

Rapport

Digital Banking

Daabal Sokaina

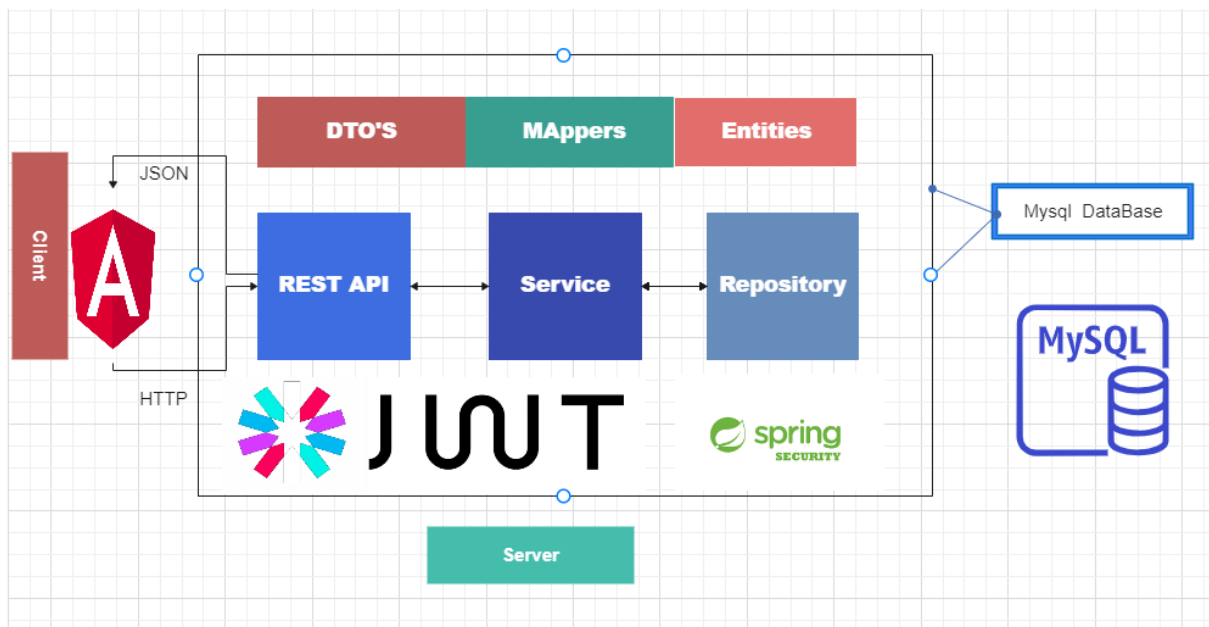
29 mai 2023


Introduction

L'objectif de ce rapport est de présenter le développement d'une application de gestion des comptes bancaires. L'application vise à fournir un système complet pour gérer les comptes bancaires des clients, en prenant en compte les opérations de débit et de crédit. De plus, elle propose deux types de comptes : les comptes courants et les comptes épargnes.

Dans ce rapport, nous allons présenter les différentes parties de l'application, en commençant par la couche DAO, suivie de la couche services, DTO et mappers, puis la couche Web avec les RestControllers. Enfin, nous aborderons la partie Front End Angular et la sécurité avec Spring Security et JWT.

Architecture de projet





L'architecture du projet de l'application de gestion des comptes bancaires suit une approche en couches (Layered Architecture), qui permet de séparer les différentes responsabilités et de favoriser la modularité et la maintenabilité du code.

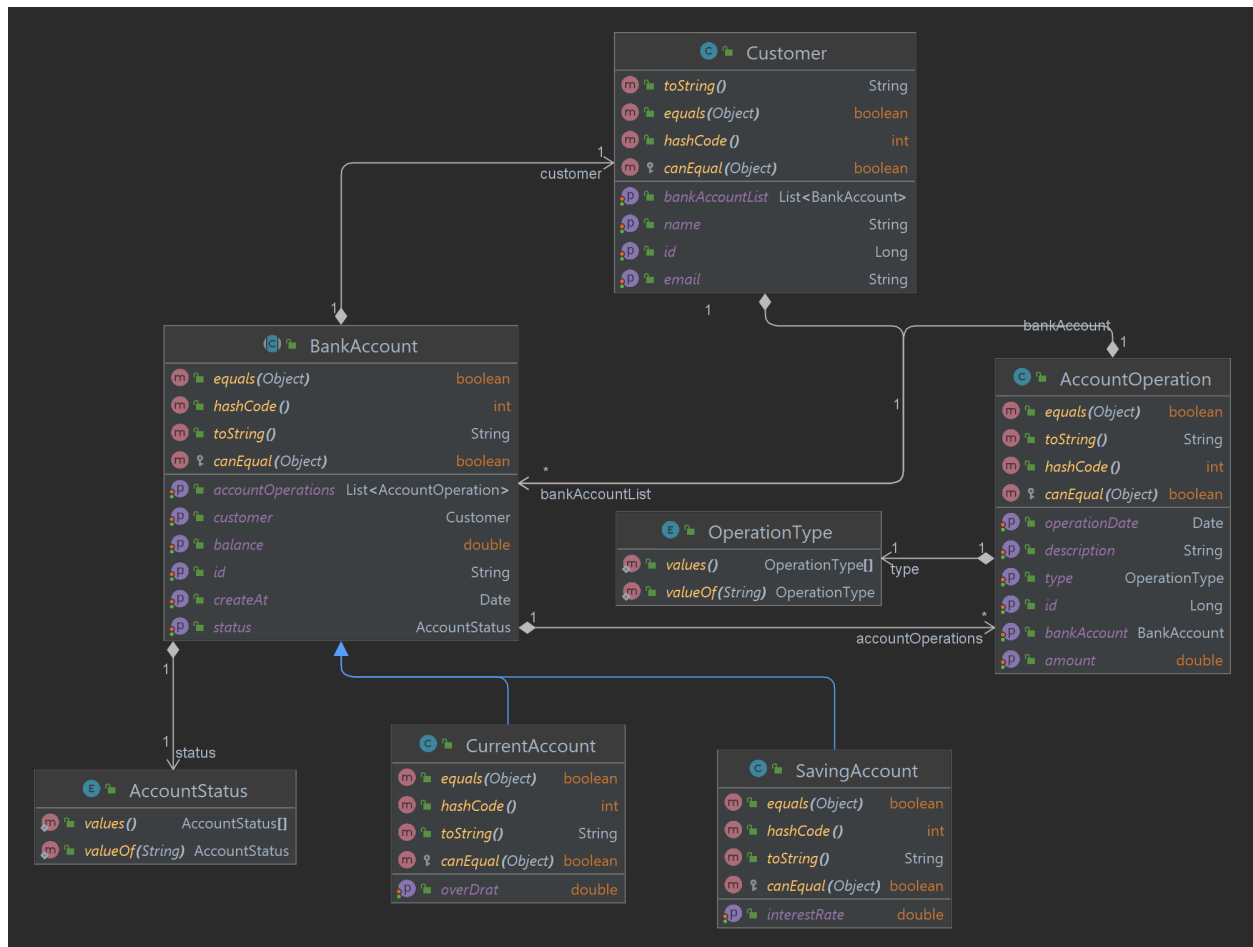
Voici les couches de notre applications:

1. Couche DAO (Data Access Object) :
 - Cette couche est responsable de l'accès aux données et de la communication avec la base de données.
 - Elle comprend les entités JPA (Java Persistence API) qui représentent les objets métier de l'application, tels que les clients, les comptes bancaires, et les opérations sur les comptes.
 - Les interfaces JPA Repository basées sur Spring Data sont utilisées pour effectuer des opérations CRUD (Create, Read, Update, Delete) sur les entités et interagir avec la base de données.
2. Couche services, DTO et mappers :
 - Cette couche contient la logique métier de l'application, telle que la gestion des opérations sur les comptes et les clients.
 - Les services sont des classes qui implémentent la logique métier et utilisent les interfaces JPA Repository pour accéder aux données.
 - Les objets DTO (Data Transfer Objects) sont utilisés pour transférer les données entre les différentes couches de l'application, en ne transmettant que les informations nécessaires et en évitant les dépendances directes sur les entités JPA.
 - Les mappers sont utilisés pour convertir les entités JPA en objets DTO et vice versa, facilitant ainsi la manipulation des données entre les différentes couches.
3. Couche Web (RestController) :
 - Cette couche fournit une interface Web pour interagir avec l'application.
 - Les RestControllers sont des classes qui exposent les points de terminaison (endpoints) HTTP pour les fonctionnalités de l'application, tels que la création d'un compte, la récupération des informations d'un compte, etc.
 - Les RestControllers utilisent les services de la couche précédente pour traiter les requêtes HTTP, effectuer les opérations nécessaires et renvoyer les réponses appropriées.
4. Couche Front-end Angular :
 - Cette couche concerne l'interface utilisateur de l'application.

- Elle utilise le framework Angular pour développer des composants qui affichent les différentes fonctionnalités de l'application.
 - Les composants Angular communiquent avec les RestControllers de l'application en utilisant des services Angular pour effectuer les requêtes HTTP et obtenir les données nécessaires.
 - Les templates HTML et les styles CSS sont utilisés pour concevoir l'interface utilisateur attrayante et conviviale.
5. Couche Sécurité avec Spring Security et JWT :
- Cette couche assure la sécurité de l'application.
 - Spring Security est utilisé pour la gestion de l'authentification et de l'autorisation des utilisateurs.
 - JWT (JSON Web Tokens) est utilisé comme mécanisme d'authentification stateless, stockant les informations d'identification de l'utilisateur de manière sécurisée.
 - Des filtres de sécurité sont mis en place pour valider les jetons JWT, vérifier les autorisations des utilisateurs et sécuriser les ressources de l'application.

L'architecture en couches du projet permet de séparer les responsabilités et de faciliter la collaboration entre les différentes parties de l'application. Elle favorise également la maintenabilité et l'évolutivité du code en permettant des modifications spécifiques à une couche sans affecter les autres couches.

Conception de l'application



Partie 1 : Couche DAO

Dans cette partie, vous avez créé un projet Spring Boot pour votre application.

Nous avons défini les entités JPA suivantes :

- **Customer** : représente un client avec ses informations personnelles.

```

1 package ma.enset.ebankingbackend.entities;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6
7 import javax.persistence.*;
8 import java.util.List;
9
10 @Entity
11 @Data
12 @NoArgsConstructor
13 @AllArgsConstructor
14 public class Customer
15 {
16     @Id
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     private Long id;
19     private String name;
20     private String email;
21
22     // un client peut avoir plusieurs comptes pour cela on va voir une list des comptes
23     // relation bidirectionnelle
24     // @JsonProperty(access = JsonProperty.Access.WRITE_ONLY) pour ne pas recuperer toute les accounts lorsque va utiliser la methode get
25     @OneToMany(mappedBy = "customer") // il faut dire a JPA que cette et le meme presente dans la class BanckAccount avec l'attribut customer
26     private List<BankAccount> bankAccountList;
27
28 }
29

```


- **BankAccount** : représente un compte bancaire générique.

```

1 package ma.enset.ebankingbackend.entities;
2
3
4 import lombok.AllArgsConstructor;
5 import lombok.Data;
6 import lombok.NoArgsConstructor;
7 import ma.enset.ebankingbackend.enums.AccountStatus;
8
9 import javax.persistence.*;
10 import java.util.Date;
11 import java.util.List;
12
13 @Entity
14 @Data
15 @NoArgsConstructor
16 @AllArgsConstructor
17 // heritage
18 /**
19  * les strategies de mapping heritage
20  * 1 -> InheritanceType.SINGLE_TABLE : {
21  *   - creation d'une seule table pour toute hierarchy
22  *   - toutes les attributs de classe BankAccount + attributs de les attributs de classe qui herite de classe principale
23  *   - plus columns TYPE : DiscriminatorColumn qui va dire que si le compte est de type current ou saving account
24  *   - Inconvenient : pour chacun ligne, un columns ne sera pas utilisées
25  * };
26  * 2 -> Table Per class : {
27  *   - on va cree table pour currentAccount et autre pour SavingAccount, les deux tables ont la meme structure ;
28  *   - on utilise lorsqu'on a une grande difference dans les classes derive
29  * };
30  * 3 -> Joined Table : {
31  *   - creation de trois table BankAccount, CurrentAccount ( leur attribut, lien avec table BankAccount -> id_Account ), SavingAccount ( leur attribut, lien avec table BankAccount -> id_Account);
32  *   - creation d'une relation entre les tables;
33  * };
34  */
35 // @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
36 @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
37 // la columns qui sera ajouter a la table pour determiner le type de compte
38 @DiscriminatorColumn(name = "Type", length = 4)
39 public abstract class BankAccount {
40     @Id
41     private String id; // dans ce contexte, c'est le rib
42     private double balance;
43     private Date createAt;
44     @Enumerate(EnumType.STRING) // enumerator dans la base de donnee au format string
45     private AccountStatus status;
46     @ManyToOne
47     private Customer customer; // lien le compte a un client
48     @OneToMany(mappedBy = "bankAccount", fetch = FetchType.LAZY) // fetch lazy(par default) ne charge pas la liste des operation sur compte, Eager va charger la liste des operations danger risque de charger des donnee que nous n'avons pas besoins
49     private List<AccountOperation> accountOperations; // les operations que peut effectuer dans un compte
50
51 }

```

- **SavingAccount** : représente un compte épargne, qui est un type spécifique de compte bancaire.




```

1  package ma.enset.ebankingbackend.entities;
2
3  import lombok.AllArgsConstructor;
4  import lombok.Data;
5  import lombok.NoArgsConstructor;
6
7  import javax.persistence.DiscriminatorValue;
8  import javax.persistence.Entity;
9
10 @Entity
11 @Data
12 @NoArgsConstructor
13 @AllArgsConstructor
14 // determiner le type que va insert dans la columns type
15 @DiscriminatorValue("SA")
16 public class SavingAccount extends BankAccount{
17     private double interestRate;
18 }
19

```

- CurrentAccount : représente un compte courant, qui est un autre type spécifique de compte bancaire.



```


1  package ma.enset.ebankingbackend.entities;
2
3  import lombok.AllArgsConstructor;
4  import lombok.Data;
5  import lombok.NoArgsConstructor;
6
7  import javax.persistence.DiscriminatorValue;
8  import javax.persistence.Entity;
9
10 @Entity
11 @DiscriminatorValue("CA")
12 @Data @NoArgsConstructor @AllArgsConstructor
13 public class CurrentAccount extends BankAccount{
14     // credit negative
15     private double overDrat;
16 }
17

```

- AccountOperation : représente une opération effectuée sur un compte, telle que DEBIT ou CREDIT.

```
1 package ma.enset.ebankingbackend.entities;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6 import ma.enset.ebankingbackend.enums.OperationType;
7
8 import javax.persistence.*;
9 import java.util.Date;
10
11 @Entity
12 @Data
13 @NoArgsConstructor
14 @AllArgsConstructor
15 public class AccountOperation {
16     @Id
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     private Long id;
19     private Date operationDate;
20     private double amount;
21     private String description;
22     @Enumerated(EnumType.STRING)
23     private OperationType type;
24     @ManyToOne
25     private BankAccount bankAccount;
26
27 }
28
```

Nous avons mis en place les interfaces JPA Repository en utilisant Spring Data, qui nous permettent de communiquer avec la base de données pour effectuer des opérations CRUD (Create, Read, Update, Delete) sur les entités.




Powered by KikiManjaro

```

1 package ma.enset.ebankingbackend.repositories;
2
3 import ma.enset.ebankingbackend.entities.AccountOperation;
4 import org.springframework.data.domain.Page;
5 import org.springframework.data.domain.Pageable;
6 import org.springframework.data.jpa.repository.JpaRepository;
7
8 import java.util.List;
9
10 public interface AccountOperationRepository extends JpaRepository<AccountOperation, Long> {
11     // permet de retourner une liste operation realise dans le compte recherche par le compte
12     List<AccountOperation> findByBankAccountId(String accountId);
13
14     // Pagination
15     Page<AccountOperation> findByBankAccountId(String accountId, Pageable pageable);
16
17     Page<AccountOperation> findByBankAccountIdOrderByOperationDateDesc(String accountId, Pageable pageable);
18 }
19
20

```




Powered by KikiManjaro

```

1 package ma.enset.ebankingbackend.repositories;
2
3 import ma.enset.ebankingbackend.entities.BankAccount;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 import java.util.List;
7
8 public interface BankAccountRepository extends JpaRepository<BankAccount, String> {
9     List<BankAccount> findByCustomerId(Long id);
10 }
11

```



Powered by KikiManjaro

```

1 package ma.enset.ebankingbackend.repositories;
2
3 import ma.enset.ebankingbackend.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> findByNameContains(@Param("kw") String keyWord);
13 }
14

```

Nous avons écrit des tests unitaires pour vérifier le bon fonctionnement de la couche DAO. Ces tests nous ont permis de s'assurer que les opérations CRUD fonctionnent correctement, que les relations entre les entités sont gérées de manière appropriée, et que les requêtes vers la base de données se déroulent sans erreur.

```
1 CommandLineRunner start(CustomerRepository customerRepository,
2                           BankAccountRepository bankAccountRepository,
3                           AccountOperationRepository accountOperationRepository){
4     return args -> {
5
6         // test d'entité customer
7         Stream.of("AhmedSaad", "Saad", "MohammedSaad", "SokainaSaad").forEach(name->{
8             Customer customer = new Customer();
9             customer.setName(name);
10            customer.setEmail(name+"@gmail.com");
11            customerRepository.save(customer);
12        });
13
14        // test d'entité bankAccount
15        customerRepository.findAll().forEach(customer -> {
16            // current account
17
18            CurrentAccount account = new CurrentAccount();
19            account.setId(UUID.randomUUID().toString());
20            account.setBalance(Math.random()*90000);
21            account.setCreateAt(new Date());
22            account.setStatus(AccountStatus.CREATED);
23            account.setCustomer(customer);
24            account.setOverDrat(9000);
25            bankAccountRepository.save(account);
26            // saving account
27            SavingAccount savingAccount = new SavingAccount();
28            savingAccount.setId(UUID.randomUUID().toString());
29            savingAccount.setBalance(Math.random()*90000);
30            savingAccount.setCreateAt(new Date());
31            savingAccount.setStatus(AccountStatus.CREATED);
32            savingAccount.setCustomer(customer);
33            savingAccount.setInterestRate(9000);
34            bankAccountRepository.save(savingAccount);
35        });
36        // test d'entité List operation dans les comptes cree
37
38        bankAccountRepository.findAll().forEach(bankAccount -> {
39            for (int i=0; i<10; i++) {
40                AccountOperation accountOperation = new AccountOperation();
41                accountOperation.setOperationDate(new Date());
42                accountOperation.setAmount(Math.random()*12000);
43                accountOperation.setType(Math.random() > 0.5 ? OperationType.DEBIT : OperationType.CREDIT);
44                accountOperation.setBankAccount(bankAccount);
45                accountOperationRepository.save(accountOperation);
46            }
47        });
48
49        BankAccount bankAccount = bankAccountRepository.findById("39ebc57a-e601-4e1b-b93a-279cf7b6ed07").orElse(null);
50        System.out.println("*****");
51        System.out.println(bankAccount.getId());
52        System.out.println(bankAccount.getBalance());
53        System.out.println(bankAccount.getStatus());
54        System.out.println(bankAccount.getCreateAt());
55        System.out.println(bankAccount.getCustomer().getName());
56        System.out.println(bankAccount.getClass().getSimpleName());
57        if(bankAccount instanceof CurrentAccount)
58        {
59            System.out.println(((CurrentAccount)bankAccount).getOverDrat());
60        }
61        else if(bankAccount instanceof SavingAccount){
62            System.out.println(((SavingAccount)bankAccount).getInterestRate());
63        }
64        bankAccount.getAccountOperations().forEach(accountOperation -> {
65            System.out.println("*****");
66            System.out.println(accountOperation.getAmount());
67            System.out.println(accountOperation.getType());
68            System.out.println(accountOperation.getOperationDate());
69        });
70
71    };
72 }
```

Partie 2 : Couche services, DTO et mappers

Dans cette partie, nous avons mis en place la couche services de l'application. Nous avons créé des classes de services qui contiennent la logique métier pour gérer les opérations sur les comptes bancaires et les clients. Pour faciliter l'échange de données entre les différentes couches de l'application, nous avons défini des objets DTO (Data Transfer Objects) qui représentent les données échangées. De plus, nous avons utilisé des mappers pour convertir les entités JPA en DTO et vice versa, simplifiant ainsi la manipulation des données.

Les DTOS

voilà un exemple de l'implémentation des mappers :

```
1  package ma.enset.ebankingbackend.dtos;
2
3  import lombok.Data;
4
5  import java.util.List;
6
7  @Data
8  public class AccountHistoryDTO {
9      private String accountId;
10     private double balance;
11     private String type;
12     private int currentPage;
13     private int totalePage;
14     private int pageSize;
15     private List<AccountOperationDTO> accountOperationDTOS;
16 }
```



Powered by KikiManjaro

```
1  package ma.enset.ebankingbackend.dtos;
2
3
4  import lombok.Data;
5  import ma.enset.ebankingbackend.enums.OperationType;
6
7  import java.util.Date;
8
9  @Data
10 public class AccountOperationDTO {
11     private Long id;
12     private Date operationDate;
13     private double amount;
14     private OperationType type;
15     private String description;
16 }
17
```

Mappers

voilà un exemple de mappers

```

1  package ma.enset.ebankingbackend.mappers;
2
3  import ma.enset.ebankingbackend.dtos.AccountOperationDTO;
4  import ma.enset.ebankingbackend.dtos.CurrentBankAccountDTO;
5  import ma.enset.ebankingbackend.dtos.CustomerDTO;
6  import ma.enset.ebankingbackend.dtos.SavingBankAccountDTO;
7  import ma.enset.ebankingbackend.entities.AccountOperation;
8  import ma.enset.ebankingbackend.entities.CurrentAccount;
9  import ma.enset.ebankingbackend.entities.Customer;
10 import ma.enset.ebankingbackend.entities.SavingAccount;
11 import org.springframework.beans.BeanUtils;
12 import org.springframework.stereotype.Service;
13
14 // framework : MapStruct qui permet de convertir d'un objet a un autre, ont besoin seulement de cree la signature de fonction
15 @Service // service que va injecter dans nos services
16 public class BankAccountMapperImpl
17 {
18     public CustomerDTO fromCustomer(Customer customer){
19         CustomerDTO customerDTO = new CustomerDTO();
20         // va prendre un objet de type customer et le copy dans un autre objet de type customerDTO
21         // BeanUtils.copyProperties(source,destination);
22         BeanUtils.copyProperties(customer, customerDTO);
23         return customerDTO;
24     }
25     public Customer fromCustomerDTO(CustomerDTO customerDTO){
26         Customer customer = new Customer();
27         BeanUtils.copyProperties(customerDTO, customer);
28         return customer;
29     }
30     public SavingBankAccountDTO fromSavingBankAccount(SavingAccount savingAccount){
31         SavingBankAccountDTO savingBankAccountDTO = new SavingBankAccountDTO();
32         BeanUtils.copyProperties(savingAccount, savingBankAccountDTO);
33         savingBankAccountDTO.setCustomerDTO(fromCustomer(savingAccount.getCustomer()));
34         savingBankAccountDTO.setType(savingAccount.getClass().getSimpleName());
35         return savingBankAccountDTO;
36     }
37
38     public SavingAccount fromSavingBankAccountDTO(SavingBankAccountDTO savingBankAccountDTO){
39         SavingAccount savingAccount = new SavingAccount();
40         BeanUtils.copyProperties(savingBankAccountDTO, savingAccount);
41         savingAccount.setCustomer(fromCustomerDTO(savingBankAccountDTO.getCustomerDTO()));
42         return savingAccount;
43     }
44
45     public CurrentBankAccountDTO fromCurrentBankAccount(CurrentAccount currentBankAccount){
46         CurrentBankAccountDTO currentAccountDto = new CurrentBankAccountDTO();
47         BeanUtils.copyProperties(currentBankAccount, currentAccountDto);
48         currentAccountDto.setCustomerDTO(fromCustomer(currentBankAccount.getCustomer()));
49         currentAccountDto.setType(currentBankAccount.getClass().getSimpleName());
50         return currentAccountDto;
51     }
52
53     public CurrentAccount fromCurrentBankAccountDTO(CurrentBankAccountDTO currentBankAccountDTO){
54         CurrentAccount currentAccount = new CurrentAccount();
55         BeanUtils.copyProperties(currentBankAccountDTO, currentAccount);
56         currentAccount.setCustomer(fromCustomerDTO(currentBankAccountDTO.getCustomerDTO()));
57         return currentAccount;
58     }
59     public AccountOperationDTO fromAccountOperation(AccountOperation accountOperation){
60         AccountOperationDTO accountOperationDTO = new AccountOperationDTO();
61         BeanUtils.copyProperties(accountOperation, accountOperationDTO);
62         return accountOperationDTO;
63     }
64 }
65

```

Les Services

```
1 package ma.enset.ebankingbackend.services;
2
3 import ma.enset.ebankingbackend.dtos.*;
4 import ma.enset.ebankingbackend.exceptions.BalanceNotSufficientException;
5 import ma.enset.ebankingbackend.exceptions.BankAccountNotFoundException;
6 import ma.enset.ebankingbackend.exceptions.CustomerNotFoundException;
7 import java.util.List;
8
9 // define les besoins fonctionnels de l'application
10
11 public interface BankAccountService {
12
13     // creation d'un client
14     CustomerDTO saveCustomer(CustomerDTO customerDTO);
15
16     // creation d'un compte current
17     CurrentBankAccountDTO saveCurrentBankAccount(double initialBalance, double ovrDraft, Long customerId) throws CustomerNotFoundException;
18
19     // creation d'un compte saving
20     SavingBankAccountDTO saveSavingBankAccountDTO(double initialBalance, double interestRate, Long customerId) throws CustomerNotFoundException;
21
22     // recuperate une list des clients
23     List<CustomerDTO> listCustomer();
24
25     // recuperate un compte en utilisant leur identifiant
26     BankAccountDTO getBankAccount(String accountId) throws BankAccountNotFoundException;
27     // ajout
28     void debit(String accountId, double amount, String description) throws BankAccountNotFoundException, BalanceNotSufficientException;
29     // retry
30     void credit(String accountId, double amount, String description) throws BankAccountNotFoundException;
31     // transfer de compte a un autre compte
32     void transfer(String accountIdSource, String accountIdDestination, double amount) throws BankAccountNotFoundException, BalanceNotSufficientException;
33     List<BankAccountDTO> bankAccountList();
34     CustomerDTO getCustomer(Long customerId) throws CustomerNotFoundException;
35     CustomerDTO updateCustomer(CustomerDTO customerDTO) throws CustomerNotFoundException;
36     void deleteCustomer(Long customerId);
37     List<AccountOperationDTO> accountHistory(String accountId);
38     AccountHistoryDTO getAccountHistory(String accountId, int page, int size) throws BankAccountNotFoundException;
39
40     List<BankAccountDTO> getCustomerBankAccount(Long id) throws CustomerNotFoundException;
41
42     List<CustomerDTO> searchCustomer(String keyword);
43 }
44
```

l'implémentation de quelque fonctionnalité bankAccountService est comme suite :

```

1  @Override
2  public List<BankAccountDTO> bankAccountList() {
3      List<BankAccount> bankAccounts = bankAccountRepository.findAll();
4      List<BankAccountDTO> bankAccountDTOS = bankAccounts.stream().map(bankAccount -> {
5          if (bankAccount instanceof SavingAccount) {
6              SavingAccount savingAccount = (SavingAccount) bankAccount;
7              return dtoMapper.fromSavingBankAccount(savingAccount);
8          } else {
9              CurrentAccount currentAccount = (CurrentAccount) bankAccount;
10             return dtoMapper.fromCurrentBankAccount(currentAccount);
11         }
12     }).collect(Collectors.toList());
13     return bankAccountDTOS;
14 }
15
16 @Override
17 public CustomerDTO getCustomer(Long customerId) throws CustomerNotFoundException {
18     Customer customer = customerRepository.findById(customerId).orElseThrow(() -> new CustomerNotFoundException("Customer not Found"));
19     return dtoMapper.fromCustomer(customer);
20 }
21
22 @Override
23 public CustomerDTO updateCustomer(CustomerDTO customerDTO) throws CustomerNotFoundException {
24     log.info("update a customer");
25     Customer customer = dtoMapper.fromCustomerDTO(customerDTO);
26     Customer updateCustomer = customerRepository.save(customer);
27     return dtoMapper.fromCustomer(updateCustomer);
28 }
29
30 @Override
31 public void deleteCustomer(Long customerId) {
32     customerRepository.deleteById(customerId);
33 }
34
35 @Override
36 public List<AccountOperationDTO> accountHistory(String accountId) {
37     List<AccountOperation> accountOperations = accountOperationRepository.findByBankAccountId(accountId);
38     List<AccountOperationDTO> accountOperationDTOS = accountOperations.stream().map(op -> dtoMapper.fromAccountOperation(op)).collect(Collectors.toList());
39     return accountOperationDTOS;
40 }
41
42 @Override
43 public AccountHistoryDTO getAccountHistory(String accountId, int page, int size) throws BankAccountNotFoundException {
44     BankAccount bankAccount = bankAccountRepository.findById(accountId).orElse(null);
45     if (bankAccount == null) throw new BankAccountNotFoundException("bank Account not found");
46     Page<AccountOperation> accountOperationPage = accountOperationRepository.findByBankAccountIdOrderByOperationDateDesc(accountId, PageRequest.of(page, size));
47     AccountHistoryDTO accountHistoryDTO = new AccountHistoryDTO();
48     List<AccountOperationDTO> accountOperationDTOS = accountOperationPage.getContent().stream().map(op -> dtoMapper.fromAccountOperation(op)).collect(Collectors.toList());
49     accountHistoryDTO.setAccountOperationDTOS(accountOperationDTOS);
50     accountHistoryDTO.setAccountId(bankAccount.getId());
51     accountHistoryDTO.setBalance(bankAccount.getBalance());
52     accountHistoryDTO.setCurrentPage(page);
53     accountHistoryDTO.setPageSize(size);
54     if (bankAccount instanceof SavingAccount) {
55         accountHistoryDTO.setType("Saving Account");
56     } else if (bankAccount instanceof CurrentAccount) {
57         accountHistoryDTO.setType("Current Account");
58     }
59     accountHistoryDTO.setTotalPage(accountOperationPage.getTotalPages());
60     return accountHistoryDTO;
61 }
62
63 @Override
64 public List<BankAccountDTO> getCustomerBankAccount(Long id) throws CustomerNotFoundException {
65     Customer customer = customerRepository.findById(id).orElseThrow(() -> new CustomerNotFoundException("Customer not Found"));
66     List<BankAccount> bankAccounts = bankAccountRepository.findByCustomerId(id);
67
68     return bankAccounts.stream().map(bankAccount -> {
69         if (bankAccount instanceof CurrentAccount) {
70             return dtoMapper.fromCurrentBankAccount((CurrentAccount) bankAccount);
71         } else return dtoMapper.fromSavingBankAccount((SavingAccount) bankAccount);
72     }).collect(Collectors.toList());
73 }
74
75 @Override
76 public List<CustomerDTO> searchCustomer(String keyword) {
77     List<Customer> customers = customerRepository.findByNameContains("%"+keyword+"%");
78     List<CustomerDTO> customerDTOS = customers.stream().map(customer -> dtoMapper.fromCustomer(customer)).collect(Collectors.toList());
79     return customerDTOS;
80 }

```

Partie 3 : Couche Web (RestController)

Dans cette partie, nous avons mis en place la couche Web de l'application en utilisant des RestControllers. Nous avons créé des RestControllers pour gérer les requêtes HTTP liées aux opérations sur les comptes bancaires et les clients. Nous avons défini des points de terminaison pour les différentes fonctionnalités, tels que la création d'un compte, la récupération des

informations d'un compte, l'exécution d'opérations sur un compte, etc. Les services de la couche précédente sont utilisés pour traiter les requêtes reçues, effectuer les opérations nécessaires et renvoyer les réponses appropriées.

```
1 package ma.enset.ebankingbackend.web;
2
3
4 import io.swagger.v3.oas.annotations.security.SecurityRequirement;
5 import lombok.AllArgsConstructor;
6 import lombok.extern.slf4j.Slf4j;
7 import ma.enset.ebankingbackend.dtos.*;
8 import ma.enset.ebankingbackend.exceptions.BalanceNotSufficientException;
9 import ma.enset.ebankingbackend.exceptions.BankAccountNotFoundException;
10 import ma.enset.ebankingbackend.services.BankAccountService;
11 import org.springframework.security.access.prepost.PreAuthorize;
12 import org.springframework.stereotype.Service;
13 import org.springframework.web.bind.annotation.*;
14
15 import java.util.List;
16
17
18 @RestController
19 @Slf4j
20 @RequestMapping("/bankAccount")
21 @CrossOrigin(value = "*", maxAge = 3600)
22 @SecurityRequirement(name = "digitalBankApi")
23 public class BankAccountRestAPI {
24     private BankAccountService bankAccountService;
25
26     public BankAccountRestAPI(BankAccountService bankAccountService) {
27         this.bankAccountService = bankAccountService;
28     }
29
30     @PreAuthorize("hasAuthority('USER')")
31     @GetMapping("/{find}/{accountId}")
32     public BankAccountDTO getBankAccount(String accountId) throws BankAccountNotFoundException {
33         return bankAccountService.getBankAccount(accountId);
34     }
35
36     @PreAuthorize("hasAuthority('USER')")
37     @GetMapping("/{findAll}")
38     public List<BankAccountDTO> getAllBankAccount(){
39         return bankAccountService.bankAccountList();
40     }
41
42     @PreAuthorize("hasAuthority('USER')")
43     @GetMapping("/{id}/operation")
44     public List<AccountOperationDTO> getHistory(@PathVariable("id") String accountId){
45         return bankAccountService.accountHistory(accountId);
46     }
47
48     @PreAuthorize("hasAuthority('USER')")
49     @GetMapping("/{id}/operationPage")
50     public AccountHistoryDTO getAccountHistory(@PathVariable("id") String accountId,
51                                                @RequestParam(name="page",defaultValue = "0") int page,
52                                                @RequestParam(name="size",defaultValue = "5") int size) throws BankAccountNotFoundException{
53         return bankAccountService.getAccountHistory(accountId,page,size);
54     }
55
56     @PreAuthorize("hasAuthority('USER')")
57     @PostMapping("/debit")
58     public DebitDTO debit(@RequestBody DebitDTO debitDTO) throws BalanceNotSufficientException, BankAccountNotFoundException {
59         bankAccountService.debit(debitDTO.getAccountID(),debitDTO.getAmount(), debitDTO.getDescription());
60         return debitDTO;
61     }
62
63     @PreAuthorize("hasAuthority('USER')")
64     @PostMapping("/credit")
65     public CreditDTO credit(@RequestBody CreditDTO creditDTO) throws BankAccountNotFoundException{
66         bankAccountService.credit(creditDTO.getAccountID(),creditDTO.getAmount(),creditDTO.getDescription());
67         return creditDTO;
68     }
69
70     @PreAuthorize("hasAuthority('USER')")
71     @PostMapping("/transfer")
72     public TransferDTO transefer(@RequestBody TransferDTO transferDTO) throws BankAccountNotFoundException, BalanceNotSufficientException
73     {
74         bankAccountService.transfer(transferDTO.getAccountSourceID(),transferDTO.getAccountDestinationID(), transferDTO.getAmount());
75         return transferDTO;
76     }
77 }
78
79
```



```

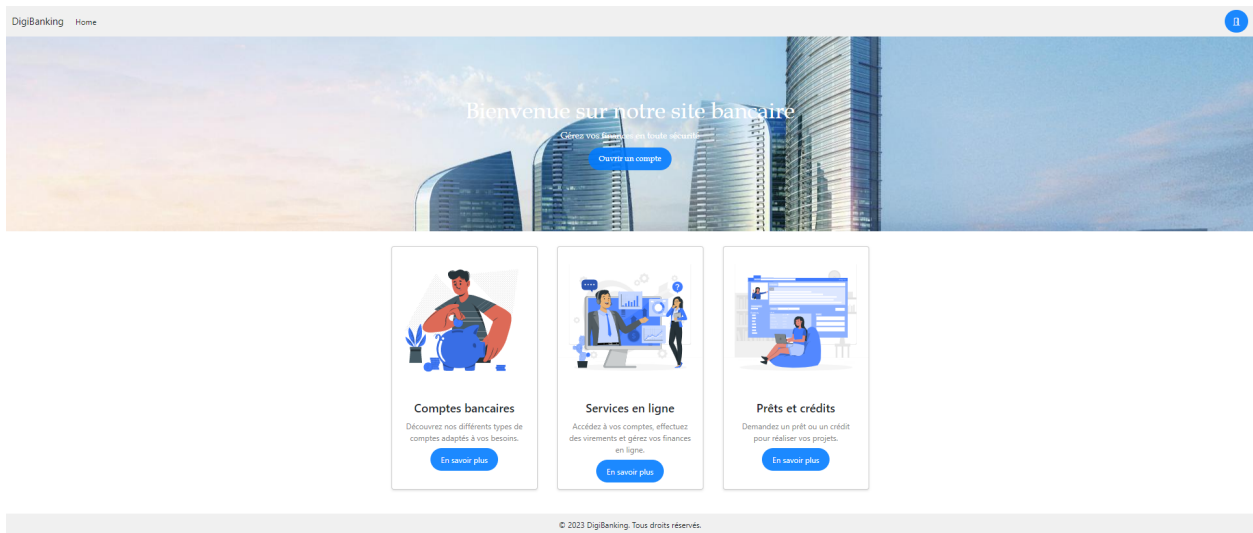
1 package ma.enset.ebankingbackend.web;
2
3
4 import io.swagger.v3.oas.annotations.security.SecurityRequirement;
5 import lombok.AllArgsConstructor;
6 import lombok.extern.slf4j.Slf4j;
7 import ma.enset.ebankingbackend.dtos.BankAccountDTO;
8 import ma.enset.ebankingbackend.dtos.CustomerDTO;
9 import ma.enset.ebankingbackend.exceptions.CustomerNotFoundException;
10 import ma.enset.ebankingbackend.services.BankAccountService;
11 import org.springframework.security.access.prepost.PreAuthorize;
12 import org.springframework.web.bind.annotation.*;
13
14 import java.util.List;
15
16 @RestController
17 @AllArgsConstructor
18 @Slf4j // les logs
19 @RequestMapping("/customer")
20 @SecurityRequirement(name = "digitalBankApi")
21 @CrossOrigin(value = "*", maxAge = 3600)
22 public class CustomerRestController {
23     private BankAccountService bankAccountService;
24     @GetMapping("/findAll")
25     public List<CustomerDTO> customerDTOS(){
26         return bankAccountService.listCustomer();
27     }
28     @GetMapping("/find/{id}")
29     public CustomerDTO getCustomer(@PathVariable("id") Long customerId) throws CustomerNotFoundException {
30         return bankAccountService.getCustomer(customerId);
31     }
32     @GetMapping("/search")
33     public List<CustomerDTO> searchCustomer(@RequestParam(name = "keyword", defaultValue = "")String keyword)
34     {
35         return bankAccountService.searchCustomer(keyword);
36     }
37     @PreAuthorize("hasAuthority('ADMIN')")
38     @PostMapping("/add")
39     public CustomerDTO saveCustomer(@RequestBody CustomerDTO customerDTO){
40         return bankAccountService.saveCustomer(customerDTO);
41     }
42     @PreAuthorize("hasAuthority('ADMIN')")
43     @PutMapping("/update/{id}")
44     public CustomerDTO updateCustomer(@PathVariable("id") Long customerId, @RequestBody CustomerDTO customerDTO) throws CustomerNotFoundException{
45         customerDTO.setId(customerId);
46         return bankAccountService.updateCustomer(customerDTO);
47     }
48     @PreAuthorize("hasAuthority('ADMIN')")
49     @DeleteMapping("/delete/{id}")
50     public void deleteCustomer(@PathVariable("id") Long customerId){
51         bankAccountService.deleteCustomer(customerId);
52     }
53 }
54
55 @GetMapping("/{id}/bankAccounts")
56 public List<BankAccountDTO> getBankAccounts(@PathVariable("id") Long id) throws CustomerNotFoundException{
57     return bankAccountService.getCustomerBankAccount(id);
58 }
59 }
60

```

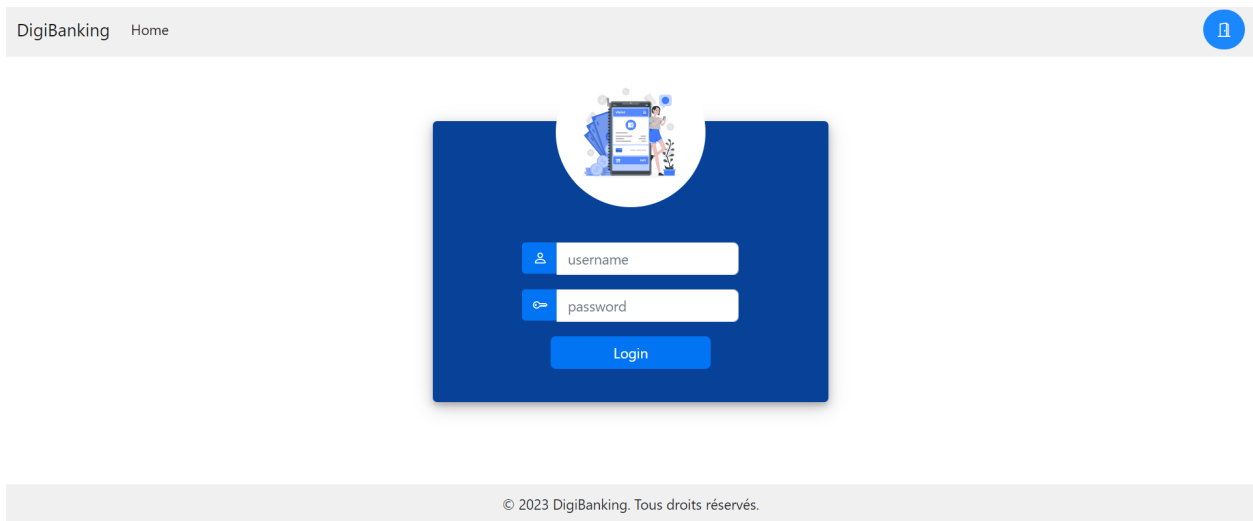
Partie 4 : Front End Angular

Dans cette partie, nous avons développé l'interface utilisateur de l'application en utilisant le framework Angular. Nous avons créé des composants Angular pour afficher les différentes fonctionnalités de l'application, tels que la liste des comptes, le formulaire de création d'un compte, la visualisation des détails d'un compte, etc. Nous avons utilisé des services Angular pour communiquer avec les RestControllers de l'application et effectuer les requêtes HTTP nécessaires. Les templates HTML et les styles CSS ont été conçus pour rendre l'interface utilisateur attrayante et conviviale.

Interface Home



Interface Login



Interface Account

DigiBanking

Home

Account

Customer

saad

Account

Account ID

Account Information

Account ID :6477447e-4713-4d64-b6e0-a7775470c903

Type :Current Account

Balance :151,055.48 MAD.

ID	Date & Heure	Type	Description	Amount
2328	29/05/2023 16:05	CREDIT	credit	34,999.00 MAD
2327	29/05/2023 16:05	DEBIT	kridi	19,786.00 MAD
1993	21/05/2023 13:05	CREDIT	CREDIT	11,096.79 MAD
1994	21/05/2023 13:05	DEBIT	DEBIT	11,186.17 MAD
1995	21/05/2023 13:05	CREDIT	CREDIT	10,259.60 MAD

Previous

1

2

3

4

5

6

7

Next

Operations

Debit

Credit

Transfer

Amount

0

Description

Save Operation

© 2023 DigiBanking. Tous droits réservés.

Interface Search Customer

User avec le rôle admin

DigiBanking

Home

Account

Customer









saad






Customers

KeyWord

Id	Name	E-mail	Action
1	Ahemed Saad	AhemedSaad@gmail.com	<div></div> <div>Accounts</div>
2	Saad	Saad@gmail.com	<div></div> <div>Accounts</div>
3	MohammedSaad	MohammedSaad@gmail.com	<div></div> <div>Accounts</div>
4	SokainaSaad	SokainaSaad@gmail.com	<div></div> <div>Accounts</div>
5	SaadBel	SaadBel@gmail.com	<div></div> <div>Accounts</div>

User avec le rôle user

Customers			
KeyWord			
Id	Name	E-mail	Action
1	Ahmed Saad	AhmedSaad@gmail.com	 Accounts
2	Saad	Saad@gmail.com	 Accounts
3	MohammedSaad	MohammedSaad@gmail.com	 Accounts
4	SokainaSaad	SokainaSaad@gmail.com	 Accounts
5	SaadBel	SaadBel@gmail.com	 Accounts
6	SokainaAhmed	SokainaAhmed@gmail.com	 Accounts
7	Mohammed	Mohammed@gmail.com	 Accounts

Customers			
KeyWord mohammed			
Id	Name	E-mail	Action
3	MohammedSaad	MohammedSaad@gmail.com	  Accounts
7	Mohammed	Mohammed@gmail.com	  Accounts

DigiBanking

Home

Account

Customer

saad

Customer Information

ID : 3

Name : MohammedSaad

Email : MohammedSaad@gmail.com

Accounts Information

Account Number	Account Type	Balance	
39ebc57a-e601-4e1b-b93a-279cf7b6ed07	CurrentAccount	82,425.48 MAD	
c33c9661-e968-47be-96de-9011db114a46	SavingAccount	152,771.54 MAD	
c6c3a208-79de-4807-8562-a63f7b1b6cb6	CurrentAccount	171,071.07 MAD	
db945849-7b62-4ed9-9b9c-abcd41bd51b1	SavingAccount	135,156.67 MAD	

Interface Add Customer

DigiBanking

Home

Account

Customer

saad

New customer

Name :

E-mail :

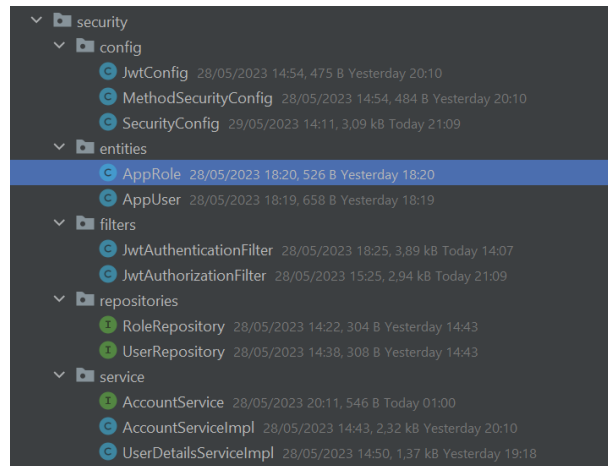
Save

© 2023 DigiBanking. Tous droits réservés.

Partie 5 : Sécurité avec Spring Security et JWT

Dans cette dernière partie, nous avons renforcé la sécurité de l'application en utilisant Spring Security et JWT (JSON Web Tokens). Nous avons configuré Spring Security pour protéger les points de terminaison de l'application, gérer l'authentification et l'autorisation des utilisateurs. Nous avons utilisé JWT comme mécanisme d'authentification stateless, stockant les

informations d'identification de l'utilisateur dans un jeton sécurisé. Des filtres de sécurité ont été mis en place pour valider les jetons JWT, vérifier les autorisations des utilisateurs et sécuriser les ressources de l'application.



Powered by KikiManjaro

```

1  package ma.enset.ebankingbackend.web;
2
3
4  import com.auth0.jwt.JWT;
5  import com.auth0.jwt.JWTVerifier;
6  import com.auth0.jwt.algorithms.Algorithm;
7  import com.auth0.jwt.interfaces.DecodedJWT;
8  import com.fasterxml.jackson.databind.ObjectMapper;
9  import io.swagger.v3.oas.annotations.security.SecurityRequirement;
10 import lombok.AllArgsConstructor;
11 import lombok.extern.slf4j.Slf4j;
12 import ma.enset.ebankingbackend.security.config.JwtConfig;
13 import ma.enset.ebankingbackend.security.entities.AppRole;
14 import ma.enset.ebankingbackend.security.entities.AppUser;
15 import ma.enset.ebankingbackend.security.service.AccountService;
16 import org.springframework.security.access.prepost.PreAuthorize;
17 import org.springframework.web.bind.annotation.CrossOrigin;
18 import org.springframework.web.bind.annotation.GetMapping;
19 import org.springframework.web.bind.annotation.RequestMapping;
20 import org.springframework.web.bind.annotation.RestController;
21
22 import javax.servlet.http.HttpServletRequest;
23 import javax.servlet.http.HttpServletResponse;
24 import java.security.Principal;
25 import java.util.Date;
26 import java.util.HashMap;
27 import java.util.Map;
28 import java.util.stream.Collectors;
29
30 @RestController
31 @Slf4j
32 @AllArgsConstructor
33 @CrossOrigin("")
34 @RequestMapping("/V1")
35 @SecurityRequirement(name = "digitalBankApi")
36 public class SecurityRestController {
37
38     private AccountService securityService;
39
40     @GetMapping("/refresh-token")
41     public void refreshToken(HttpServletRequest request, HttpServletResponse response) throws Exception {
42         String jwt_token = request.getHeader(JwtConfig.AUTHORIZATION_HEADER); // Authorization: Bearer xxx
43
44         if (jwt_token != null && jwt_token.startsWith(JwtConfig.TOKEN_HEADER_PREFIX)) { // Authorization: Bearer xxx
45
46             try {
47                 String jwt = jwt_token.substring(7);
48                 Algorithm algorithm = Algorithm.HMAC256(JwtConfig.SECRET_PHRASE);
49                 JWTVerifier verifier = JWT.require(algorithm).build();
50                 DecodedJWT decodedJWT = verifier.verify(jwt); // verify the token
51
52                 String username = decodedJWT.getSubject(); // get the username
53                 AppUser user = securityService.loadByUsername(username); // get the user using the username
54
55                 String jwtAccessToken = JWT.create() // create a new JWT with the username
56                     .withSubject(user.getUsername())
57                     .withExpiresAt(new Date(System.currentTimeMillis() + JwtConfig.ACCESS_TOKEN_EXPIRATION)) // 24hrs
58                     .withIssuer(request.getRequestURL().toString()) // url of the issuer
59                     .withClaim("roles", user.getRole().stream().map(AppRole::getRoleName).collect(Collectors.toList())) // roles
60                     .sign(algorithm); // secret
61
62                 Map<String, String> tokens = new HashMap<>(); // create a map with the tokens
63                 tokens.put("access_token", jwtAccessToken); // access token
64                 tokens.put("refresh_token", jwt); // refresh token
65                 response.setContentType("application/json"); // set the content type
66                 new ObjectMapper().writeValue(response.getOutputStream(), tokens);
67
68             } catch (Exception e) {
69                 throw e;
70             }
71         } else {
72             throw new RuntimeException(" You should refresh your token ");
73         }
74     }
75
76     @PreAuthorize("hasAuthority('USER')") // hasAuthority('USER')
77     @GetMapping("/profile")
78     public AppUser getUser(Principal principal) { // Principal is the user
79         return securityService.loadByUsername(principal.getName());
80     }
81 }
82

```



Conclusion

En conclusion, nous avons réussi à développer une application de gestion des comptes bancaires complète, en suivant les différentes étapes du processus de développement. La couche DAO a été mise en place pour gérer l'accès et la manipulation des données dans la base de données. La couche services, DTO et mappers a été utilisée pour gérer la logique métier et faciliter l'échange de données entre les différentes couches. La couche Web avec les RestControllers a permis de gérer les requêtes HTTP et de fournir les fonctionnalités de l'application aux utilisateurs. Le front-end Angular a fourni une interface utilisateur attrayante et conviviale. Enfin, la sécurité a été renforcée avec Spring Security et JWT pour protéger les données et les ressources de l'application.

L'application de gestion des comptes bancaires offre une solution complète pour les clients et les opérations bancaires, en prenant en compte les différents types de comptes et les opérations de débit et de crédit. Elle offre une expérience utilisateur fluide et sécurisée.