Online Banking Application

Problem Statement:

Classroom

Mini Project: Secure Online Banking Application
Secure Online Banking Application

Problem | Submissions | Solutions | Doubts

Secure Online Banking Application

Hard • Score 360/360 Spring Hibernate

Problem statement

Send feedback

Project Goal

The organization requires a Bank Security Application that allows users to access banking services. The application should provide account creation, card selection, and investment features. It should have an admin panel that can be accessed to view user and account information only for users with admin roles.

▼ Features of the Application:

The application should allow users to:

- Register themselves into the application.
- Create an account of your choice
- Save Nominee for your respective account.
- Retrieve details for the granted card.
- Invest Money through the bank account.

▼ Steps:

Use the following guidelines and hints to build the project.

1. We will be utilizing the Spring version **2.7.16** for this mini-project.

2. Create entities for Account, AccountType (enum), BranchType (enum), CardType (enum), InvestmentType (enum), Card, Investment, Nominee, Role, and User.

3. Below is an example of how to create an enum AccountType and use it in the Account class:

▼ Steps:

Use the following guidelines and hints to build the project.

1. We will be utilizing the Spring version **2.7.16** for this mini-project.

2. Create entities for Account, AccountType (enum), BranchType (enum), CardType (enum), InvestmentType (enum), Card, Investment, Nominee, Role, and User.

3. Below is an example of how to create an enum AccountType and use it in the Account class:

▼ Reference image

```
public enum AccountType {
    SAVINGS,
    CURRENT,
    PPF,
    SALARY
}
```

▼ Reference image

```
@Enumerated(EnumType.STRING)
private AccountType accountType;
```

4. Using a similar approach as shown above create the following enum with the required attribute:

▼ BranchType

```
BranchType: SBI, ICIC, BOB, HDFC
```

▼ CardType

```
CardType:  DEBIT_CLASSIC, DEBIT_GLOBAL, CREDIT_PREMIUM, CREDIT_MASTER
```

▼ InvestmentType

```
InvestmentType: GOLD, STOCKS, MUTUAL_FUND, FIXED_DEPOSITS
```

Problem Submissions Solutions Doubts

5. Create the following classes in the entity package with the given attributes and required annotatio

a. Account class

- Long id
- AccountType accountType (@Enumerated(EnumType.STRING))
- String status
- double balance
- float interestRate
- BranchType branch (@Enumerated(EnumType.STRING))
- String proof
- Date openingDate
- Long accountNumber
- Nominee nominee (One To One Mapping with Nominee entity)
- Card card (One To One Mapping with Card entity)
- User user (Many To One Mapping with User entity)

b. Card class

- Long id
- Long cardNumber
- String cardHolderName
- CardType cardType (@Enumerated(EnumType.STRING))
- double dailyLimit
- int cvv
- Date allocationDate

- Date allocationDate
- Date expiryDate
- Long pin
- String status

c. Investment class

- Long id
- InvestmentType investmentType (@Enumerated(EnumType.STRING))
- String risk
- double amount
- float returns
- String duration
- String companyName
- User user (Many To One Mapping with User entity)

d. Nominee class

- Long id
- String relation
- String name
- Long accountNumber
- String gender
- int age

e. Role class

- Long id
- String roleName

f. User class: It should implement the UserDetails interface

- Long id
- String name
- String username
- String password
- String address
- Long number
- String identityProof
- Role roles (Many To One Mapping with Role entity)
- List accountList = new ArrayList<>() (One To Many Mapping with Account entity)
- List investmentList = new ArrayList<>() (One To Many Mapping with Investment entity)

6. Create the following classes in the dto package with the given attributes and required annotations:

a. AccountDto class

- String accountType
- double balance
- String proof
- Nominee nominee

b. AdminDto class

- String name
- String username
- String password
- String address
- Long number
- String identityProof

c. CardDto class

- String cardHolderName
- String cardType
- Long pin

d. InvestmentDto class

- String investmentType
- double amount
- String duration

e. KycDto class

- String name
- String address
- Long number
- String identityProof

f. NomineeDto class

- String relation
- String name
- Long accountNumber
- String gender
- int age

g. UserDto class

- String name
- String username
- String password
- String address
- Long number
- String identityProof
- List accountList = new ArrayList<>()
- List investmentList = new ArrayList<>()

7. Create repository interfaces for the following entity with the methods shown below to handle database operations using Spring Data JPA:

a. Account Entity

- Optional findByAccountNumber(Long accountNumber): Derived Query to fetch all accounts by accountNumber.
- List findAllActiveAccounts(): JPQL for fetching active accounts from the database.
- List findAllInActiveAccounts(): JPQL for fetching inactive accounts from the database.
- List findAllByAccountType(AccountType accountType): JPQL for fetching accounts by the given accountType from the database.
- List findAllByBranch(BranchType branchType): JPQL for fetching accounts by the given branchType from the database.

b. Card Repository

- Optional findByCardNumber(Long cardNumber): Derived Query to fetch Card by cardNumber.

c. Investment Repository

d. Nominee Repository

e. User Repository

- Optional findByUsername(String username): Derived Query to fetch User by username.

8. Implement spring security using the below logic:

a. Create two DTO classes: JwtRequest and JwtResponse. JwtRequest will encompass a (String) username and (String) password, while JwtResponse will solely contain the jwtToken of type string.

b. Additionally, you'll need to create JwtAuthenticationFilter and JwtAuthenticationHelper classes for the implementation.

c. Create a CustomUserDetailService that implements UserDetailsService and overrides the loadUserByUsername method.

e. Lastly, in the service layer, implement the login and doAuthenticate methods as taught in previous lectures.

9. Generate a security configuration containing beans for authenticationManager and passwordEncoder.

10. Expose only the "/user/register" and "/user/login" endpoints through antMatchers.

11. Develop a service interface and corresponding implementation for each entity to manage business logic.

12. Implement the necessary CRUD operations for managing users, accounts, cards, and investments.

13. Test the application using tools such as Postman to ensure data is saved and retrieved correctly from the database.

Problem Submissions Solutions Doubts

▼ End Points To Be Created:

1. UserController Endpoints:

- POST "/user/login" (Body JwtRequest jwtRequest): It logins the user after authenticating it through authService and assigns a JWT token.

- POST "/user/register" (Body UserDto user): It registers a new User and by default assigns it ROLE _ CUSTOMER, the password is encoded using BCryptPasswordEncoder.

2. UserAccountController Endpoints:

- POST "/account/create/{userId}" (Body AccountDto accountDto,@PathVariable Long userId): An account will be created for the user with the unique account number and its status will be set as "ACTIVE". To assign card and account parameters, please adhere to the criteria corresponding to the respective account type. Upon card assignment, the card number and CVV will be randomly generated where the current date will be used as the allocation date and the expiry date will be set 5 years ahead.

▼ Reference image

```
"SAVINGS":{
            CardType -> CardType.DEBIT_GLOBAL
            DailyLimit -> 40000
            AccountType -> AccountType.SAVINGS
            InterestRate -> 2.70
            Branch -> BranchType.BOB
            }

"CURRENT":{
            CardType -> CardType.CREDIT_PREMIUM
            DailyLimit -> 50000
            AccountType -> AccountType.CURRENT
            InterestRate -> 5.2
            Branch -> BranchType.ICIC
            }
```

```
    "PPF":{
            AccountType -> AccountType.PPF
            InterestRate -> 7.4
            Branch -> BranchType.SBI
            }

  "SALARY":{
            CardType -> CardType.CREDIT_MASTER
            DailyLimit -> 75000
            AccountType -> AccountType.SALARY
            InterestRate -> 4.1
            Branch -> BranchType.HDFC
            }
```

- GET "/account/all/{userId} (@PathVariable Long userId)": It fetches the list of all the accounts associated with the given user.

- GET "/account/balance" (@RequestParam Long accountNumber): It fetches the account balance for the given account number.

- GET "/account/nominee" (@RequestParam Long accountNumber): It fetches the nominee based on the given accountNumber.

- PUT "/account/updateNominee/{accountId}" (@RequestBody NomineeDto nomineeDto, @PathVariable Long accountId): It Updates nominee for the given accountId.

- GET "/account/getKycDetails (@RequestParam Long accountNumber)": It fetches the User through the given accountNumber. While returning the user set AccountList and InvestmentList to null.

- PUT "/account/updateKyc/{accountId} (@RequestBody KycDto kycDto,@PathVariable Long accountId)": It fetches the User through the given accountNumber and while returning the user it sets the AccountList and InvestmentList to null.

- GET "/account/getAccount/summary (@RequestParam Long accountNumber)": It fetches the Account through the given accountNumber and while returning the account it sets the user to null.

3. UserCardController Endpoints:

- GET "/card/block" (@RequestParam Long accountNumber,@RequestParam Long cardNumber): It deletes the card that is linked to the specified account number.

Problem Submissions Solutions Doubts

3. UserCardController Endpoints:

- GET "/card/block" (@RequestParam Long accountNumber,@RequestParam Long cardNumber): It deletes the card that is linked to the specified account number.

- POST "/card/apply/new" (@RequestParam Long accountNumber, @RequestBody CardDto cardDto): It creates a new card based on the below criteria for the given accountNumber. Also while creating a new card it checks if any card is already assigned to the given accountNumber if a card is found the execution of the program stops.

▶ **Reference image**

- PUT "/card/setting" (@RequestBody Card card, @RequestParam Long cardNumber): It updates the card limit and PIN for the given cardNumber. You need to follow the below criteria for updating the card limit based on the card type.

▶ **Reference image**

4. UserInvestmentController Endpoints:

- POST "/invest/now" (@RequestParam Long accountId, @RequestBody InvestmentDto investmentDto): It creates an investment associated with the given account ID. Also while investing the API confirms if the given amount of investment is valid by checking the balance of the given account.

5. AdminController Endpoints:

- POST "/admin/add" (Body AdminDto admin): It registers a new User and by default assigns it ROLE _ ADMIN, the password is encoded using BCryptPasswordEncoder.

- GET "/admin/getAllUser": It fetches the list of all users present in the database.

- GET "/admin/getUserByName/{username}" (@PathVariable String username): It fetches a user by the given name.

- DELETE "/admin/deleteUser/{userId}" (@PathVariable Long userId): It deletes the given user.

- PUT "/admin/account/deactivate" (@RequestParam Long userId,@RequestParam Long accountId): It first confirms the existence of the user and account and then modifies the account status to INACTIVE for the given accountId.

- PUT "/admin/account/activate" (@RequestParam Long userId,@RequestParam Long accountId): It first confirms the existence of the user and account and then modifies the account status to ACTIVE for the given accountId.

- GET "/admin/account/getActiveAccountsList": It fetches the list of Active accounts from the database.

- GET "/admin/account/getInActiveAccountsList": It fetches the list of INACTIVE accounts from the database.

- GET "/admin/account/getActiveAccountsList" (@PathVariable AccountType accType): It fetches the list of accounts by accType from the database.

- GET "/admin/account/getInActiveAccountsList" (@PathVariable BranchType branchType): It fetches the list of accounts by branchType

- DELETE "/admin/deleteUser/{userId}" (@PathVariable Long userId): It deletes the given user.

- PUT "/admin/account/deactivate" (@RequestParam Long userId,@RequestParam Long accountId): It first confirms the existence of the user and account and then modifies the account status to INACTIVE for the given accountId.

- PUT "/admin/account/activate" (@RequestParam Long userId,@RequestParam Long accountId): It first confirms the existence of the user and account and then modifies the account status to ACTIVE for the given accountId.

- GET "/admin/account/getActiveAccountsList": It fetches the list of Active accounts from the database.

- GET "/admin/account/getInActiveAccountsList": It fetches the list of INACTIVE accounts from the database.

- GET "/admin/account/getActiveAccountsList" (@PathVariable AccountType accType): It fetches the list of accounts by accType from the database.

- GET "/admin/account/getInActiveAccountsList" (@PathVariable BranchType branchType): It fetches the list of accounts by branchType from the database.

▼ **Testing on Postman:**

After successfully creating the application, you need to test its functionality. Your application should be tested for the following scenarios:

- Registering a user: The application should store the user details.

- Account Opening: Opening an account for the existing user.

- Investment: The application should fetch the account and confirm the balance before investing money.

- Retrieving Account Details: The application should retrieve the account details.

- Blocking Card: The application should block the card associated with the user account.

- Test all the exposed APIs of the backend on Postman to ensure proper functionality and data handling.

▼ Reference image

Output:

http://localhost:8080/user/register

POST http://localhost:8080/user/register

Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
{
    "name": "mohit",
    "username": "Mohit107",
    "password": "Mohit1234@",
    "address": "Erick Wave",
    "number": 9988997766,
    "identityProof": "addharCard"
}
```

Body Cookies Headers (10) Test Results Status: 201 Created Time: 422 ms Size: 310 B Save as example

Pretty Raw Preview Visualize Text

```
1
```

http://localhost:8080/user/login

POST http://localhost:8080/user/login

Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
{
    "username": "Mohit107",
    "password": "Mohit1234@"
}
```

Body Cookies Headers (11) Test Results Status: 200 OK Time: 276 ms Size: 539 B Save as example

Pretty Raw Preview Visualize JSON

```
{
    "jwtToken": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJNb2hpdDEwNyIsImlhdCI6MTcwMjI3NjMyOSwiZXhwIjoxNzAyMjc5OTI5fQ.QhpNB3tABfbODfurhszeh8jNeTMQ8QMIuX-Wp2iJ9
}
```

http://localhost:8080/account/create/1

POST http://localhost:8080/account/create/1

Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
{
    "accountType": "SALARY",
    "proof": "Pan Card",
    "balance": 80000,
    "nominee": {
        "relation": "Father",
        "name": "Mazra Tehr",
        "accountNumber": 2233446127128,
        "gender": "Male",
        "age": 38
    }
}
```

Body Cookies Headers Test Results Status: 201 Created Time: 185 ms Size: 310 B Save as example

Pretty Raw Preview Visualize Text

```
1
```