Library Management System (LMS) - User & Deployment Guide

1. Overview

This document provides comprehensive instructions for setting up, deploying, and utilizing the Library Management System (LMS). It also outlines key considerations and potential areas for future enhancements to further strengthen the system's security, stability, and maintainability. The LMS is a Java-based application designed to manage library resources, member information, and book transactions through a command-line interface.

2. System Requirements & Setup

To successfully deploy and run the Library Management System, the following components and configurations are necessary:

2.1. Prerequisites

- **Java Development Kit (JDK):** Version 8 or higher is required. Ensure the JDK is properly installed and configured on your system.
- **Build Tool (Recommended):** Apache Maven or Gradle is highly recommended for streamlined dependency management. Key dependencies include:
  - A JPA provider (e.g., `org.hibernate:hibernate-core` for Hibernate, or EclipseLink).
  - The appropriate JDBC driver for your chosen database (e.g., `mysql:mysql-connector-java` for MySQL, `org.postgresql:postgresql` for PostgreSQL, `com.h2database:h2` for H2).
- **Relational Database Server:** A running instance of a relational database such as MySQL, PostgreSQL, or H2 is required.

2.2. JPA Configuration (`persistence.xml`)

The `persistence.xml` file is essential for JPA to function. It defines the persistence unit, database connection details, and entity classes. This file must be located in the `src/main/resources/META-INF/` directory of your project.

Note:

- Adjust the `javax.persistence.jdbc.*` properties according to your specific database server and credentials.
- The `hibernate.hbm2ddl.auto` property manages schema generation. Use `update` or `create` for development and initial setup. For production environments, it's often preferred to manage schema changes manually or use `validate`.

3. Compiling and Running the Application

- **Compile Project:**
  - If using an IDE (e.g., IntelliJ IDEA, Eclipse), the IDE typically handles compilation automatically.
  - If using Maven: `mvn clean compile`
  - If using Gradle: `gradle clean build`
- **Run Application:**
  - Execute the main method located in the `org.lms.Main` class. This can be done via your IDE or from the command line if your project is packaged into a JAR.

4. Application Usage

Upon successful execution, the system will present the main menu:

```
****************************************
Welcome to Library Management System
1.Login As Librarian
2.Login As Member
3.Exit the System
****************************************
Enter your choice:
```

4.1. Librarian Portal

Access by selecting option 1.
Default Credentials:

- Username: admin
- Password: admin
  (Note: These are hardcoded in AuthService.java. Refer to Section 5 for recommendations on credential management.)

Upon successful login, the Librarian Main Menu is displayed:

```
********Login Successful*******
1.Manage Books
2.Manage Members
3.Manage Transactions
4.Exit
*******************************
Enter your Choice:
```

Functionalities:

- **Manage Books:**
  - **Add Book:** Prompts for title, author, available copies, and genre. Books are initially set to AVAILABLE status.
  - **Update Book stocks:** Lists all books, prompts for a Book ID, and then the quantity to add/remove (negative values reduce stock).
  - **Delete Book:** Lists available books and prompts for a Book ID. Deletion is prevented if the book has active borrow transactions. The book's status is set to UNAVAILABLE, and its available copies are set to zero (soft delete).
  - **List Books:** Displays a comprehensive list of all books.
  - **List Available Books:** Displays books with copiesAvailable > 0 and status = AVAILABLE.
  - **Exit:** Returns to the Librarian Main Menu.
- **Manage Members:**
  - **Add Members:** Prompts for name, username, and password.
    - If the username exists and the provided name matches the existing record, it updates the password and sets the member's status to ACTIVE.
    - If the username exists but the name differs, the system prompts for a different username.
    - Otherwise, a new member is created with ACTIVE status.
  - **Delete Members:** Lists active members and prompts for a Member ID. Deletion is prevented if the member has active transactions. The member's status is set to INACTIVE.
  - **View All Members:** Displays all registered members (passwords are masked in the display).
  - **View All Active Members:** Displays members currently with status = ACTIVE.
  - **Find By ID:** Prompts for a Member ID and displays the corresponding member's details.
  - **Find By Username:** Prompts for a username and displays the corresponding member's details.
  - **Make User Active:** Lists inactive members and prompts for a Member ID to reactivate, setting their status to ACTIVE.
  - **Exit:** Returns to the Librarian Main Menu.

- **Manage Transactions:**
  - **All Transactions:** Displays a complete history of all recorded transactions.
  - **Transactions of A Member:** Lists members, prompts for a Member ID, and then displays all transactions associated with that member.
  - **Transactions of Type:** Prompts for transaction type (BORROW/RETURN) and displays relevant transactions.
  - **Transactions by Status:** Prompts for status (Active/Completed) and type (Borrow/Return), then displays matching transactions.
  - **Find** By **ID:** Prompts for a Transaction ID and displays its details.
  - **Find By Book ID:** Lists books, prompts for a Book ID, and displays all transactions involving that specific book.
  - **Exit:** Returns to the Librarian Main Menu.

4.2. Member Portal

Access by selecting option 2 from the main welcome screen.
Members must enter their registered username and password. Only members with ACTIVE status can log in.
Upon successful login, the Member Main Menu is displayed:

```
********Login Successful*******
1.Borrow Books
2.Return Books
3.View My Transactions
4.Exit
*******************************
Enter your Choice:
```

Functionalities:

- **Borrow Books:**
  - Lists all currently available books.
  - Prompts the member for the Book ID they wish to borrow.
  - If successful, a BORROW transaction is created, the book's available copies are decremented, and the expected return date is set (e.g., 7 days from borrowing).
- **Return Books:**
  - Lists the member's active BORROW transactions (books currently borrowed by them).
  - Prompts for the Transaction ID corresponding to the book being returned.

    - A RETURN transaction is created, the original BORROW transaction is marked as COMPLETED, and the book's available copies are incremented. The system may flag late returns.
- **View My Transactions:**
    - Displays a complete history of all transactions (both BORROW and RETURN) associated with the logged-in member's account.
- **Exit:** Logs the member out and returns to the main welcome screen.

5. Key Considerations & Future Enhancements

The following points highlight areas for potential refinement and future development, aiming to bolster the system's security, robustness, and overall maintainability, aligning with industry best practices.

- persistence.xml **Configuration:** As detailed in Section 2.2, the correct setup of this configuration file is fundamental for JPA functionality and database interaction.
- **Administrator Credential Management:**
    - **Observation:** Administrator credentials are currently hardcoded within the application (AuthService.java).
    - **Recommendations for Enhanced Security:**
        - Transition to storing administrator credentials in a secure manner, such as an external configuration file with appropriate access controls.
        - Consider implementing an initial setup routine where an administrator account is created in the database, prompting for a strong, unique password.
- **Password** Security **(Member Accounts):**
    - **Observation:** Member passwords appear to be handled in plaintext during authentication and storage.
    - **Best Practice Recommendation:** To significantly enhance security, implement password hashing.
        - Utilize strong, industry-standard hashing algorithms (e.g., BCrypt, SCrypt, or Argon2).
        - Store only the generated password hashes in the database.
        - During login, hash the user-provided password and compare it against the stored hash for verification.
- **Service-Layer Transaction Management:**
    - **Current Approach:** Data Access Objects (DAOs) presently manage their own database transactions.
    - **Recommendation for Improved Atomicity:** For business operations that involve multiple DAO interactions (e.g., updating book stock and recording a

transaction simultaneously), consider centralizing transaction management at the service layer. This ensures that all related database changes within a single operation either complete successfully together or are entirely rolled back if an error occurs, maintaining data integrity.

- **Error Handling and Input Validation Strategies:**
  - **Error Handling Refinement:**
    - Enhance specificity in exception handling. Instead of generic `catch (Exception e)` blocks, aim to catch more precise exception types. This facilitates more targeted error recovery and simplifies debugging. For instance, `em.find()` returns `null` if an entity isn't found, which can be handled with a direct `null` check, reserving `try-catch` blocks for other persistence-related exceptions.
    - Revisit `NullPointerException` handling, for example, in `LibrarianController.MainMenu` during login. A direct check for a `null` result from `AuthService.authenticate` can lead to clearer and more explicit control flow for authentication failures.
  - **Comprehensive Input Validation:** Augment current input validation (which primarily addresses `InputMismatchException`) to include checks for empty strings, adherence to expected data formats (e.g., for dates), and validation of numerical inputs against logical ranges.
- **Book Deletion Workflow (**`LibrarianController.manageBooks` **case 3):**
  - **Observation:** The logic to check if a book is currently in use before deletion involves `transactions.getFirst()`, which might throw `NoSuchElementException` for an empty list.
  - **Suggested Refinement:** A clearer approach would be to first check if `(!transactions.isEmpty())`. If active borrow transactions exist for the book, the system should explicitly prevent deletion (even soft deletion) or provide a clear warning and require confirmation from the librarian.
- **Book Availability Check (**`TransactionService.borrowBook`**):**
  - **Observation:** The condition `if(book.getCopiesAvailable() < 0)` is used.
  - **Suggested Refinement:** To accurately reflect book availability, this condition should ideally be `if(book.getCopiesAvailable() <= 0)` or, more precisely, `if(book.getCopiesAvailable() < 1)`, as a book with zero copies is unavailable.
- **Member Creation and Update Logic (**`LibrarianController.manageMembers` **case 1):**
  - **Observation:** The current workflow for adding members when a username might already exist has multiple paths that could be streamlined.
  - **Recommendations for Clarity:**
    - Clearly define the policy for existing usernames: either enforce unique usernames by disallowing creation if the username exists, or

        - Separate the "Add New Member" functionality from "Update Existing Member" or "Reactivate Member" functionalities for better user understanding and system logic. The current message when a username exists but the name differs can also be clarified.
- **Distinction in Book Deletion Methods (**`BookDAO.delete(int id)` **vs. Controller's Soft Delete):**
    - **Observation:** `BookDAO.delete(int id)` performs a physical (hard) delete from the database, while the `LibrarianController` implements a logical (soft) delete by updating the book's status.
    - **Clarification and Best Practice:** The controller's soft delete approach is generally preferred as it preserves historical data and maintains referential integrity with transaction records. If a hard delete capability is necessary, its use should be carefully controlled and documented, ensuring all data relationships and potential impacts are considered.
- **Structured Logging Implementation:**
    - **Recommendation:** Adopt a dedicated logging framework (e.g., Log4j2 or SLF4j with Logback). This offers significant advantages over `System.out.println`, including configurable log levels, output destinations, and formatting, which are invaluable for debugging, auditing, and application monitoring.
- **User Experience (UX) Considerations for Future Development:**
    - **Data Presentation:** Strive for consistent date and time formatting across all user-facing displays.
    - **Feedback Messages:** Ensure that messages for successful operations and error conditions are clear, concise, and informative.
    - **Scalability of Display:** For potentially long lists (e.g., books, members, transactions), consider implementing pagination or searchable/filterable views to enhance usability.