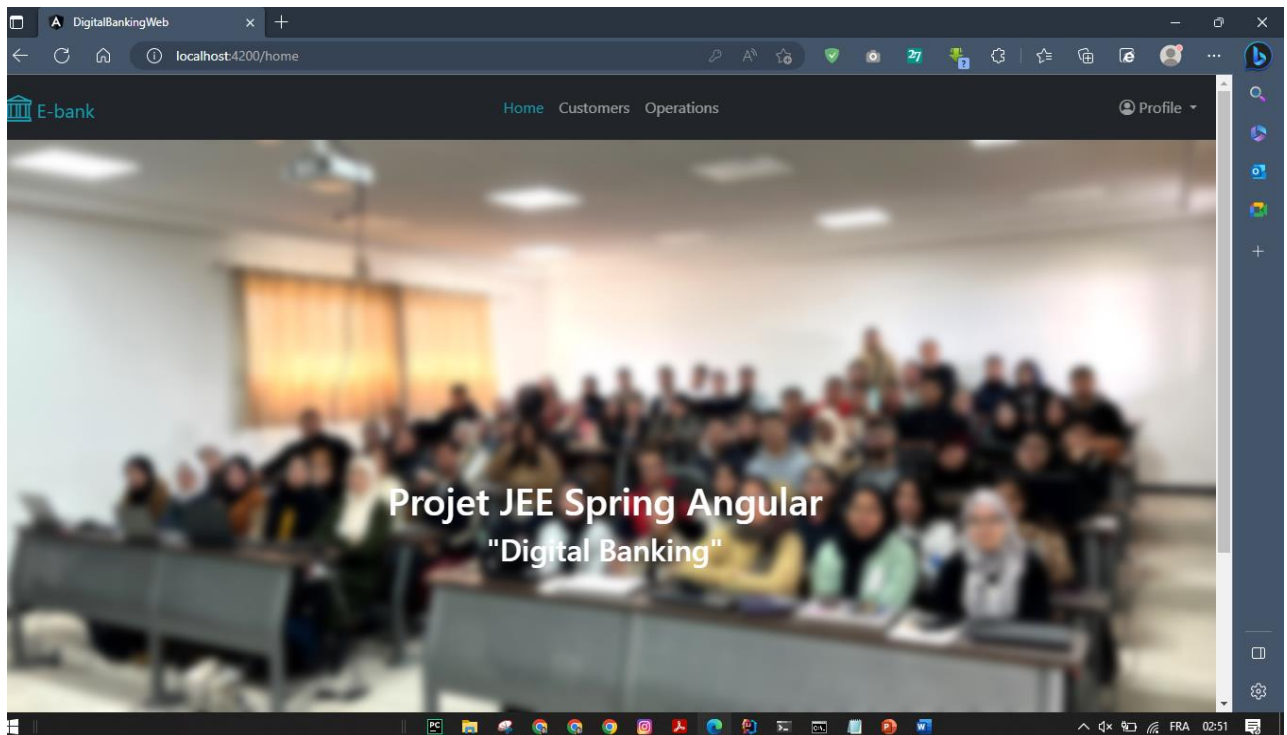
Voir le code source dans le fichier .zip

### 3. Interfaces

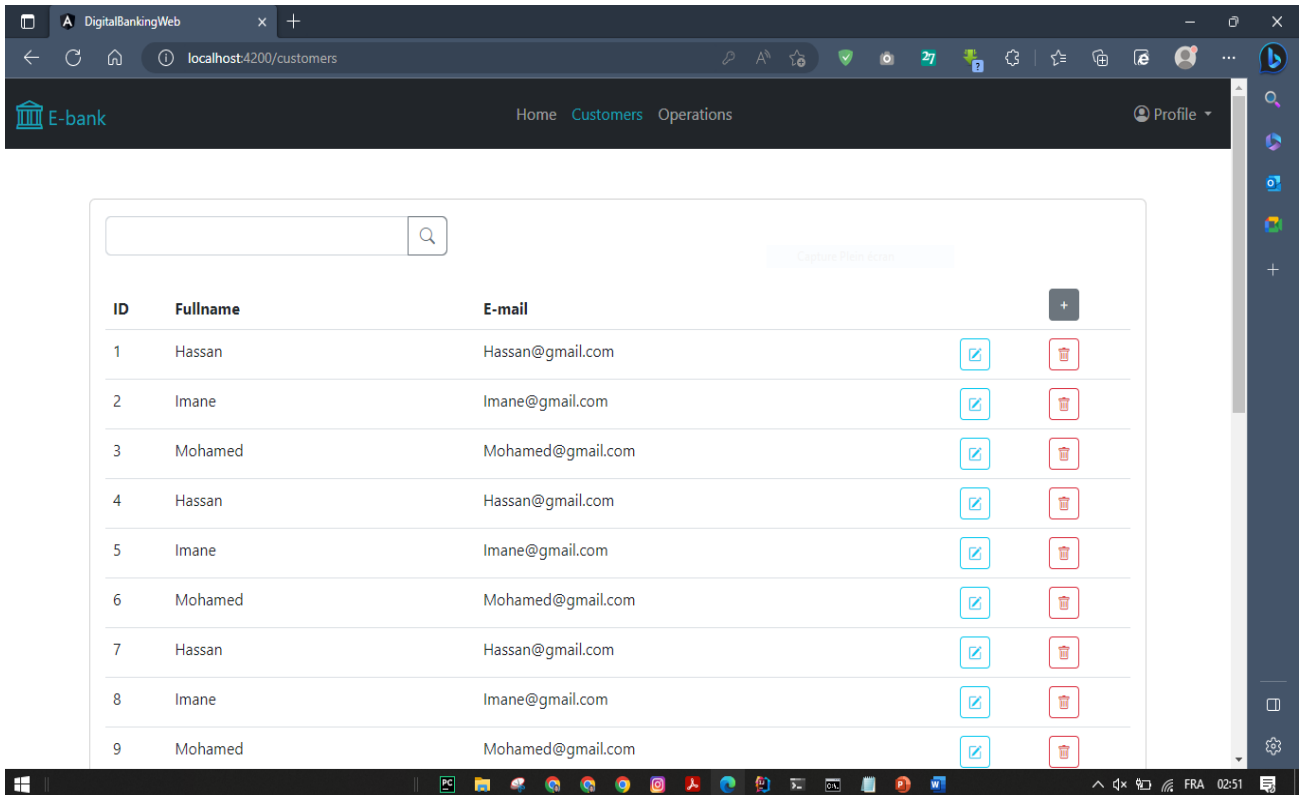
#### - Authentication



#### - Home



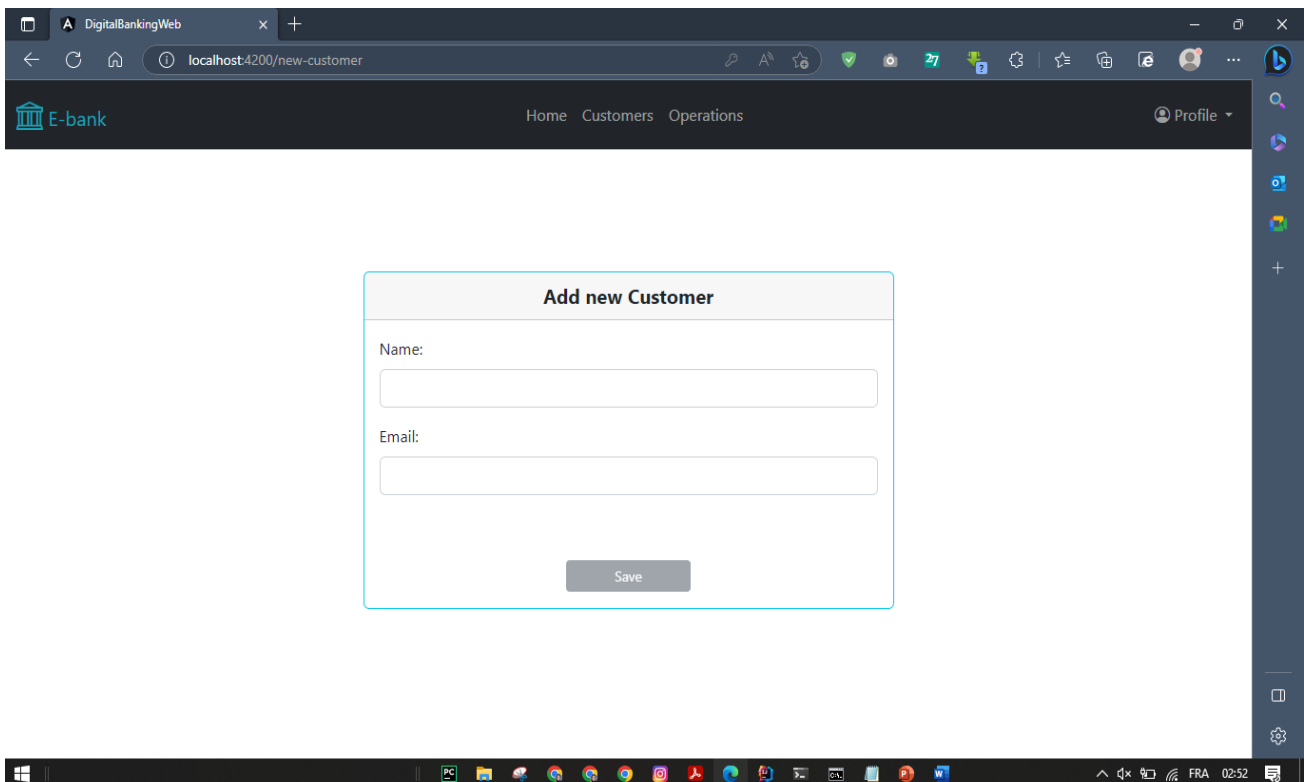
## - Liste de customers



The screenshot shows a web browser window with the URL `localhost:4200/customers`. The application header includes the 'E-bank' logo and navigation links for 'Home', 'Customers', and 'Operations'. A 'Profile' dropdown menu is visible in the top right corner. The main content area features a search bar and a table of customers.

ID	Fullname	E-mail		
1	Hassan	Hassan@gmail.com		
2	Imane	Imane@gmail.com		
3	Mohamed	Mohamed@gmail.com		
4	Hassan	Hassan@gmail.com		
5	Imane	Imane@gmail.com		
6	Mohamed	Mohamed@gmail.com		
7	Hassan	Hassan@gmail.com		
8	Imane	Imane@gmail.com		
9	Mohamed	Mohamed@gmail.com		

## - Ajouter customer



The screenshot shows a web browser window with the URL `localhost:4200/new-customer`. The application header is the same as the previous screenshot. The main content area displays a form titled 'Add new Customer'.

**Add new Customer**

Name:

Email:

## - Modifier customer

The screenshot shows a web browser window with the address bar displaying 'localhost:4200/update-customer/1'. The page has a dark header with 'E-bank' logo, 'Home', 'Customers', 'Operations', and a 'Profile' dropdown. The main content area features a 'Modify Customer' form with the following fields: 'ID' (value: 1), 'Name' (value: Hassan), and 'Email' (value: Hassan@gmail.com). An 'Update' button is at the bottom of the form. A 'Capture Photo Icon' button is visible above the form. The Windows taskbar at the bottom shows various application icons and the system clock at 02:52.

Modify Customer

ID:  
1

Name:  
Hassan

Email:  
Hassan@gmail.com

Update

## - Effectuer opération

The screenshot shows a web browser window with the address bar displaying 'localhost:4200/operations'. The page has a dark header with 'E-bank' logo, 'Home', 'Customers', 'Operations', and a 'Profile' dropdown. The main content area is divided into two sections: 'Accounts' on the left and 'Add Operation' on the right. The 'Accounts' section has a search bar with '1' and a magnifying glass icon. The 'Add Operation' section has radio buttons for 'DEBIT', 'CREDIT', and 'TRANSFER', followed by 'Amount' (value: 0) and 'Description' fields. A 'Save Operation' button is at the bottom of the 'Add Operation' section. Below this is an 'Operations list' section with a table. The Windows taskbar at the bottom shows various application icons and the system clock at 02:52.

Accounts

1

Add Operation

☐ DEBIT: ☐ CREDIT: ☐ TRANSFER:

Amount :  
0

Description :

Save Operation

Operations list

## IV. Conclusion

En conclusion, ce rapport a présenté en détail le projet de création d'une application de gestion de comptes bancaires. Nous avons abordé les différentes parties du projet, notamment la conception et l'architecture, la réalisation et l'implémentation, ainsi que les technologies et les outils utilisés. Nous avons mis en évidence l'utilisation de l'architecture MVC avec Spring et Angular, ainsi que la base de données MySQL.

Ce projet nous a permis de mettre en pratique nos connaissances en développement web, en utilisant des frameworks tels que Spring Boot et Angular pour créer une application fonctionnelle et conviviale. Nous avons également mis en place des fonctionnalités clés, telles que la gestion des comptes bancaires, les opérations de débit et de crédit, et la sécurité avec Spring Security et JWT.

Au cours de ce projet, nous avons pu constater l'importance d'une bonne planification et d'une conception solide pour assurer le bon déroulement du développement. Nous avons également compris l'importance de la communication et de la collaboration au sein de l'équipe de développement, en travaillant conjointement sur les différentes parties du projet.

En conclusion, ce projet nous a permis d'acquérir une expérience précieuse dans le développement d'applications web, en mettant en pratique les principes de l'architecture MVC, en utilisant des technologies modernes et en relevant les défis liés à la gestion des comptes bancaires. Nous sommes fiers du résultat obtenu et nous sommes convaincus que cette application pourra répondre aux besoins des utilisateurs en matière de gestion de comptes bancaires de manière efficace et sécurisée.

**- Merci pour votre lecture -**