System Requirements Specification

Index

For

Banking Application

Version 1.0

TABLE OF CONTENTS

BANKING APPLICATION
System Requirements Specification

BACKEND-SPRING DATA RESTFUL APPLICATION

1 PROJECT ABSTRACT

The **Banking App** is implemented using Spring Data with a MySQL database, designed to optimize dynamic and secure banking operations. This app supports a variety of banking services, including account management, transaction processing, and loan handling, ensuring that users can manage their financial activities efficiently.

You are tasked with constructing a system where users can effortlessly manage banking accounts, transactions, loans, and user profiles. The application should enable users to create, update, and delete user profiles, accounts, and loan applications dynamically and transactionally. Functions such as fund transfers, applying for loans, repayment, and generating account statements should be seamlessly integrated to provide real-time financial management capabilities. The system must also support searching and filtering transactions and loans by various parameters to ensure robust data handling and user convenience, maintaining a high level of transactional integrity and operational dynamics efficiently.

Following is the requirement specifications:

	Banking Application
Modules	
1	User
2	Transaction
3	Loan
4	Account
User Module Functionalities	
1	Get user by id.
2	Create a user.
3	Update user by id.
4	Delete user by id.
5	Search users by name **(must use dynamic method).**

Transaction Module Functionalities	
1	Add a transaction **(must be transactional).**
2	Get all transactions for a user **(must return transactions by date in ascending order and that also in pages).**
3	Get user transactions with date range **(must use custom query).**

4	Get user transactions with amount range **(must use custom query).**

Loan Module Functionalities	
1	Apply for a loan **(must be transactional).**
2	Get loan details by loan id.
3	Update loan details by loan id **(must be transactional)**.
4	Delete loan application by loan id **(must be transactional).**
5	Get loans by user id **(must return lists of loans for a specified user id).**
6	Repay a loan.
7	Get loan status.

Account Module Functionalities	
1	Create an account**.**
2	Get an account by account id.
3	Update account by account id.
4	Delete account by account id**.**
5	Get accounts by user id **(must use custom query)**.
6	Get account balance.
7	Transfer funds **(must be transactional)**.
8	Get account statements**(must return the list of transactions for the specified period)**.

2 Assumptions, Dependencies, Risks / Constraints

2.1 User Constraints

- When fetching a user by ID, if the user ID does not exist, the service method should throw a NotFoundException with “User not found” message.
- When updating a user, if the user ID does not exist, the service method should throw a NotFoundException with “User not found” message.
- When removing a user, if the user ID does not exist, the service method should throw a NotFoundException with “User not found” message.

2.2 Transaction Constraints

- When adding a transaction:
    1) If the account ID does not exist, the service method should throw a NotFoundException with "Account not found" message.
    2) If the user ID does not exist, the service method should throw a NotFoundException with "User not found" message.
    3) If the account does not belong to the user specified by userId, the service method should throw a NotFoundException with "Account does not belong to user" message.

2.3 Loan Constraints

- When applying for a loan, if the user ID does not exist, the service method should throw a NotFoundException with the message "User not found".
- When fetching loan details, if the loan ID does not exist or does not belong to the given user ID, the service method should throw a NotFoundException with the message "Loan not found".
- When updating loan details, if the loan ID does not exist or does not belong to the given user ID, the service method should throw a NotFoundException with the message "Loan not found".
- When deleting a loan application, if the loan ID does not exist or does not belong to the given user ID, the service method should throw a NotFoundException with the message "Loan not found".
- When repaying a loan, if the loan ID does not exist or does not belong to the given user ID, the service method should throw a NotFoundException with the message "Loan not found".
- When fetching loan status, if the loan ID does not exist or does not belong to the given user ID, the service method should throw a NotFoundException with the message "Loan not found".

2.4 Account Constraints

- When creating an account, if the user ID does not exist, the service method should throw a NotFoundException with the message "User not found".
- When fetching an account by id, if the account ID does not exist, or if the account does not belong to the given user ID, the service method should throw a NotFoundException with the message "Account not found".
- When updating an account by id, if the account ID does not exist, or if the account does not belong to the given user ID, the service method should throw a NotFoundException with the message "Account not found".
- When deleting an account by id, if the account ID does not exist, or if the account does not belong to the given user ID, the service method should throw a NotFoundException with the message "Account not found".

- When fetching account balance, if the account ID does not exist, or if the account does not belong to the given user ID, the service method should throw a NotFoundException with the message "Account not found".
- When transferring funds between accounts:
    1) If either the 'from' or 'to' account ID does not exist, the service method should throw a NotFoundException with the message "Account not found".
    2) If the 'from' account does not belong to the given user ID, the service method should throw a NotFoundException with the message "From account not found".
    3) If the 'from' account has insufficient funds to cover the transfer, the service method should throw an IllegalArgumentException with the message "Insufficient balance".
- When fetching account statements, if the account ID does not exist, or if the account does not belong to the given user ID, the service method should throw a NotFoundException with the message "Account not found".

COMMON CONSTRAINTS

- For all rest endpoints receiving @RequestBody, validation check must be done and must throw custom exceptions if data is invalid.
- All the business validations must be implemented in dto classes only.
- All the database operations must be implemented on entity object only.
- Do not change, add, remove any existing methods in the service layer.
- In Repository interfaces, custom methods can be added as per requirements.
- All RestEndpoint methods and Exception Handlers must return data wrapped in **ResponseEntity**.

3 Business Validations

3.1 User

- Id must be of type id.
- Name should not be empty.
- Email should not be empty and must be of type email and unique in the system.
- Password should not be empty.

3.2 Transaction

- Id must be of type id.
- Account should not be null.
- Amount should not be null and must be a positive value.
- Date should not be null.
- Type should not be null, min 1 and max 6 characters and must match either 'DEBIT' or 'CREDIT'.

3.3 Loan

- Id must be of type id.
- User should not be null.
- Amount should not be null and must be a positive value.
- Apply Date should not be null.
- Approval Date should not be null.
- Disbursement Date should not be null.
- Balance should not be null and must be a positive value.
- Status should not be null, must be between 1 and 20 characters, and must match one of 'APPLIED', 'APPROVED', 'DISBURSED', 'REPAID'.

3.4 Account

- Id must be of type id.
- User should not be null.
- Account Number should not be null, must be exactly 10 characters long, and must be numeric.
- Balance should not be null and must be a positive value.

4 REST ENDPOINTS

Rest End-points to be exposed in the controller along with method details for the same to be created.

4.1 USERCONTROLLER

URL Exposed		Purpose
1. /users/{id}		Get a user by id
Http Method	GET	
Parameter 1	Long (id)	
Return	UserDTO	
2. /users		Create a new user
Http Method	POST **The user data to be created must be received in the controller using @RequestBody.**	
Parameter	-	
Return	UserDTO	
3. /users/{id}		Updates existing user by id
Http Method	PUT **The user data to be updated must be received in the controller using @RequestBody.**	
Parameter 1	Long (id)	
Return	UserDTO	
4. /users/{id}		Deletes a user by id
Http Method	DELETE	
Parameter 1	Long (id)	
Return	-	
5. /users/search		Search users by name
Http Method	GET	
Request Parameter	String (name)	
Return	List<UserDTO>	

4.2 TRANSACTIONCONTROLLER

<table>
<tr><th>URL Exposed</th><th>Purpose</th></tr>
<tr><td>1. /users/{userId}/accounts/{accountId}/transactions
<table>
<tr><td>Http Method</td><td>POST

The transaction data to be created must be received in the controller using @RequestBody.</td></tr>
<tr><td>Parameter 1</td><td>Long (userId)</td></tr>
<tr><td>Parameter 2</td><td>String (accountId)</td></tr>
<tr><td>Return</td><td>TransactionDTO</td></tr>
</table></td><td>Creates a new transaction</td></tr>
<tr><td>2. /users/{userId}/accounts/{accountId}/transactions
<table>
<tr><td>Http Method</td><td>GET</td></tr>
<tr><td>Parameter 1</td><td>Long (userId)</td></tr>
<tr><td>Return</td><td>List<TransactionDTO></td></tr>
</table></td><td>Fetches a list of all transaction for specified user</td></tr>
<tr><td>3. /users/{userId}/accounts/{accountId}/transactions/amount-range
<table>
<tr><td>Http Method</td><td>GET</td></tr>
<tr><td>Parameter 1</td><td>Long (userId)</td></tr>
<tr><td>Request Parameter 1</td><td>BigDecimal (minAmount)</td></tr>
<tr><td>Request Parameter 2</td><td>BigDecimal (maxAmount)</td></tr>
<tr><td>Return</td><td>List<TransactionDTO></td></tr>
</table></td><td>Fetches all transactions in amount range</td></tr>
<tr><td>4. /users/{userId}/accounts/{accountId}/transactions/date-range
<table>
<tr><td>Http Method</td><td>GET</td></tr>
<tr><td>Parameter 1</td><td>Long (userId)</td></tr>
<tr><td>Request Parameter 1</td><td>LocalDateTime (startDate)</td></tr>
</table></td><td>Fetches all transactions in date range</td></tr>
</table>

Request Parameter 2	LocalDateTime (endDate)	
Return	List<TransactionDTO>	

4.3 LoanController

URL Exposed		Purpose
1. /users/{userId}/loans/{loanId}		Retrieves loan details by loan ID for a specified user
Http Method	GET	
Parameter 1	Long (userId)	
Parameter 2	Long (loanId)	
Return	LoanDTO	
2. /users/{userId}/loans		Apply for a loan for a specific user
Http Method	POST **The loan data to be created must be received in the controller using @RequestBody.**	
Parameter 1	Long (userId)	
Return	LoanDTO	
3. /users/{userId}/loans/{loanId}		Updates specific loan details for a given loan ID and user
Http Method	PUT **The loan data to be updated must be received in the controller using @RequestBody.**	
Parameter 1	Long (userId)	
Parameter 2	Long (loanId)	
Return	LoanDTO	
4. /users/{userId}/loans/{loanId}		Deletes a loan application by its ID for a specified user
Http Method	DELETE	
Parameter 1	Long (userId)	
Parameter 2	Long (loanId)	
Return	-	
5. /users/{userId}/loans		
Http Method	GET	

Parameter 1	Long (userId)	Retrieves a list of all loans associated with a specified user ID
Return	List<LoanDTO>	

6. /users/{userId}/loans/{loanId}/repay		
Http Method	POST	Submits a repayment towards a loan for a specific user
Parameter 1	Long (userId)	
Parameter 2	Long (loanId)	
Request Parameter	BigDecimal (amount)	
Return	-	

7. /users/{userId}/loans/{loanId}/status		
Http Method	GET	Retrieves the status of a specific loan
Parameter 1	Long (userId)	
Parameter 2	Long (loanId)	
Return	String	

4.4 AccountController

URL Exposed		Purpose
1. /users/{userId}/accounts/{accountId}		Retrieves details of an account by account ID for a specified user
Http Method	GET	
Parameter 1	Long (userId)	
Parameter 2	Long (accountId)	
Return	AccountDTO	
2. /users/{userId}/accounts		Create a new account for a specific user
Http Method	POST **The account data to be created must be received in the controller using @RequestBody.**	
Parameter 1	Long (userId)	
Return	AccountDTO	
3. /users/{userId}/accounts/{accountId}		Updates details of an existing account
Http Method	PUT **The account data to be updated must be received in the**	

	controller using @RequestBody.	
Parameter 1	Long (userId)	
Parameter 2	Long (accountId)	
Return	AccountDTO	

4. /users/{userId}/accounts/{accountId}	
Http Method	DELETE
Parameter 1	Long (userId)
Parameter 2	Long (accountId)
Return	-
	Deletes an account by its ID for a specified user

5. /users/{userId}/accounts	
Http Method	GET
Parameter 1	Long (userId)
Return	List<AccountDTO>
	Retrieves lists of all accounts associated with a specific user ID

6. /users/{userId}/accounts/{accountId}/balance	
Http Method	GET
Parameter 1	Long (userId)
Parameter 2	Long (accountId)
Return	BigDecimal
	Retrieves the balance of a specified account for a specific user

7. /users/{userId}/accounts/transfer	
Http Method	POST
Parameter 1	Long (userId)
Request Parameter 1	Long (fromAccountId)
Request Parameter 2	Long (toAccountId)
Request Parameter 3	BigDecimal (amount)
Return	-
	Transfers funds between two accounts of the same user

8. /users/{userId}/accounts/{accountId}/statements	
Http Method	GET
Parameter 1	Long (userId)
Parameter 2	Long (accountId)
Request Parameter 1	LocalDateTime (startDate)
Request Parameter 2	LocalDateTime (endDate)
	Retrieves the transaction statements for an account within a specified date range

Return	List<TransactionDTO>	

5 Template Code Structure

5.1 Package: com.bankingapplication

Resources

BankingApplication (Class)	This is the Spring Boot starter class of the application.	Already Implemented

5.2 Package: com.bankingapplication.repository

Resources

Class/Interface	Description	Status
TransactionRepository (interface)	● Repository interface exposing CRUD functionality for Transaction Entity. ● It must contain the methods for: ○ Finding all users ordered by transaction date ascendingly. ○ Finding all transactions by amount in range. ○ Finding all transactions by user id and transaction date range. ● You can go ahead and add any custom methods as per requirements.	Partially implemented.

UserRepository (interface)	• Repository interface exposing CRUD functionality for User Entity.
	• It must contain the methods for:
	  ○ Finding all users by name.
	  ○ Find users by email.
	• You can go ahead and add any custom methods as per requirements.
		Partially implemented.

LoanRepository (interface)	• Repository interface exposing CRUD functionality for Loan Entity.
	• It must contain the methods for:
	  ○ Finding all loans for a specific user id.
	  ○ Finding loans by their status.
	• You can go ahead and add any custom methods as per requirements.
		Partially implemented.

AccountRepository (interface)	• Repository interface exposing CRUD functionality for Account Entity. • It must contain the methods for: ○ Finding all accounts for a specific user. • You can go ahead and add any custom methods as per requirements.	Partially implemented.

5.3 PACKAGE: COM.BANKINGAPPLICATION.SERVICE

Resources

Class/Interface	Description	Status
TransactionService (interface)	• Interface to expose method signatures for transaction related functionality. • Do not modify, add or delete any method.	Already implemented.
UserService (interface)	• Interface to expose method signatures for user related functionality. • Do not modify, add or delete any method.	Already implemented.
LoanService (interface)	• Interface to expose method signatures for loan related functionality. • Do not modify, add or delete any method.	Already implemented.
AccountService (interface)	• Interface to expose method signatures for account related functionality. • Do not modify, add or delete any method.	Already implemented.

5.4 PACKAGE: COM.BANKINGAPPLICATION.SERVICE.IMPL

Class/Interface	Description	Status
TransactionServiceImpl (class)	• Implements TransactionService. • Contains template method implementation. • Need to provide implementation for transaction related functionalities. • Do not modify, add or delete any method signature	To be implemented.

UserServiceImpl (class)	• Implements UserService. • Contains template method implementation. • Need to provide implementation for user related functionalities. • Do not modify, add or delete any method signature	To be implemented.
LoanServiceImpl (class)	• Implements LoanService. • Contains template method implementation. • Need to provide implementation for loan related functionalities. • Do not modify, add or delete any method signature	To be implemented.
AccountServiceImpl (class)	• Implements AccountService. • Contains template method implementation. • Need to provide implementation for account related functionalities. • Do not modify, add or delete any method signature	To be implemented.

5.5 PACKAGE: COM.BANKINGAPPLICATION.CONTROLLER

Resources

Class/Interface	Description	Status
TransactionController (Class)	• Controller class to expose all rest-endpoints for transaction related activities. • May also contain local exception handler methods	To be implemented

UserController (Class)	• Controller class to expose all rest-endpoints for user related activities. • May also contain local exception handler methods	To be implemented
LoanController (Class)	• Controller class to expose all rest-endpoints for loan related activities. • May also contain local exception handler methods	To be implemented
AccountController (Class)	• Controller class to expose all rest-endpoints for account related activities. • May also contain local exception handler methods	To be implemented

5.6 PACKAGE: COM.BANKINGAPPLICATION.DTO

Resources

Class/Interface	Description	Status
TransactionDTO (Class)	Use appropriate annotations for validating attributes of this class.	Partially implemented.
UserDTO (Class)	Use appropriate annotations for validating attributes of this class.	Partially implemented.
LoanDTO (Class)	Use appropriate annotations for validating attributes of this class.	Partially implemented.
AccountDTO (Class)	Use appropriate annotations for validating attributes of this class.	Partially implemented.

5.7 PACKAGE: COM.BANKINGAPPLICATION.ENTITY

Resources

Class/Interface	Description	Status
Transaction (Class)	• This class is partially implemented. • Annotate this class with proper annotation to declare it as an entity class with **id** as primary key. • Map this class with a **transaction table**. • Generate the **id** using the IDENTITY strategy.	Partially implemented.
User (Class)	• This class is partially implemented. • Annotate this class with proper annotation to declare it as an entity class with **id** as primary key. • Map this class with a **user table**. • Generate the **id** using the IDENTITY strategy.	Partially implemented.
Loan (Class)	• This class is partially implemented. • Annotate this class with proper annotation to declare it as an entity class with **id** as primary key. • Map this class with a **loan table**. • Generate the **id** using the IDENTITY strategy.	Partially implemented.

Account (Class)	• This class is partially implemented. • Annotate this class with proper annotation to declare it as an entity class with **id** as primary key. • Map this class with an account **table**. • Generate the **id** using the IDENTITY strategy.	Partially implemented.

5.8 PACKAGE: COM.BANKINGAPPLICATION.EXCEPTION

Resources

Class/Interface	Description	Status
NotFoundException (Class)	• Custom Exception to be thrown when trying to fetch or delete the user/ transaction/loan/account info which does not exist. • Need to create Exception Handler for the same wherever needed (local or global).	Already implemented.
ErrorResponse (Class)	• RestControllerAdvice Class for defining global exception handlers. • Contains Exception Handler for **InvalidDataException** class. • Use this as a reference for creating exception handler for other custom exception classes.	Already implemented.

RestExceptionHandler (Class)	• RestControllerAdvice Class for defining rest exception handlers. • Contains Exception Handler for **NotFoundException** class. • Use this as a reference for creating exception handler for other custom exception classes.	Already implemented.

6 EXECUTION STEPS TO FOLLOW FOR BACKEND

1. **All actions like build, compile, running application, running test cases will be through Command Terminal.**
2. **To open the command terminal the test takers need to go to the Application menu (Three horizontal lines at left top) -> Terminal -> New Terminal.**
3. **cd into your backend project folder**
4. **To build your project use command:**

   mvn clean package -Dmaven.test.skip
5. **To launch your application, move into the target folder (cd target). Run the following command to run the application:**

   java -jar <your application jar file name>
6. **This editor Auto Saves the code.**
7. **If you want to exit(logout) and continue the coding later anytime (using Save & Exit option on Assessment Landing Page) then you need to use CTRL+Shift+B-command compulsorily on code IDE. This will push or save the updated contents in the internal git/repository. Else the code will not be available in the next login.**
8. **These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.**
9. **To test any Restful application, the last option on the left panel of IDE, you can find ThunderClient, which is the lightweight equivalent of POSTMAN. Please use**

127.0.0.1 instead of localhost to test rest endpoints.

10. To test any UI based application the second last option on the left panel of IDE, you can find Browser Preview, where you can launch the application.
11. Default credentials for MySQL:
    a. Username: root
    b. Password: pass@word1

12. To login to mysql instance: Open new terminal and use following command:
    a. sudo systemctl enable mysql
    b. sudo systemctl start mysql

    NOTE: After typing any of the above commands you might encounter any warnings.

    >> Please note that this warning is expected and can be disregarded. Proceed to the next step.

    c. mysql -u root -p

    The last command will ask for password which is 'pass@word1'
13. Mandatory: Before final submission run the following command:

    mvn test
14. You need to use CTRL+Shift+B - command compulsorily on code IDE, before final submission as well. This will push or save the updated contents in the internal git/repository, and will be used to evaluate the code quality.