# Library Booking System: Database
# Week 4 Report – Integration

**Database CS A**

**Made by:**

| | |
|---|---|
| **Klara Ahinta Daniswara** | **24/532751/PA/22531** |
| **Daffa Aryza Pasya** | **24/532884/PA/22549** |
| **Shadiq Arif Aryasatya** | **24/536702/PA/22763** |

**DEPARTEMEN ILMU KOMPUTER dan ELEKTRONIKA**

**FAKULTAS MATEMATIKA dan ILMU PENGETAHUAN ALAM**

**UNIVERSITAS GADJAH MADA**

**YOGYAKARTA**

**2025**

## Tables of Content

## Objective

Connect the database with a web or mobile frontend; implement forms and simple search or report functions.

## Frontend

In our project, React was used as the frontend framework to provide an interactive interface for the Library Tracking System. This frontend serves as a bridge between the users (Staff and Members) and the Spring Boot REST API.

The React code can be accessed on our GitHub in the directory: week4_integration/frontend/src/LibraryDashboard.jsx.

The frontend is designed as a Single Page Application (SPA) that dynamically renders content based on the user's role and authentication status. It utilizes modern React hooks such as useState for data management and useEffect for automated API synchronization.

The system implements a dual-login mechanism within the LoginScreen component:

- Staff Login: Authenticates via email. Upon success, the user is granted "Staff" privileges, enabling administrative tools like adding books and managing fines.
- Member Login: Authenticates via a unique Member ID. Members are restricted to browsing the catalog and requesting book borrows.

The AdminDashboard component provides full CRUD (Create, Read, Update, Delete) capabilities for the library database:

- Entity Management: Forms are provided to add new Books, Book Copies, Staff, and Members directly into the database.
- Operational Controls: Staff can process book returns and issue or mark fines as paid.
- Request Processing: A notification system alerts staff to "Pending Borrow Requests" initiated by members, which can be approved or denied.
- Database Viewer: A specialized sub-component, AdminDashboardViewer, allows staff to view raw JSON data or filtered tables for all database entities.

The MemberDashboard simplifies the library experience for the end-user:

- Book Browser: Allows members to search the catalog by title, author, or ISBN and check the real-time availability of copies.
- Borrowing Requests: Members can submit a request for a specific Book Copy ID, which then appears in the Admin's notification queue.
- Personal History: Members can view their active borrows, due dates, and return history.

The frontend communicates with the backend through a centralized apiRequest helper function:

- RESTful Communication: It supports standard HTTP methods (GET, POST, PUT, DELETE) to interact with the Spring Boot controllers.
- Data Security: For display purposes, the frontend includes logic to strip or redact sensitive information, such as passwords, before rendering them in the UI.
- Error Handling: The system captures API error codes and translates them into user-friendly messages for the login and operation screens.

The UX features include:

- Theme Support: Includes a "Dark Mode" and "Light Mode" toggle to accommodate user preference.
- Real-time Search: Search bars across all dashboards allow for instant filtering of large datasets (books, members, or fines) without reloading the page.

## Backend

Our backend system is built using Java with the Spring Boot framework. It follows a layered architectural pattern to ensure a clean separation of concerns between web handling, business logic, and database persistence.

### Backend Structure Overview

Based on our project organization, the backend is structured into several key packages:

- Config: Contains configuration files like WebConfig.java to handle CORS (Cross-Origin Resource Sharing), allowing our React frontend to communicate with this API.
- Controllers: The entry point of the API. These classes handle incoming HTTP requests and return data to the client.
- Entities: These are the "Blueprints" or data models (e.g., book.java, member.java) that represent our database tables in Java code.
- Services: This layer contains the core business logic, such as calculating fine amounts or verifying if a book copy is available before allowing a borrow.
- Repositories: These interfaces act as the bridge to the database, performing the actual SQL operations (Save, Find, Delete).

### Layered Functionality Explanation

#### 1. Controllers (Web Layer)

The controllers act as the traffic directors for our API. They use Spring annotations like @RestController and @RequestMapping to define the API endpoints accessed by the frontend.

- Request Mapping: They map specific URLs (like /books or /members) to Java methods.
- Response Handling: They use ResponseEntity to return the correct HTTP status codes (e.g., 200 OK for success or 404 Not Found if a record doesn't exist).
- Validation: By using @Valid, the controllers ensure that incoming data—like a new book's ISBN—meets our required format before it ever reaches the database.

#### 2. Services (Business Logic Layer)

The services (e.g., bookservice.java) sit between the controllers and repositories. While the controller simply receives the request, the service performs the "thinking":

- Borrow Logic: When a borrowdetail is created, the service verifies if the member is active and if the book copy isn't already checked out.
- Fine Management: The fineservice calculates if a return is overdue and updates the fine entity status accordingly.
- Data Transformation: It ensures that data is properly formatted and business rules are followed before saving.

**3. Repositories & Entities (Data Layer)**

- Entities: Classes like book, member, and staff define the structure of our library's information. Each field in the class (like staffEmail) corresponds to a column in our SQL database.
- Repositories: Using Spring Data JPA, these interfaces (like bookrepository.java) provide built-in methods to perform CRUD operations without us having to write complex SQL queries manually.

**CRUD Logic Summary by Controller**

| Controller | Create Logic | Read Logic | Update/Delete Logic |
|---|---|---|---|
| BookController | Adds new master records via ISBN. | Supports ISBN lookup and keyword search. | Allows full metadata updates and ISBN-based removal. |
| BookCopyController | Tracks individual physical copies. | Lists all copies and tracks their specific status. | Updates "Availability" (Available/Borrowed). |
| BorrowDetailController | Links Member, Staff, and Book IDs. | Tracks history of all transactions. | Return Logic: Updates the returnDate for a specific borrow. |
| FineController | Triggered when books are | Allows searching by payment status. | Payment Logic: Marks fines as "Paid" with a specific date. |

| | | | |
|---|---|---|---|
| | returned late. | | |
| MemberController | Registers new users. | Searches for members by name or email. | Manages user profile changes and account closures. |
| StaffController | Sets up library employee accounts. | Features email-based lookup for staff logins. | Updates employment status and contact info. |

**Integration Example: The Borrowing Flow**

When a user clicks "Borrow" in our React frontend, the following sequence occurs:

1. React sends a POST request to /borrow.
2. BorrowDetailController receives the IDs and passes them to the borrowService.
3. BorrowService checks the bookcopyrepository to see if the book is available.
4. If available, it saves a new record via the borrowDetailRepository and returns the data to the frontend.

## Notes

- While the system is currently hosted in a local development environment, it is structurally complete. Both the React Frontend and Spring Boot Backend have been fully developed and integrated, ensuring that the user interface correctly communicates with the underlying database logic.
- The "bridge" between the two stacks is fully established. Using a centralized API utility in React, we have successfully mapped frontend actions (like clicking "Borrow") to backend Service methods. This ensures that data persistence is seamless and that the frontend state updates dynamically based on backend responses.