System Requirements Specification

Index

For

Expense Splitter

Version 1.0

TABLE OF CONTENTS

EXPENSE SPLITTER APPLICATION

System Requirements Specification

BACKEND-SPRING BOOT RESTFUL APPLICATION

1 PROJECT ABSTRACT

The **Expense Splitter Application** is implemented using Spring Boot with a MySQL database, designed to facilitate the management of shared expenses. This application acts as a comprehensive financial tool, allowing users to track, manage, and settle their shared expenses effectively.

You are tasked with creating a system that enables users to effortlessly add, update, delete, list and manage users and their expenses. The application will include functionalities to create new expenses, update and delete existing ones, settle expenses, as well as view all expenses and balances, either individually or between users.

Following is the requirement specifications:

	Expense Splitter Application
Modules	
1	User
2	Expense
User Module Functionalities	
1	List all users **(must return all users by name and that also in list)**
2	Get user by id
3	Create an user
4	Update an user by id
5	Delete an user by id

Expense Module Functionalities	
1	Create an expense
2	Update an expense by id
3	Delete an expense by id
4	Get an expense by id
5	Get list of all expenses **(must use custom query)**
6	Settle an expense by its id **(must use custom query)**
7	Get list of expenses by user id **(must use custom query to return list of expenses by user id)**
8	Get list of user balance **(must use custom query)**
9	Calculate balance between two users **(must use custom query)**

Overall Application	
1	Actuator support needs to be added in the properties file. Expose all actuator endpoints except beans.
2	In application.properties file expose a property “profile.validate.data” with value as “This is default profile”. Create application-qa.properties file (for QA profile) and expose a property “profile.validate.data” with value as “This is qa profile”.
3	Create an endpoint in UserController with following configurations: 1. Method – GET 2. Endpoint - /profile 3. Return – String **The method for this endpoint must read the “profile.validate.data” property file and return its value based on the active profile.**

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

2.2 Expense Constraints

- When deleting an expense by ID, if the expense ID does not exist, the service method should throw a ResourceNotFoundException with “Expense not found” message.
- When fetching an expense by ID, if the expense ID does not exist, the service method should throw a ResourceNotFoundException with “Expense not found” message.
- When updating an expense by ID, if the expense ID does not exist, the service method should throw a ResourceNotFoundException with “Expense not found” message.
- When settling an expense, if the expense ID does not exist, the service method should throw a ResourceNotFoundException with “Expense not found” message.
- When listing expenses by user ID, if the user ID does not exist, the service method should throw a ResourceNotFoundException with "User not found" message.

- When listing user balances, if the user ID does not exist, the service method should throw a ResourceNotFoundException with the message "User not found with id: [userID]".
- When calculating the balance between two users, if either of the user IDs (userId or otherUserId) does not exist, the service method should throw a ResourceNotFoundException with the message "One or both users not found".

COMMON CONSTRAINTS

- For all rest endpoints receiving @RequestBody, validation check must be done and must throw custom exception 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 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

User:

- Id must be of type id.
- Name should not be blank and max 200 characters.
- Email value is not blank and of type email and max 200 characters.

Expense:

- Id must be of type id.
- Description should not be blank and max 500 characters.
- Amount should not be null and must be a positive value.
- paidById should not be null.
- sharedWithIds should not be null.
- isSettled should not be null.

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. /api/users		Retrieves a list of all users
Http Method	GET	
Parameter	-	
Return	List<UserDTO>	
2. /api/users/{userId}		Get a user by id
Http Method	GET	
Parameter 1	Long (id)	
Return	UserDTO	
3. /api/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	
4. /api/users/{userId}		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	
5. /api/users/{userId}		Deletes a user by id
Http Method	DELETE	
Parameter 1	Long (id)	
Return	-	

6. /api/users/profile	
Http Method	GET
Parameter 1	-
Return	String
Fetches the profile	

4.2 ExpenseController

URL Exposed		Purpose
1. /api/expenses		
Http Method	POST **The expense data to be created must be received in the controller using @RequestBody.**	Creates a new expense record
Parameter	-	
Return	ExpenseDTO	
2. /api/expenses/{expenseId}		
Http Method	PUT **The expense data to be updated must be received in the controller using @RequestBody.**	Updates an existing expense by its ID
Parameter 1	Long (id)	
Return	ExpenseDTO	
3. /api/expenses/{expenseId}		
Http Method	DELETE	Deletes an expense by its ID
Parameter	Long (id)	
Return	-	
4. /api/expenses/{expenseId}		
Http Method	GET	Fetches an expense by id
Parameter 1	Long (id)	
Return	ExpenseDTO	
5. /api/expenses		
Http Method	GET	Retrieves a list of all expenses
Parameter 1	-	
Return	List<ExpenseDTO >	

6. /api/expenses/{expenseId}/settle		Settles an expense by its ID
Http Method	POST	
Parameter 1	Long (expenseId)	
Return	-	

7. /api/expenses/user/{userId}		Retrieves the expenses of a specific user
Http Method	GET	
Parameter 1	Long (userId)	
Return	List<ExpenseDTO >	

8. /api/expenses/balances/{userId}		Retrieves the balance of a user
Http Method	GET	
Parameter 1	Long (userId)	
Return	Map<String, Double>	

9. /api/expenses/balance/{userId}/{otherUserId}		Calculates and returns the balance between two specified users
Http Method	GET	
Parameter 1	Long (userId)	
Parameter 2	Long (otherUserId)	
Return	Double	

5 TEMPLATE CODE STRUCTURE

5.1 PACKAGE: COM.EXPENSESPLITTER

Resources

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

5.2 PACKAGE: COM.EXPENSESPLITTER.REPOSITORY

Resources

Class/Interface	Description	Status
UserRepository (interface)	● Repository interface exposing CRUD functionality for User Entity. ● You can go ahead and add any custom methods as per requirements.	Already implemented.
ExpenseRepository (interface)	● Repository interface exposing CRUD functionality for Expense Entity. ● It must contain the methods for: ○ Fetching all expenses paid by a specific user and it must return data in the list. ○ Fetching all expenses where a specific user is part of the sharedWith list and it must return data in the list . ○ Finding list of all expenses involving two users.	Partially implemented.

	• You can go ahead and add any custom methods as per requirements.	

5.3 PACKAGE: COM.EXPENSESPLITTER.SERVICE

Resources

Class/Interface	Description	Status
UserService (interface)	• Interface to expose method signatures for user related functionality. • Do not modify, add or delete any method.	Already implemented.
ExpenseService (interface)	• Interface to expose method signatures for expense related functionality. • Do not modify, add or delete any method.	Already implemented.

5.4 PACKAGE: COM.EXPENSESPLITTER.SERVICE.IMPL

Class/Interface	Description	Status
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.

ExpenseServiceImpl (class)	• Implements ExpenseService. • Contains template method implementation. • Need to provide implementation for expense related functionalities. • Do not modify, add or delete any method signature	To be implemented.

5.5 PACKAGE: COM.EXPENSESPLITTER.CONTROLLER

Resources

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

5.6 PACKAGE: COM.EXPENSESPLITTER.DTO

Resources

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

5.7 PACKAGE: COM.EXPENSESPLITTER.ENTITY

Resources

Class/Interface	Description	Status
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.
Expense (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 **expense table**. • Generate the **id** using the IDENTITY strategy	Partially implemented.

5.8 PACKAGE: COM.EXPENSESPLITTER.EXCEPTION

Resources

Class/Interface	Description	Status
ResourceNotFoundException (Class)	• Custom Exception to be thrown when trying to fetch, update or delete the user or expense info which does not exist.	Already implemented.

	• Need to create Exception Handler for same wherever needed (local or global)	
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 **ResourceNotFoundException** class. • Use this as a reference for creating exception handler for other custom exception classes	Already implemented.

5.9 PROPERTIES FILES

Resources

Class/Interface	Description	Status
application.properties	• This file is treated as the default properties file for this application. • You need to write properties to add actuator support. • You need to write property to expose all endpoints. • You need to write property to exclude /beans endpoint. • Add "profile.validate.data" property with value as "This is default profile".	Partially implemented.
application-qa.properties	• This file is treated as the qa properties file for this application. • You need to write properties to add actuator support. • You need to write property to expose all endpoints. • You need to write property to exclude /beans endpoint. • Add "profile.validate.data" property with value as "This is qa profile".	To be 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.