**Digital Banking Web Application - InfyMeMobile**

**Problem Statement:**

InfyMeMobile is a digital banking application and, one of the leading digital banking solutions in India. It allows its users to access a wide range of digital banking services conveniently from their smartphones. So, the user can use this app to link their different bank accounts such that they can see the balance, and transfer Funds as well. They can also see the Account Statement for this digital banking application.

By using the InfyMeMobile app user can register/signup with the InfyMeMobile by using their mobile number and other details. After signing up the user must be able to view all the bank account(s) linked to the user’s mobile number.

If the account is/are not linked, then the user must be able to link his/her account to his/her mobile number after a successful OTP verification.

Once the bank account is linked with the user’s mobile number (after the OTP verification) the user must be able to do various operations like – Checking Account Balance, Fund Transfer, and Account Statement.

Similarly, if the user has multiple accounts in different banks, then he/she must be able to link all the accounts to the same mobile number with an OTP verification in the InfyMeMobile

**[Note: You can think of the functionality of PAYTM/ GPAY / PHONEPE, when we are installing it for the first time, it will ask for your primary mobile number. And once you selected your primary mobile number, automatically it will display all the bank accounts associated with it. Then you can select the appropriate bank to link to your PAYTM/ GPAY / PHONEPE UPI. Maybe during linking they use OTP/GRID numbers on Debit cards/others. Once the bank account added to the PAYTM/ GPAY / PHONEPE you can transfer Funds/Check Balance and see account details also.]**

Below are the functionalities to be implemented:

• Register/Sign Up

• Login (Using OTP / Using Password)

• View Accounts

• Link Account(s)

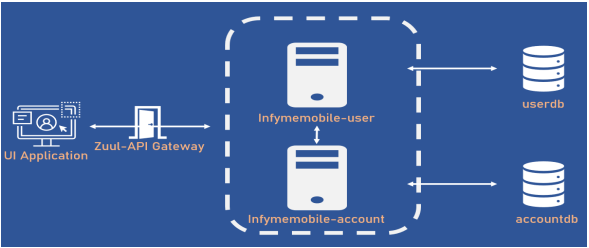
• Link Account With OTP(Optional)

• Check Balance

• Send Money/Fund Transfer

• Account Statement

**Application Architecture:**

Spring Cloud API Gateway, Configuration Server for centralised configuration

Netflix Eureka Server for Service Registry and Discovery

InfyMeMobile application should be developed as a full-stack application with the following specification:

Frontend: Angular

Backend: Java, Spring Boot and Spring REST

Database: MySQL

**In this Capstone Project, you need to implement the backend using Spring REST.**

These must be two Spring REST Services with various endpoints to serve the above functionalities:

**1. Infymemobile-user: Infymemobile-user deals with only User functionalities like Signup and Login.**

**2. Infymemobile-account: Infymemobile-account deals with only Account functionalities like- Displaying the list of accounts for a mobile number, Link Accounts, Checking Balance, Fund Transfer, Account statement, etc.**

In the first phase, you can implement both the Spring REST Services as independent Spring REST projects with appropriate endpoints to serve the above functionalities.

At any point, if these two Services need to be connected to exchange data, use the RestTemplate class to join the appropriate end points.

Write the appropriate configurations in the application.properties files of these two Spring REST applications to auto-generate the table scripts based on the Entity classes created and use appropriate DML statements to insert the sample data. You can also create the appropriate end points with POST requests to insert the data as well.

**Further Enhancements:**

* Later, in the second phase you can configure the previous two services with the Spring Cloud API Gateway server.
* Use Swagger for documenting RESTful web service
* Configuration server for centralized configuration
* Implement service registry and discovery pattern- Service Discovery Pattern is a mechanism for that enables the clients of service to make requests to a dynamically changing set of ephemeral service instances as port number & IP addresses of service keeps changing in case of Cloud-native application.

It helps the client of a service -

1. for eg: the API gateway or another service - discover the location of a service instance.
2. the client obtains the location of a service instance by querying a Service Registry, which knows the locations of all service instances.

* Try to implement the UI layer of the application using your favourite UI technology.

**InfyMeMobile Database Structures:**

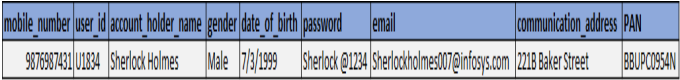
**Note: The ‘Datatype’ column for all the tables is given as Java Datatypes. So, if you are creating the tables manually, kindly use the appropriate datatypes for the respective relational database.**

**User table:**

The User Table will store the User details.

|  |  |  |  |
| --- | --- | --- | --- |
| **Column** | **Datatype** | **Description** | **Constraints** |
| mobile\_number | Long | Mobile Number of User | Primary Key |
| user\_id | String | Unique ID of User | Unique |
| account\_holder\_name | String | Name of User | Not Null |
| gender | String | Gender of User | Not Null |
| date\_of\_birth | LocalDate | DOB of User | Not Null |
| password | String | Password for Login of User | Not Null |
| email | String | Email of User | Not Null |
| communication\_address | String | Communication address of User | Not Null |
| PAN | String | Mobile Number of User | Not Null |

Sample Data:



**BankAccount table:**

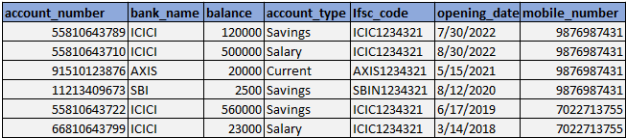
This table stores a single or different bank account detail associated with a mobile number. Here we have taken the account number as the primary key and the mobile number as a foreign key which is referring to the mobile number of the user table. So, multiple bank account details can be mapped to a single mobile number.

**[Note: In real-life applications like Paytm/ PhonePe / GPay the respective bank account details are getting fetched from its respective bank’s database with the help of the mobile number associated with the bank account. But, in our scenario instead of creating different bank account tables for different banks, we have created a single table containing multiple bank account details.]**

|  |  |  |  |
| --- | --- | --- | --- |
| **Column** | **Datatype** | **Description** | **Constraints** |
| account\_number | Long | Account Number | Primary Key |
| bank\_name | String | Name of the Bank | Not Null |
| balance | Double | Balance Available | Not Null |
| account\_type | String | Type of Account | Not Null |
| ifsc\_code | String | IFSC code of the Branch | Not Null |
| opening\_date | LocalDate | Opening date of the Account | Not Null |
| mobile\_number | Long | Which the account/mobile number is mapped to | Not Null, FK |

Sample Data:

So, as per the Sample Data, you can see 4 different accounts (maybe from the same bank or different banks) have been mapped to the mobile number – 9876987431

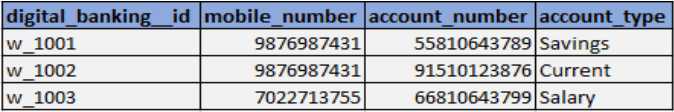


**DigitalBankAccount table:**

When the user will link the bank account with the InfyMeMobile application the details will be stored in the DigitalBankAccount table as below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Column** | **Datatype** | **Description** | **Constraints** |
| digital\_banking\_\_id | String | Wallet Id of the User | Primary Key |
| mobile\_number | Long | Mobile Number for the Digital Bank | Not Null, FK |
| account\_number | Long | Exact account number associated with | Not Null, FK |
| account\_type | String | Type of Account | Not Null |

Sample Data:

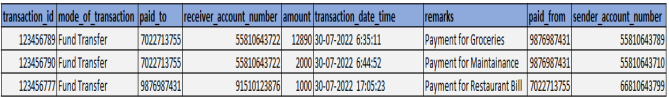


**Transaction table:**

This table will store all the transactions made by the users through the InfyMeMobile application. Note: The column ‘mode\_of\_transaction’ may be – ‘Fund Transfer’ / ‘Electricity Bill Pay’ / ‘DTH Recharge’ / ‘QR Scan payment’. So, just to simplify the implementation we have kept all the ‘mode\_of\_transaction’ as ‘Fund Transfer’ so that while implementing the Account Statement functionality for a mobile number it will be easy to implement.

|  |  |  |  |
| --- | --- | --- | --- |
| **Column** | **Datatype** | **Description** | **Constraints** |
| transaction\_id | Integer | Unique Transaction ID | Primary Key |
| mode\_of\_transaction | String | Mode of Transaction(Always ‘Fund Transfer’) | Not Null |
| paid\_to | Long | The mobile number to which the amount is paid | Not Null |
| receiver\_account\_number | Long | Respective Account number of the receiver which is linked to the above Mobile Number(paid\_to) to which the amount is paid | Not Null |
| amount | Double | Actual amount transferred | Not Null |
| transaction\_date\_time | LocalDateTime | Date and Time of the Transaction made | Not Null |
| remarks | String | Custom comments for Transaction | Not Null |
| paid\_from | Long | The mobile number from which the amount is paid | Not Null |
| sender\_account\_number | Long | Respective Account number of the sender which is linked to the above Mobile Number(paid\_from) from which the amount is paid | Not Null |

Sample Data:



**Note:**

* **Sample Data Structures and Sample Data are given. If you feel that as per the requirement, you need more/fewer columns/data, you can proceed with that too.**
* **And create both the tables in a single database only as of now so that mapping them will be easier.**

**1)infymemobile-user:**

**REST Controller Layer:**

• Create a class called UserApi.

• Add annotation to create the below classes as REST controller classes and add cross-origin resource-sharing annotation.

• Autowire required service and logger classes.

• Log the name of the methods using Logger.

• Create methods to provide the below functionality.

**UserApi.java:**

This is a REST controller class and is used to map APIs for sign-up and login for a User, view profiles, and display all the Users.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Method** | **Request Type** | **API** | **Path Variable** | **Request Body** | **Response** |
| **createUser(UserDTO userDTO)** | **POST** | **/users** | **-** | **UserDTO userDTO** | **ResponseEntity <String>** |
| **loginUser(LoginDTO loginDTO)** | **POST** | **/users/login** | **-** | **LoginDTO loginDTO** | **ResponseEntity<Boolean>** |
| **getUserProfile(String userId)** | **GET** | **/users/{userId}** | **userId** | **-** | **ResponseEntity<UserDTO>** |
| **showAllUsers()** | **GET** | **/users/all** | **-** | **-** | **ResponseEntity<List<UserDTO>>** |

**Service Layer:**

**•** Create an interface called UserService and add the below methods as abstract methods.

• Create a class UserServiceImpl which will implement the UserService interface.

• Add annotation to mark it as a service class in the Spring IoC container.

• Autowire required repository interfaces and logger class.

• Log the name of the method using Logger.

• Implement the below methods as per the descriptions to provide the below functionalities.

• You can improve the functionalities based on your needs as well.

**UserServiceImpl.java:**

This is a service implementation class that provides functionality for signing up and logging in users, viewing user profiles, and displaying all users.

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Input Parameters** | **Return Type** | **Description** |
| createUser(UserDTO userDTO) | UserDTO userDTO | String | This method is used to create a new User. Invoke the save() method of Repository and return Mobile Number. Handle Exceptions if any. Before saving the User you can also check whether that user is already present or not. |
| loginUser(LoginDTO loginDTO) | LoginDTO loginDTO | Boolean | This method is used to log in with an existing Mobile number and password. Invoke findById() method of Repository. If the mobile number is available, then compare passwords and return the result otherwise throw InfyMeMobileException with the message AUTHENTICATION\_FAILED. |
| getUserProfile(String userId) | String userId | UserDTO | This method is used to send UserDTO for a given user id. Invoke findById() method of Repository. If the user is available, then send userDTO otherwise throw InfyMeMobileException with the message USERID\_NOT\_FOUND. |
| showAllUsers() | - | List<UserDTO> | This method is used to send a list of all UserDTOs. Invoke findAll() method of Repository. If no users are found in the database throw InfyMeMobileException with the message NO\_USERS\_FOUND.is |

**Repository Layer:**

• Create an interface called UserRepository and make it a Spring Data JPA Repository wherever possible.

• Refer to the Service class- UserServiceImpl.java to implement the UserRepository.

• Create queries using method names if required.

• Use @Query annotation if required.

• Use @NamedQuery in Entity classes if required and reuse those methods at the Repository layer if needed.

• Note: The repository layer should not contain any Implementation classes. Only make usage of Query creation based on method names OR @Query annotation OR @NamedQuery annotations with appropriate JPQLs

**Entity and DTO Layer:**

**Entity Class:**

• Create a class called UserEntity.

• Annotate the class with required annotations to make it an Entity class.

• Refer to the below instances.

**UserEntity.java:**

This is an entity class for the User and should be mapped with the User table. Create the below variables in this class.

|  |  |
| --- | --- |
| **Datatype** | **Variable** |
| Long | mobileNumber |
| String | userId |
| String | accountHolerName |
| String | gender |
| LocalDate | dateOfBirth |
| String | password |
| String | email |
| String | communicationAddress |
| String | PAN |

Make mobileNumber as primary key and use the appropriate annotations to map the variables to the columns if needed.

**DTO Class:**

• Create a class called UserDTO.

• Create the below variables in DTO classes and add validations with custom error messages.

**UserDTO.java:**

• This is a DTO class for Users.

• Create the below variables in this class with given validations

|  |  |  |
| --- | --- | --- |
| **Datatype** | **Variable** | **Validation** |
| Long | mobileNumber | Cannot be null, the minimum length should be 10, and the maximum length should be 10. |
| String | userId | Must start with ‘U’. |
| String | accountHolerName | Cannot be null, the minimum length should be 3, and the maximum length should be 50. |
| String | gender | Must be Male or Female. |
| LocalDate | dateOfBirth | Must be a Past Date. (If you want you can also add the validation for age must be greater than 18) |
| String | password | Cannot be null, the minimum length should be 5, and the maximum length should be 10. And it must be alpha-numeric. |
| String | email | Must contain at least one .[dot] Or \_[underscore] and one ‘@’. |
| String | communicationAddress | Cannot be null, the minimum length should be 3, and the the maximum length should be 50. |
| String | PAN | Cannot be null, and the length should be 10. Must follow the pattern of 5 capital characters followed by 4 digits followed by 1 capital character. |

**[Note: Sample implementations for each layer have been given as per the requirement. If you feel that you need more implementations, you can proceed and add them too.]**

**2)infymemobile-account:**

**REST Controller Layer:**

• Create a class with name AccountApi.

• Add annotation to create the below classes as REST controller classes and add cross-origin resource-sharing annotation.

• Autowire required service and logger classes.

• Log the name of the methods using Logger.

• Create methods to provide the below functionality.

**AccountApi.java:**

This is a REST controller class and is used to map URIs for account-related operations like – showing all accounts for a registered mobile number, showing balance, linking accounts, linking accounts with OTP, Fund Transfer, and account statements.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Method** | **Request Type** | **API** | **Query Parameter** | **Path Variable** | Request Body | Response |
| createAccount (AccountDTO accountDTO) | POST | /accounts | - | - | AccountDTO accountDTO | ResponseEntity <String> |
| listAccounts(Long mobileNo) | GET | /accounts/ {mobileNo} | - | mobileNo | - | ResponseEntity<List<BankAccountDTO>> |
| linkAccount(Long mobileNo, Long accountNo) | POST | /accounts/ {mobileNo} | - | mobileNo | accountNo | ResponseEntity<String> |
| linkAccount(Long mobileNo, Long accountNo, Integer otp) | POST | /accounts/{mobileNo} | - | mobileNo | accountNo,otp | ResponseEntity<String> |
| checkBalance(Long mobileNo, Long accountNo) | GET | /accounts/ balance/ {mobileNo}/ | accountNo | mobileNo | - | ResponseEntity<Double> |
| fundTransfer(TransactionDTO transactionDTO) | PUT/PATCH | /accounts/ fundtransfer/ | - | - | TransactionDTO transactionDTO | ResponseEntity<String> |
| accountStatement(Long mobileNo) | GET | /accounts/ statement/{ mobileNo} | - | mobileNo | - | ResponseEntity<List<TransactionDTO>> |

**Note: For sending any request in the form of a Request Body, if you need an extra DTO class you can create it on your own.**

**Service Layer:**

• Create an interface called AccountService, BankAccountService, DigitalBankAccountService, and TransactionService and add the respective methods as abstract methods in the respective interfaces based on the functionalities.

• Create classes AccountServiceImpl, BankAccountServiceImpl, DigitalBankAccountServiceImpl, and, TransactionServiceImpl which will implement the respective interfaces.

• Add annotation to mark these as service classes in Spring IoC container.

• Autowire required repositories interfaces and logger class.

• Log the name of the method using Logger.

• Implement the below methods as per the descriptions to provide the below functionalities.

• You can improve the functionalities based on your needs as well.

Implement the below functionalities in the respective ServiceImpl classes for displaying a list of accounts for a mobile number, link accounts, link accounts using OTP, checking bank balance, fund transfer, account statements, etc.

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Input Parameters** | **Return Type** | **Description** |
| createAccount(AccountDTO accountDTO) | AccountDTO accountDTO | String | This method is used to create a new Account in the BankAccount table. Invoke the save() method of Repository and return the Account Number. Handle Exceptions if any. Account numbers can be auto-generatedare  to automate the process. |
| listAccounts(Long mobileNo) | Long mobileNo | List<BankAccountDTO> | This method is used to return a list of all BankAccountDTO. Invoke findAll() method of Repository. If no accounts are found in the database for a mobile number throw InfyMeMobileException with the message NO\_ACCOUNTS\_FOUND. |
| linkAccount(Long mobileNo, Long accountNo) | Long mobileNo, Long accountNo | String | This method is used to link a mobile number and respective bank account to a digital\_banking\_\_id without any OTP verification. Check whether for the given mobile number one or more bank account is present or not by calling the appropriate method of Repository. If no accounts are found in the database for a mobile number throw InfyMeMobileException with the message NO\_ACCOUNTS\_FOUND. If all details are valid then the bank account must be linked to the mobile number and details must be persisted in the DigitalBankAccount table. |
| linkAccount(Long mobileNo, Long accountNo, Integer OTP) | Long mobileNo, Long accountNo, Integer OTP | String | This method is used to link a mobile number and respective bank account to a digital\_banking\_\_id after an OTP verification. To simplify this process, we have created a Utility class for generating the OTP which always returns a hardcoded value as OTP. Check whether, for a particular mobile number, one or more bank account is present or not by calling the appropriate method of Repository. If no accounts are found in the database for a mobile number throw InfyMeMobileException with the message NO\_ACCOUNTS\_FOUND. If all the details are valid then the bank account must be linked to the mobile number and details must be persisted in the DigitalBankAccount table after validating the OTP value coming as a parameter with the OTP generated by the application. Check below for the OTP generation utility class for more information. If there is a mismatch in OTP then throw InfyMeMobileException with message OTP\_DOESNOT\_MATCH. |
| checkBalance(Long mobileNo, Long accountNo) | Long mobileNo, Long accountNo | Double | This method is used to return the account balance for a given mobile number and account number if they are linked in the InfyMeMobile App i.e., if the details are present in the DigitalBankAccount table. Invoke the appropriate method of Repository to check whether the mobile number is present in the DigitalBankAccount table. If a mobile number is available, then return the balance from BankAccount Table otherwise throw InfyMeMobileException with the message NO\_ACCOUNT\_IS\_LINKED. |
| fundTransfer(TransactionDTO transactionDTO) | TransactionDTO transactionDTO | String | This method is used to transfer funds from one mobile number to another. Invoke the appropriate method of Repository to check whether the Funds that need to be transferred are less than the account balance or not, If Yes then the amount will be debited from the Sender’s Account and must be added to the Receiver’s Account and the Transaction Table must be updated accordingly else throw InfyMeMobileException with message INSUFFICIENT\_FUNDS.  Note: While transferring the Fund the Sender and receiver may have multiple bank accounts linked to the same mobile number in the InfyMeMobile App. So while implementing the fundTransfer() take of this scenario from which account the amount is debited and to which account the amount is credited. |
| accountStatement(Long mobileNo) | Long mobileNo | List<TransactionDTO> | This method used to send a list of all transactions. Invoke the appropriate method of Repository which will return a list of TransactionDTOs based on the mobile number received as a parameter and return the list. If no transactions are found in the database for a given mobile number throw InfyMeMobileException with the message NO\_ACTIVE\_TRANSACTIONS. |

**Repository Layer:**

• Create an interface AccountRepository, DigitalBankAccountRepository, TransactionRepository and make them as Spring Data JPA Repositories wherever possible.

• Refer to the Service classes to implement the above Repository interfaces.

• Create queries using method names if required.

• Use @Query annotation if required.

• Use @NamedQuery at Entity classes if required and reuse those methods at the Repository layer if possible.

• Note: The repository layer should not contain any Implementation classes. Only make usage of Query creation based on method names OR @Query annotation OR @NamedQuery annotations with appropriate JPQLs.

**Entity and DTO Layer:**

**Entity Class:**

• Create the below Entity classes.

• Annotate the classes with required annotations to make it an Entity class.

• Refer to the below instances.

**BankAccountEntity.java:**

This is an entity class for BankAccount and should be mapped with the BankAccount table.

|  |  |
| --- | --- |
| **Datatype** | **Variable** |
| Long | accountNumber |
| String | bankName |
| Double | balance |
| String | accountType |
| String | ifscCode |
| LocalDate | openingDate |
| Long | mobileNumber |

Make accountNumber as the primary key, use the appropriate annotations to map the variables to the columns if needed, and use the required association mapping for ‘mobileNumber.’

**DigitalBankAccountEntity.java:**

This is an entity class for DigitalBankAccount and should be mapped with the DigitalBankAccount table.

|  |  |
| --- | --- |
| **Datatype** | **Variable** |
| String | digitalBankingId |
| Long | mobileNumber |
| Long | accountNumber |
| String | accountType |

Make didgitalBankingId as the primary key, use the appropriate annotations to map the variables to the columns if needed, and use required association mapping for ‘mobileNumber’, and ‘accountNumber’.

**TransactionEntity.java:**

This is an entity class for Transaction and should be mapped with the Transaction table.

|  |  |
| --- | --- |
| **Datatype** | **Variable** |
| Integer | transactionId |
| String | modeOfTransaction |
| Long | paidTo |
| Long | receiverAccountNumber |
| Double | amount |
| LocalDateTime | transactionDateTime |
| String | remarks |
| Long | paidFrom |
| Long | senderAccountNumber |

Make transactionId as the primary key, and use the appropriate annotations to map the variables to the columns if needed.

**DTO Class:**

• Create the below DTO classes.

• Create the below variables as specified in DTO classes and add validations with custom error messages wherever necessary.

**BankAccountDTO.java:**

• This is a DTO class for BankAccount.

• Create the below variables in this class with given validations

|  |  |  |
| --- | --- | --- |
| **Datatype** | **Variable** | **Validation** |
| Long | accountNumber | Cannot be null, the minimum length should be 7 |
| String | bankName | Cannot be null, the minimum length should be 5, and the maximum length should be 15 |
| Double | balance | The minimum value must be 0. Can't be a negative value. |
| String | accountType | Cannot be null, the minimum length should be 1, and the maximum length should be 10 |
| String | ifscCode | Cannot be null, the minimum length should be 1, and the maximum length should be 15 |
| LocalDate | openingDate | Must be a past Date |
| Long | mobileNumber | Cannot be null, the minimum length should be 10, and the maximum length should be 10. |

**TransactionDTO.java:**

• This is a DTO class for Transaction.

• Create the below variables in this class

|  |  |
| --- | --- |
| **Datatype** | **Variable** |
| Integer | transactionId |
| String | modeOfTransaction |
| Long | paidTo |
| Long | receiverAccountNumber |
| Double | amount |
| LocalDateTime | transactionDateTime |
| String | remarks |
| Long | paidFrom |
| Long | senderAccountNumber |

**LoginDTO.java:**

• This is a DTO class for Login.

• Create the below variables in this class.

|  |  |
| --- | --- |
| **Datatype** | **Variable** |
| Long | mobileNumber |
| String | password |

**Note: Sample implementations for each layer have been given as per the requirement. If you feel that you need more implementations, you can proceed and add them too.**

**Utility Layer:**

 • Create a class OTPUtility.

• This class is used to generate the OTP for different mobile numbers. To simplify the implementation a hardcoded OTP is being generated every time for all the mobile numbers.

• If you want, you can enhance this functionality and generate different OTPs for different mobile numbers. You can make use of a table in the database also to generate/store the OTP as well.

**OTPUtility.java:**

1. import org.slf4j.Logger;
2. import org.slf4j.LoggerFactory;
3. import org.springframework.stereotype.Component;
4. @Component
5. public class OTPUtility {
6. private Logger logger = LoggerFactory.getLogger(this.getClass());
7. public Integer sendOTP() {
8. logger.info("Inside sendOTP() method of {}",
9. this.getClass());
10. return 123456;
11. }
12. }

**InfyMeMobile Exception Classes:**

• Create a class **InfyMeMobileGlobalExceptionHandler.java** for handling the global exceptions occurring in both the REST projects.

• Add the below classes to this package.

**ExceptionConstants.java:**

• This is an exception enum and contains exception constants.

• Add the below code in this class.

**Note: You can add more exception constants based on your need. When you are using the below ExceptionConstants in both the REST projects just remove the constants which are not in use.**

1. public enum ExceptionConstants {
2. SERVER\_ERROR("server.error"),
3. AUTHENTICATION\_FAILED(“authentication.failed”),
4. USER\_NOT\_FOUND("user.not.found"),
5. USERID\_NOT\_FOUND("user.id.not.found"),
6. NO\_USERS\_FOUND("no.users.found"),
7. NO\_ACCOUNTS\_FOUND("no.account.found"),
8. NO\_ACCOUNT\_IS\_LINKED("no.account.is.linked"),
9. INSUFFICIENT\_FUNDS("insufficient.funds"),
10. NO\_ACTIVE\_TRANSACTIONS("no.active.transactions");
12. private final String type;
14. private ExceptionConstants(String type) {
15. this.type = type;
16. }
17. @Override
18. public String toString() {
19. return this.type;
20. }
21. }

**InfyMeMobileGlobalExceptionHandler.java:**

• This is a Controller Advice class and contains advice for handling controller exceptions.

• This class must be common to both the Spring REST projects.

• Annotate class with required annotation to make it Rest Controller Advice.

• Autowire an environment variable.

• Create the below methods of controller advice class.

• Handle the MethodArgumentNotValidException and ConstraintViolationExceptions as well.

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Input Parameters** | **Response** | **Description** |
| exceptionHandler(Exception ex) | Exception ex | ResponseEntity<String> OR ResponseEntity<ErrorInformation> | This method used for handling general exceptions and returns error information to the client with the status INTERNAL\_SERVER \_ERROR.  Handle the MethodArgumentNotValidException and ConstraintViolationExceptions in this method only. |
| exceptionHandler(InfyMeMobileException e) | InfyMeMobileException ex | ResponseEntity<ErrorInformation> | This method is used for handling InfyMeMobileException and returns error information to the client with the status OK. |

**ErrorInformation.java:**

• This is a model class to store the error information like an error message, error code, and error time stamp.

• Create the ErrorInformation.java as below:

|  |  |
| --- | --- |
| **Datatype** | **Variable** |
| String | errorMessage |
| Integer | errorCode |
| LocalDateTime | errorTimeStamp |

**InfyMeMobile Backend Configurations**

**InfyMeMobileUserApplication.java Or InfyMeMobileAccountApplication.java:**

• This is the starter class for the InfyMeMobile application and contains the main method.

• Annotate the class with the required annotation to make it a Spring Boot application.

• Mention property source to read values from messages.properties.

**application.properties:**

1. *#Application properties*
2. *#name of application*
3. *#Web properties*
4. *#Provide port number of application*
5. *#Database properties*
6. *#MySQL driver name*
7. *#database URL*
8. *#database username*
9. *#database password*
10. *#update or create tables automatically*
11. *#name of log file*

**messages.properties:**

**• Add the below custom properties.**

**• Add the appropriate properties for DTO class validations in a separate properties file – ValidationMessages.properties.**

1. server.error= Some Error Occurred Kindly try after some time.
2. authentication.failed= Authentication failed for the given phone number or password
3. user.not.found= User for given phone number not found.
4. user.id.not.found= User for given user id not found.
5. no.users.found= No Users present in Database.
6. no.account.found= No Bank Accounts found for given phone number.
7. no.account.is.linked= No Account is linked for given phone number.
8. insufficient.funds= Insufficient Funds. Fund Transfer can’t be done.
9. no.active.transactions= No active Transactions for the given phone number and account number

**Note: So, apart from this if at any layer you need to add/modify more functionality you can add/modify them based on your need**