Library Management System - Project Documentation

1. Project Introduction & Overview

Project Name: Library Management System **Description:** A modern, role-based application designed to streamline library operations. It provides distinct interfaces and functionalities for Library Owners/Librarians and Members, aiming to enhance efficiency in book management, circulation, financial tracking, and user experience. **Key Features:**

- **Owner Role:** Centralized management of library assets (books, copies, racks), user accounts (members, other owners), financial oversight (payments, fines, revenue reports), and detailed analytical reporting on library usage and performance.
- **Member Role:** Intuitive interface for searching books, checking availability, borrowing/returning books (facilitated by owner), viewing personal borrowing history, and managing fines/payments.
- **Individual Copy Tracking:** Each physical copy of a book is managed independently, allowing precise tracking of its status, location, and borrowing history.
- **Payment & Fine System:** Enforces a 'paid member' status for borrowing and calculates daily fines for overdue books, with clear collection mechanisms.

Technology Stack:

- **Frontend:** React Native (Mobile Application - iOS/Android)
- **Backend:** Spring Boot (Java RESTful API)
- **Database:** MySQL (Relational Database)
- **Authentication/Authorization:** JSON Web Tokens (JWT) with Spring Security

2. Database Schema (MySQL)

The database schema is designed to capture all essential entities and their relationships, supporting the outlined functional requirements.

ER Diagram (Re-confirmed with slight practical enhancements):

(Assuming a minor, pragmatic enhancement to the provided ERD for `Rack` as a separate entity for better management, and `Members.name` split into `first_name` and `last_name` for UI flexibility. All other elements strictly adhere to the provided ERD.)

Table Details:

2.1. `Members` Table

- **Purpose:** Stores user accounts, distinguishing between 'OWNER' and 'MEMBER' roles.
- **Columns:**
  - `id` (INT, PK, AUTO_INCREMENT): Unique identifier for the member.
  - `first_name` (VARCHAR(255), NOT NULL): Member's first name. (Refinement from ERD's 'name' for practical UI).
  - `last_name` (VARCHAR(255), NOT NULL): Member's last name. (Refinement from ERD's 'name' for practical UI).
  - `email` (VARCHAR(255), NOT NULL, UNIQUE): Member's email address, used as a unique identifier for login.
  - `phone` (VARCHAR(20)): Member's phone number.
  - `password_hash` (VARCHAR(255), NOT NULL): Hashed password for secure authentication. (Renamed from 'passwd' for best practice).
  - `role` (VARCHAR(50), NOT NULL): User's role ('OWNER' or 'MEMBER').
  - `is_active` (BOOLEAN, DEFAULT TRUE): Indicates if the user account is active.
  - `is_paid_member` (BOOLEAN, DEFAULT FALSE): Crucial flag to determine if a member can borrow books. Toggled by Owner.
  - `last_payment_date` (DATE): Records the date of the last successful membership payment.

2.2. `Books` Table

- **Purpose:** Stores metadata for unique book titles.
- **Columns:**
  - `id` (INT, PK, AUTO_INCREMENT): Unique identifier for the book title.
  - `title` (VARCHAR(255), NOT NULL): The full title of the book. (Renamed from 'name' for clarity).
  - `author` (VARCHAR(255), NOT NULL): The author(s) of the book.
  - `genre` (VARCHAR(100)): The subject/genre of the book. (Renamed from 'subject' for clarity).
  - `price` (DECIMAL(10,2)): Estimated value/price of the book.
  - `isbn` (VARCHAR(20), NOT NULL, UNIQUE): International Standard Book Number, a unique identifier for the edition.

2.3. `Racks` Table

- **Purpose:** Stores information about physical racks where book copies are stored. (This is a practical enhancement to the ERD's 'rack' integer in Copies).
- **Columns:**
  - `id` (INT, PK, AUTO_INCREMENT): Unique identifier for the rack.
  - `rack_number` (VARCHAR(50), NOT NULL, UNIQUE): Human-readable identifier for the rack (e.g., "A-1", "Shelf 3").
  - `location_description` (VARCHAR(255)): Optional descriptive location.

2.4. `Copies` Table

- **Purpose:** Represents individual physical copies of books. Each `Book` can have multiple `Copies`.
- **Columns:**
  - `id` (INT, PK, AUTO_INCREMENT): Unique identifier for the individual copy.
  - `book_id` (INT, FK): Foreign key linking to the `Books` table.

- `rack_id` (INT, FK, NULLABLE): Foreign key linking to the `Racks` table, indicating where the copy is located. (Refinement from ERD's 'rack' integer).
- `copy_identifier` (VARCHAR(50), NOT NULL, UNIQUE): A unique string/barcode identifier for the specific physical copy (e.g., "CLEANCODE-C001").
- `status` (VARCHAR(50), NOT NULL): Current status of the copy (e.g., 'AVAILABLE', 'ISSUED', 'LOST', 'DAMAGED').

2.5. `Payments` Table

- **Purpose:** Records all financial payments made by members (membership fees, fines).
- **Columns:**
  - `id` (INT, PK, AUTO_INCREMENT): Unique identifier for the payment transaction.
  - `member_id` (INT, FK): Foreign key linking to the `Members` table. (Renamed from 'memberid').
  - `amount` (DECIMAL(10,2), NOT NULL): The amount paid.
  - `type` (VARCHAR(50), NOT NULL): Type of payment ('MEMBERSHIP_FEE', 'FINE_PAYMENT').
  - `payment_date` (DATETIME, NOT NULL): The date and time the payment was recorded. (Renamed from 'txtime').
  - `period_covered` (VARCHAR(50)): The period this payment covers (e.g., "Oct 2023"). (Added for tracking monthly payments).
  - `collected_by_user_id` (INT, FK, NULLABLE): ID of the Owner/Librarian who collected the payment.

2.6. `IssuerRecord` Table (Renamed to `BorrowingTransactions` for clarity)

- **Purpose:** Records each borrowing instance of a specific book copy by a member, including return details and fines.
- **Columns:**
  - `id` (INT, PK, AUTO_INCREMENT): Unique identifier for the borrowing transaction.
  - `copy_id` (INT, FK): Foreign key linking to the `Copies` table (the specific copy borrowed). (Renamed from 'copyid').
  - `member_id` (INT, FK): Foreign key linking to the `Members` table (the borrower). (Renamed from 'memberid').
  - `issue_date` (DATETIME, NOT NULL): Date and time the book was issued. (Renamed from 'issued').
  - `due_date` (DATETIME, NOT NULL): Date the book is due for return (7 days after issue_date). (Renamed from 'returndue').
  - `return_date` (DATETIME, NULLABLE): Actual date and time the book was returned (NULL if not yet returned). (Renamed from 'returned').
  - `fine_amount` (DECIMAL(10,2), DEFAULT 0.00): Fine incurred for this transaction (0 if returned on time, calculated otherwise). (Renamed from 'fine').
  - `status` (VARCHAR(50), NOT NULL): Status of the transaction ('BORROWED', 'RETURNED', 'OVERDUE').

2.7. `Fines` Table (Implicitly suggested by fine collection, for detailed tracking of fine payments)

- **Purpose:** Tracks individual fines incurred by members, including their payment status. This allows for detailed reporting on outstanding and collected fines.
- **Columns:**
  - `id` (INT, PK, AUTO_INCREMENT): Unique identifier for the fine record.
  - `borrowing_transaction_id` (INT, FK, NOT NULL): Links to the specific `BorrowingTransactions` record that incurred this fine.
  - `member_id` (INT, FK, NOT NULL): Denormalized for easier lookup of fines per member.
  - `amount` (DECIMAL(10,2), NOT NULL): The fine amount.
  - `incurred_date` (DATETIME, NOT NULL): When the fine was assessed.
  - `is_paid` (BOOLEAN, DEFAULT FALSE): Status of the fine payment.
  - `paid_date` (DATETIME, NULLABLE): When the fine was paid.
  - `collected_by_user_id` (INT, FK, NULLABLE): ID of the Owner/Librarian who collected the fine.

---

3. Functional Requirements - Detailed Role Breakdown

3.1. Owner Role Functionalities

The 'Owner' role (also acts as 'Librarian') has full administrative control and oversight.

Dashboard & Reporting:

- **View User Statistics:**
  - Total members, active/inactive members, new members this month.
  - Member activity levels (high, medium, low).
  - Member growth trends over time.
  - Key insights: most active member, avg age, gender split, peak activity day.
  - Recent member activity log.
- **View Book Inventory (Library Assets):**
  - Total asset value, total books, total copies, utilization rate.
  - Breakdown of assets by category (e.g., Programming, Science books).
  - Asset valuation by category, including quantity, total value, avg value, utilization, growth.
  - Investment analysis (total investment, depreciation, net book value).
  - Valuation trends over time.
- **Manage Library Operations:**
  - **Book & Copy Management:**
    - Add new book titles (with details like ISBN, author, genre, price).
    - Update existing book details.
    - Add new physical copies for existing books.
    - Update status (available, damaged, lost) and rack location for individual copies.
    - Delete book titles (if no outstanding copies/transactions).
    - Delete specific book copies (if not issued).
  - **Rack Management (Implicit/Enhanced):**
    - Add new racks.

        - Update rack details (e.g., location description).
        - Delete racks (if no copies are assigned).
    - **User Management:**
        - View a list of all users (members and owners).
        - View detailed user profiles.
        - Toggle `is_paid_member` status for members.
        - Record membership fee payments for members.
        - Create new user accounts (for members or other owners).
        - Deactivate/Activate user accounts.
- **Circulation Management:**
    - **Process Book Return:**
        - Input `copy_identifier` to mark a book as returned.
        - Automatically calculate and apply fines if overdue (Rs. 5/- per day).
        - Update copy status to 'AVAILABLE'.
        - Record fine (if any).
    - **Monitor Overdue Books:**
        - View a list of all currently overdue books, showing borrower, due date, days overdue, and estimated fine.
- **View Financial Reports:**
    - Monthly revenue, expenses, net profit, profit margin.
    - Breakdown of revenue (membership fees, fines) and expenses.
    - Key financial ratios (profit margin, ROI).
    - Cash flow analysis.
    - Profit & Loss trend over time.
- **Monitor Revenue (Collections Reports):**
    - Total collections, breakdown by membership fees and fines.
    - Collection efficiency.
    - Outstanding dues summary.
    - Daily/weekly/monthly collection summaries.
    - Collection distribution (fees vs. fines) chart.

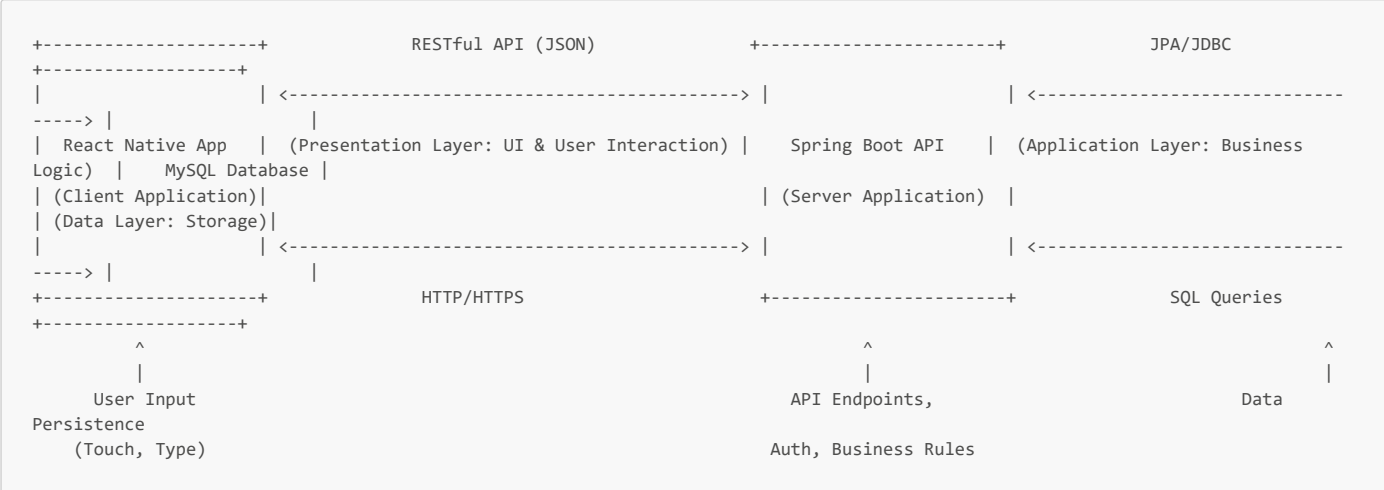
3.2. Member Role Functionalities

The 'Member' role focuses on personal library interactions.

- **Personal Dashboard:**
    - Welcome message with membership validity.
    - Summary of current borrowings, total books read, outstanding fines (if any), days until next return.
    - Quick actions (Search Books, My Books, Payment History, Account Settings).
    - Recent activity log.
- **Search for Books:**
    - Search by title, author, ISBN, or subject.
    - Filter by subject and availability.
    - View search results with basic book info and availability status.
- **Check Book Availability & Details:**
    - View detailed information for a specific book (description, ISBN, genre, price).
    - See overall availability status (Available, Limited, Unavailable).
    - View a list of all physical copies for a book, their `copy_identifier`, rack location, and individual status.
- **Borrow Books:**
    - Select an available copy of a book to borrow.
    - System validates if the member is a 'paid user'.
    - Borrowing initiates a `BorrowingTransaction` record, and updates `Copy` status to 'ISSUED'.
    - (Note: Actual physical handover/confirmation is done by Librarian).
- **View Borrowing History:**
    - Comprehensive list of all past and current borrowing transactions.
    - Includes issue date, due date, return date, status (Current, Returned, Overdue), and associated fines.
    - Filter history by date range, status, or subject.
- **View Personal Fines:**
    - See current outstanding fines, if any.
    - View details on how fines are calculated (Rs. 5/- per day overdue).
    - Tips to avoid future fines.
    - (Note: Members view, Owners collect).
- **View Payment History:**
    - See a comprehensive list of all membership fees and fine payments made.
    - Includes payment amount, date, type (fee/fine), and relevant details.
    - Summary of total fees/fines paid.
    - Next membership fee due date reminder.
- **Account Management:**
    - View and update personal profile information (name, email, phone).
    - Change account password.

4. Technical Architecture

The system employs a standard three-tier architecture to ensure separation of concerns, scalability, and maintainability.



- **React Native (Frontend):** Builds the cross-platform mobile application, handling all user interface elements, user interactions, and local state management. It communicates with the backend via RESTful API calls.
- **Spring Boot (Backend):** Serves as the application's brain. It's responsible for:
  - Exposing RESTful API endpoints.
  - Implementing all business logic (e.g., borrowing rules, fine calculation, data aggregation for reports).
  - Handling user authentication and role-based authorization using Spring Security and JWTs.
  - Interacting with the MySQL database via Spring Data JPA (Hibernate).
- **MySQL (Database):** A relational database management system used for persistent storage of all application data, including user details, book inventory, transactions, and financial records.
- **JWT Authentication:** JSON Web Tokens are used for secure, stateless authentication. Upon successful login, the Spring Boot API issues a JWT to the client. The client then includes this token in subsequent requests, allowing the backend to verify the user's identity and apply role-based access control without needing to query the database for every request.

5. API Endpoints - Detailed Documentation

All API endpoints will be prefixed with `/api`. Security will be enforced using JWT for authenticated endpoints.

5.1. Authentication & Common Endpoints

Method	URL Path	Description	Access	Request	Response (Success)	Business Logic/Validation
POST	/api/auth/login	Authenticate a user and issue a JWT.	Public	{"email": "...", "password": "..."}	{"token": "...", "userId": ..., "username": "...", "email": "...", "role": "...", "firstName": "...", "isPaidMember": true/false}	Validate credentials against Members table. If valid, generate JWT. Retrieve and return user details including role and isPaidMember status for client-side routing and display.
POST	/api/auth/register	Register a new MEMBER user.	Public	{"firstName": "...", "lastName": "...", "email": "...", "phone": "...", "password": "..."}	{"message": "User registered successfully.", "userId": ...} or {"errors": {"field": "message"}}	Validate input fields (e.g., email format, password strength). Hash password before storing. Create Members record with role='MEMBER', is_active=TRUE, is_paid_member=FALSE.
POST	/api/auth/forgot-password	Initiate password reset process.	Public	{"email": "..."}	{"message": "Password reset instructions sent to your email."}	Check if email exists in Members. If yes, trigger logic to send reset email (placeholder for now).

Method	URL Path	Description	Access	Request	Response (Success)	Business Logic/Validation
GET	/api/user/profile	Retrieve authenticated user's profile details.	AUTHENTICATED	(None)	{"id": ..., "username": "...", "email": "...", "firstName": "...", "lastName": "...", "phone": "...", "role": "...", "isPaidMember": true/false, "memberSince": "YYYY-MM-DD", "membershipStatus": "Active/Inactive", "nextPaymentDueDate": "YYYY-MM-DD"}	Retrieve user details based on JWT. Format memberSince, nextPaymentDueDate for display. Determine membershipStatus based on isPaidMember and is_active.
PUT	/api/user/profile	Update authenticated user's personal information.	AUTHENTICATED	{"firstName": "...", "lastName": "...", "phone": "..."}	{"message": "Profile updated successfully."}	Update first_name, last_name, phone in Members table for the authenticated user. Email cannot be changed via this endpoint.
PUT	/api/user/change-password	Change authenticated user's password.	AUTHENTICATED	{"currentPassword": "...", "newPassword": "..."}	{"message": "Password updated successfully."}	Verify currentPassword against stored password_hash. Validate newPassword against strength requirements. Hash and update password_hash in Members table. Invalid current password or weak new password results in error.

5.2. Member-Specific Endpoints

Method	URL Path	Description	Access	Request	Response (Success)
GET	/api/member/dashboard-summary	Get summary data for member dashboard.	MEMBER	(None)	{"userName": "...", "membershipExpires": "...", "booksCurrentlyBorrowed": ..., "totalBooksRead": ..., "outstandingFines": ..., "daysUntilNextReturn": ..., "recentActivity": [{"descriptio "...", "date": "..."}]}
GET	/api/books	Search and list books.	MEMBER, OWNER	Query Params: query (str), category (str: title/author/subject/isbn), subject (str), availability (boolean), page (int), size (int)	{"totalResults": ..., "books": [{"id": ..., "title": "...", "author": "...", "genre": "..." "isbn": "...", "price": ..., "totalCopies": ..., "availableCopies": ..., "overallAvailabilityStatus": "AVAILABLE/LIMITED/UNAVAILABLE"

Method	URL Path	Description	Access	Request	Response (Success)
GET	/api/books/{bookId}	Get details for a specific book.	MEMBER, OWNER	Path Variable: bookId (int)	{"id": ..., "title": "...", "author": "...", "isbn": "...", "genre": "...", "price": ..., "totalCopies": ..., "availableCopies": ..., "overallAvailabilityStatus": "...", "description": "...", "copiesSummary": [{"id": ..., "copyIdentifier": "...", "rack" "...", "status": "..."}], "relatedBooks": [{"id": ..., "title": "...", "author": "..."
GET	/api/books/{bookId}/copies	Get all copies for a specific book.	MEMBER, OWNER	Query Params: status (str: all/available/issued), rackId (int)	{"bookId": ..., "bookTitle": "...", "bookAuthor": "...", "totalCopies": ..., "availableCopiesCount": ..., "issuedCopiesCount": ..., "bookPrice": ..., "racks": [{"i ..., "rackNumber": "...", "description": "..."}], "copies [{"id": ..., "copyIdentifier": "...", "rackLocation": "...", "status": "...", "condition": "...", "addedDate": "...", "lastBorrowedDate": "...", "issuedToMemberName": "...", "dueDate": "..."}]}
POST	/api/member/borrow/{copyIdentifier}	Member requests to borrow a specific copy.	MEMBER	Path Variable: copyIdentifier (string)	{"message": "Book borrowed successfully."}
POST	/api/member/borrow/{transactionId}/renew	Member requests to renew a borrowed book.	MEMBER	Path Variable: transactionId (int)	{"message": "Renewal request submitted. Please visit the library desk to confirm."}
GET	/api/member/borrowing-history	View member's complete borrowing history.	MEMBER	Query Params: page (int), size (int), startDate (date), endDate (date), status (str: all/returned/current/overdue), subject (str)	{"totalBooksRead": ..., "currentlyBorrowedCount": ..., "successfullyReturnedCount": .. "totalFinesPaid": ..., "favoriteSubject": "...", "totalRecords": ..., "transactions": [{"id": ..., "bookTitle": "...", "bookAuthor "...", "issueDate": "...", "dueDate": "...", "returnDate": "...", "status": "CURRENT/RETURNED/OVERDUE", "fineAmount": ..., "fineStatus" "PAID/PENDING/N/A"}]}

Method	URL Path	Description	Access	Request	Response (Success)
GET	/api/member/fines/outstanding-summary	Get summary of member's outstanding fines.	MEMBER	(None)	{"hasOutstandingFines": true/false, "currentOutstandingAmount": ... "totalFinesPaid": ..., "lateReturnsCount": ..., "fineRatePerDay": ..., "fineCalculationDetails": {...} "tipsToAvoidFines": [...]}
GET	/api/member/fines	List specific outstanding/paid fines.	MEMBER	Query Params: isPaid (boolean: true/false), page (int), size (int)	[{"id": ..., "bookTitle": "..." "amount": ..., "incurredDate": "...", "daysLate": ..., "fineRate": ..., "isPaid": true/false}]
GET	/api/member/payment-history	View member's payment history.	MEMBER	Query Params: page (int), size (int), startDate (date), endDate (date), type (str: all/fee/fine), amountRange (str)	{"totalFeesPaid": ..., "totalFinesPaid": ..., "lastPaymentAmount": ..., "lastPaymentDate": "...", "totalOverallPaid": ..., "totalTransactions": ..., "nextMembershipDueDate": "...", "nextMembershipFeeAmount": ..., "payments": [{"id": ..., "type" "FEE/FINE", "description": "... "date": "...", "amount": ..., "transactionId": "...", "validUntil": "...", "paymentMethod": "...", "collectedBy": "..."}]}

5.3. Owner-Specific Endpoints

Method	URL Path	Description	Access	Request	Response (Success)
GET	/api/owner/dashboard-summary	Get summary data for owner dashboard.	OWNER	(None)	{"userName": "...", "currentDateTime": "..." "alerts": [...], "kpis": [...], "revenueTrendData [...], "keyBusinessMetri [...], "thisMonthSummary [...], "performanceTarge [...]}
GET	/api/owner/users	List all users with filters.	OWNER	Query Params: role (str: all/MEMBER/OWNER), isPaid (boolean), isActive (boolean), search (str: name/email), page (int), size (int)	{"totalUsers": ..., "use [{"id": ..., "firstName" "...", "lastName": "..." "email": "...", "role": "...", "isPaidMember": true/false, "lastPaymentDate": "YYYY DD", "isActive": true/false}]}
GET	/api/owner/users/{userId}	Get detailed user information.	OWNER	Path Variable: userId (int)	{"id": ..., "firstName": "...", "lastName": "..." "email": "...", "phone": "...", "role": "...", "isPaidMember": true/fal "lastPaymentDate": "..." "memberSince": "...", "isActive": true/false, "totalBooksBorrowed": .. "outstandingFines": ..., "recentActivity": [...]}

Method	URL Path	Description	Access	Request	Response (Success)
PUT	/api/owner/users/{userId}/update-paid-status	Toggle a member's paid status.	OWNER	{"isPaid": true/false} (if toggling payment status)	{"message": "Member paym status updated."}
POST	/api/owner/users/{userId}/record-payment	Record a payment for a member.	OWNER	{"amount": ..., "type": "MEMBERSHIP_FEE/FINE_PAYMENT", "periodCovered": "Oct 2023"} (type is 'MEMBERSHIP_FEE' for monthly payment, or 'FINE_PAYMENT' for fine collection)	{"message": "Payment rec successfully."}
POST	/api/owner/books	Add a new book title.	OWNER	{"title": "...", "author": "...", "isbn": "...", "genre": "...", "price": ..., "description": "..."}	{"message": "Book added successfully.", "bookId" ...}
PUT	/api/owner/books/{bookId}	Update book details.	OWNER	Path Variable: bookId (int) Request Body: {"title": "...", "author": "...", "genre": "...", "price": ..., "description": "..."}	{"message": "Book update successfully."}
DELETE	/api/owner/books/{bookId}	Delete a book title.	OWNER	Path Variable: bookId (int)	{"message": "Book delete successfully."}
POST	/api/owner/racks	Add a new rack.	OWNER	{"rackNumber": "...", "locationDescription": "..."}	{"message": "Rack added successfully.", "rackId" ...}
PUT	/api/owner/racks/{rackId}	Update rack details.	OWNER	Path Variable: rackId (int) Request Body: {"rackNumber": "...", "locationDescription": "..."}	{"message": "Rack update successfully."}
DELETE	/api/owner/racks/{rackId}	Delete a rack.	OWNER	Path Variable: rackId (int)	{"message": "Rack delete successfully."}
POST	/api/owner/book-copies	Add a new physical copy for a book.	OWNER	{"bookId": ..., "copyIdentifier": "...", "rackId": ..., "condition": "Good", "addedDate": "YYYY-MM-DD"}	{"message": "Book copy a successfully.", "copyId" ...}
PUT	/api/owner/book-copies/{copyId}	Update a book copy's details or status.	OWNER	Path Variable: copyId (int) Request Body: {"status": "AVAILABLE/LOST/DAMAGED", "rackId": ...}	{"message": "Book copy updated successfully."}
DELETE	/api/owner/book-copies/{copyId}	Delete a book copy.	OWNER	Path Variable: copyId (int)	{"message": "Book copy deleted successfully."}
POST	/api/owner/return	Process the return of a book copy.	OWNER	{"copyIdentifier": "...", "actualReturnDate": "YYYY-MM-DDTHH:MM:SS"} (Optional: memberId if needed to uniquely identify the transaction for the copy)	{"message": "Book return successfully.", "fineAmo ..., "fineId": ...}

Method	URL Path	Description	Access	Request	Response (Success)
GET	/api/owner/overdue-books	Get a list of all currently overdue books.	OWNER	Query Params: page (int), size (int)	{"totalOverdue": ..., "overdueBooks": [{"transactionId": ..., "bookTitle": "...", "copyIdentifier": "...", "memberName": "...", "borrowDate": "...", "dueDate": "...", "daysOverdue": ..., "estimatedFine": ...}]}
POST	/api/owner/fines/{fineId}/collect	Mark a fine as paid.	OWNER	Path Variable: fineId (int) Request Body: {"collectedByUserId": ...} (current authenticated owner's ID)	{"message": "Fine collec successfully."}
GET	/api/owner/reports/user-statistics	Get aggregated user statistics.	OWNER	Query Params: dateRange (str: month/quarter/year/all), memberStatus (str: all/active/inactive/new), activityLevel (str: all/high/medium/low), reportType (str: summary/detailed/trends)	{"totalMembers": ..., "activeMembers": ..., "newMembersThisMonth": . "retentionRate": ..., "avgBooksPerMember": ... "memberActivityStatistic [...], "memberGrowthTrendData": [...], "memberInsights": {...}, "engagementMetric {...}, "revenueAnalysis" {...}, "recentMemberActivity": [...]}
GET	/api/owner/reports/assets	Get asset reports (inventory valuation).	OWNER	Query Params: assetType (str: all/books/equipment/furniture), valuationMethod (str: cost/current/depreciated), ageFilter (str: all/new/recent/older), statusFilter (str: all/active/damaged/retired)	{"totalAssetValue": ..., "totalBooks": ..., "totalCopies": ..., "utilizationRate": ..., "assetCategories": [...] "assetPerformance": {... "investmentAnalysis": {. "maintenanceSchedule": { "valuationTrendData": [.
GET	/api/owner/reports/books-by-copies	Get book-wise copies report.	OWNER	Query Params: page (int), size (int), subject (str), utilization (str: high/medium/low/underutilized), copiesCount (str: single/few/many), sortBy (str: title/totalCopies/utilization/value), sortDir (str: asc/desc)	{"totalBookTitles": ..., "totalCopies": ..., "avgCopiesPerBook": ..., "availableCopies": ..., "issuedCopies": ..., "avgUtilizationOverall": "books": [{"id": ..., "title": "...", "author" "...", "subject": "...", "totalCopies": ..., "availableCopies": ..., "issuedCopies": ..., "utilizationPercentage": "bookValue": ..., "totalInvestment": ...}]
GET	/api/owner/reports/collections	Get financial collection reports.	OWNER	Query Params: startDate (date), endDate (date), type (str: all/fee/fine), period (str: daily/weekly/monthly/quarterly)	{"totalCollections": ... "membershipFeesTotal": . "fineCollectionsTotal": "collectionEfficiency": "outstandingDues": ..., "collectionSummary": [.. "analysisMetrics": {...} "outstandingAnalysis": { "performanceTargets": {. "collectionDistributionD [...]}

Method	URL Path	Description	Access	Request	Response (Success)
GET	/api/owner/reports/financial	Get comprehensive financial reports.	OWNER	Query Params: startDate (date), endDate (date), reportType (str: monthly/quarterly/yearly/custom), category (str: all/revenue/expenses/profit)	{"monthlyRevenue": ..., "monthlyExpenses": ..., "netProfit": ..., "profitMargin": ..., "totalAssets": ..., "revenueExpenseBreakdown {...}, "keyFinancialRati {...}, "cashFlowAnalysis {...}, "performanceTarge {...}, "profitAndLossTrendData" [...]}
GET	/api/owner/reports/{reportType}/export/pdf	Export reports as PDF.	OWNER	Query Params: Same as corresponding GET report endpoint.	File stream (PDF)
GET	/api/owner/reports/{reportType}/export/excel	Export reports as Excel.	OWNER	Query Params: Same as corresponding GET report endpoint.	File stream (Excel)

6. Development Guidelines

6.1. React Native (Frontend) Development Guidelines

- **Project Setup:**
  - Initialize with Expo CLI (`npx create-expo-app library-app --template blank-typescript`) or React Native CLI (`npx react-native init LibraryApp --template react-native-template-typescript`). Expo is often quicker for initial development and testing.
  - Install core libraries: `@react-navigation/native`, `@react-navigation/stack`, `@react-navigation/bottom-tabs`, `axios`, `react-native-keychain` (for secure token storage), and a UI toolkit like `react-native-paper` or `NativeBase` for consistent styling.
- **Folder Structure (Recommended):**

```
src/
├── api/           // Axios instances, API client functions
├── assets/        // Images, fonts, icons
├── components/    // Reusable UI components (e.g., Button, Card, TableHeader)
├── context/       // React Context for global state (AuthContext, UserContext)
├── hooks/         // Custom hooks (e.g., useAuth, useFetch)
├── navigation/    // React Navigation navigators (AuthNavigator, MemberNavigator, OwnerNavigator)
├── screens/       // All individual screens (LoginScreen, MemberDashboardScreen, OwnerDashboardScreen, etc.)
│   ├── Auth/
│   ├── Member/
│   └── Owner/
├── utils/         // Helper functions (date formatting, validation)
├── App.tsx        // Main application entry point
└── types.ts       // TypeScript interfaces for data structures (DTOs)
```

- **Role-Based UI Development:**

1. **Authentication Context (`AuthContext`):**
   - Create a `AuthContext` to store the user's login status (`isLoggedIn`), `role` (`MEMBER`/`OWNER`), `userId`, and `JWT token`.
   - Implement `login`, `logout` functions that update this context.
   - The `token` should be stored securely using `react-native-keychain` (preferred) or `AsyncStorage` (simpler for dev, less secure).
2. **Navigation (`react-navigation`):**
   - Use `createStackNavigator` for the main app flow.
   - Define separate navigators for `AuthStack` (Login, Register, Forgot Password) and `AppStack`.
   - Within `AppStack`, conditionally render `MemberNavigator` (using `createBottomTabNavigator` for Member Dashboard, Search, My Books, etc.) or `OwnerNavigator` (using `createDrawerNavigator` or `createBottomTabNavigator` + `createStackNavigator` for Dashboard, Reports, Management) based on the user's `role` from `AuthContext`.

```
// In AppNavigator.js
function AppNavigator() {
  const { user } = useContext(AuthContext); // Get user info from context

  return (
```

```
      <Stack.Navigator>
        {user ? (
          user.role === "OWNER" ? (
            <Stack.Screen
              name="OwnerApp"
              component={OwnerNavigator}
              options={{ headerShown: false }}
            />
          ) : (
            <Stack.Screen
              name="MemberApp"
              component={MemberNavigator}
              options={{ headerShown: false }}
            />
          )
        ) : (
          <Stack.Screen
            name="Auth"
            component={AuthNavigator}
            options={{ headerShown: false }}
          />
        )}
      </Stack.Navigator>
    );
  }
```

3. **Data Fetching & State Management:**
    - Use `axios` for API calls. Create an `api.js` file with an `axios` instance configured with the backend URL.
    - For authenticated requests, automatically attach the JWT to the `Authorization` header (`Bearer <token>`). This can be done with an `axios` interceptor.
    - Manage screen-specific state using `useState` and `useEffect` hooks.
    - For shared data or complex form states, consider `Zustand` (lightweight) or `Redux Toolkit` (more robust). For this project, `AuthContext` + local state should be sufficient initially.
4. **UI Components:**
    - Break down complex screens (like reports) into smaller, reusable components (e.g., `KPICard`, `FilterDropdown`, `DataTable`).
    - Use `FlatList` for efficient rendering of long lists (search results, history).
    - Chart libraries: `react-native-chart-kit` or `Victory Native` for charting needs in reports.

- **Security:**
    - Store JWT securely using `react-native-keychain`. `AsyncStorage` is *not* recommended for sensitive data in production.
    - Do not store sensitive user data (like passwords) on the client side.

6.2. Spring Boot (Backend) API Development Guidelines

- **Project Structure:** Follow standard Spring Boot project layout.

```
src/main/java/com/library/api/
├── config/        // Spring Security config, JWT config
├── controller/    // REST API endpoints (e.g., AuthController, BookController, MemberController, OwnerController)
├── model/         // JPA Entities (User, Book, Copy, etc.)
├── repository/    // Spring Data JPA repositories (e.g., UserRepository)
├── service/       // Business logic (e.g., UserService, BookService)
├── dto/           // Data Transfer Objects (request/response models)
├── exception/     // Custom exceptions and global exception handler
├── util/          // JWT utility class, password encoder
└── LibraryManagementApiApplication.java // Main class
```

- **JPA Entities & Relationships:**
    - Annotate model classes with `@Entity`, `@Table`, `@Id`, `@GeneratedValue`, `@Column`.
    - Define relationships using `@OneToMany`, `@ManyToOne`, `@OneToOne` with appropriate `fetch` types (`LAZY` generally preferred to avoid N+1 problems) and `cascade` options.
    - Use `@JoinColumn` for foreign keys.
    - Example for `Member` and `BorrowingTransaction`:

```
@Entity
@Table(name = "members")
@Data @NoArgsConstructor @AllArgsConstructor // Lombok
public class Member {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    @Column(unique = true)
    private String email;
    private String phone;
    private String passwordHash;
    @Enumerated(EnumType.STRING)
    private Role role; // Enum Role { MEMBER, OWNER }
    private Boolean isActive;
    private Boolean isPaidMember;
    private LocalDate lastPaymentDate; // Using LocalDate for DATE type

    // Relationships (example)
    @OneToMany(mappedBy = "member", cascade = CascadeType.ALL, orphanRemoval = true)
    private Set<BorrowingTransaction> borrowingTransactions = new HashSet<>();
}

@Entity
@Table(name = "borrowing_transactions")
@Data @NoArgsConstructor @AllArgsConstructor
public class BorrowingTransaction {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "copy_id", nullable = false)
    private Copy copy;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "member_id", nullable = false)
    private Member member;

    private LocalDateTime issueDate;
    private LocalDateTime dueDate;
    private LocalDateTime returnDate;
    private BigDecimal fineAmount;
    @Enumerated(EnumType.STRING)
    private TransactionStatus status; // Enum TransactionStatus { BORROWED, RETURNED, OVERDUE }
}
```

- **Spring Security & JWT:**
    1. **Dependencies:** Ensure `spring-boot-starter-security` and `jjwt-api`, `jjwt-impl`, `jjwt-jackson` (for JWT) are in `pom.xml`.
    2. **`SecurityConfig`:** Extend `WebSecurityConfigurerAdapter` (or configure `FilterChainProxy` for Spring Security 6+).
        - Configure `HttpSecurity` to define public vs. authenticated endpoints (`.antMatchers("/api/auth/**").permitAll()`, `.anyRequest().authenticated()`).
        - Add a custom `JwtAuthenticationFilter` that extracts JWT from requests, validates it, and sets `Authentication` in SecurityContext.
    3. **`JwtTokenProvider`:** A utility class to generate, validate, and extract claims from JWTs.
    4. **`UserDetailsService`:** Implement Spring Security's `UserDetailsService` to load user details (username, password, roles) from your `Members` table.
    5. **Password Hashing:** Use `BCryptPasswordEncoder` for hashing passwords. Inject it into your `AuthService` and `UserService`.
    6. **Role-Based Authorization:** Use `@PreAuthorize("hasRole('OWNER')")` or `@PreAuthorize("hasAnyRole('OWNER', 'MEMBER')")` annotations on controller methods or service methods to restrict access.
- **Layered Architecture:**
    - **Controllers (`@RestController`):** Handle HTTP requests and responses. They should be thin, primarily delegating to service layer. Convert DTOs to/from entities.
    - **Services (`@Service`):** Contain the core business logic. Interact with repositories. Transactions are typically managed at this layer.
    - **Repositories (`@Repository`):** Data access layer using Spring Data JPA. Define interfaces extending `JpaRepository` for basic CRUD and custom queries.
- **Data Transfer Objects (DTOs):**
    - Create separate DTOs for request bodies (e.g., `LoginRequest`, `RegisterRequest`, `BookDto`) and response bodies (e.g., `LoginResponse`, `BookResponse`, `MemberProfileResponse`).
    - Use libraries like MapStruct or ModelMapper for easy mapping between Entities and DTOs to avoid manual boilerplate.
- **Error Handling:**
    - Implement global exception handling using `@ControllerAdvice` and `@ExceptionHandler` to return consistent error responses (e.g., `{"timestamp": "...", "status": ..., "error": "...", "message": "...", "path": "..."}`).
    - Define custom exceptions for specific business errors (e.g., `ResourceNotFoundException`, `UnauthorizedException`, `InvalidInputException`).
- **Validation:** Use `@Valid` annotation on DTOs in controllers combined with Javax Validation annotations (`@NotNull`, `@Email`, `@Size`, etc.) on DTO fields.

- **Logging:** Use SLF4J with Logback (Spring Boot's default) for effective logging (`@Slf4j` from Lombok makes this easy).

6.3. MySQL Database Guidelines

- **Database Creation:** Ensure you have created the `library_db` database and a dedicated user with permissions.
- **`application.properties`:** Correctly configure `spring.datasource.url`, `username`, `password`.
- **DDL Auto:** For development, `spring.jpa.hibernate.ddl-auto=update` is convenient. For production, consider `validate` or `none` and use a migration tool like Flyway or Liquibase to manage schema changes explicitly. This allows for version control of your database schema and smoother deployments.
- **Indexes:** Add indexes to frequently queried columns (e.g., `email` in `Members`, foreign key columns) to improve query performance.
- **Data Integrity:** Foreign key constraints (`ON DELETE RESTRICT`, `ON UPDATE CASCADE`) are already defined in the SQL script to maintain data integrity.
