**Library Management System**

**Technical Design Document**

**System Architecture Overview**

**Technology Stack**

* **Framework:** Spring Boot 3.x
* **Java Version:** 17+
* **Build Tool:** Maven
* **Data Storage:** JSON files
* **Authentication:** Session-based with BCrypt
* **Documentation:** Swagger/OpenAPI
* **Testing:** JUnit 5 + Mockito

**Package Structure**

com.library.management/

├── LibraryApplication.java // @SpringBootApplication

├── config/

│ ├── SecurityConfig.java // Authentication setup

│ └── SwaggerConfig.java // API documentation

├── controller/

│ ├── AuthController.java // Login/register/logout

│ ├── BookController.java // Book CRUD + search

│ ├── UserController.java // User management

│ └── AdminController.java // Import/export

├── service/

│ ├── AuthService.java // Authentication logic

│ ├── BookService.java // Book business logic

│ ├── UserService.java // User management

│ └── RentalService.java // Rental operations

├── repository/

│ ├── UserRepository.java // Interface

│ ├── BookRepository.java // Interface

│ ├── RentalRepository.java // Interface

│ ├── impl/

│ │ ├── JsonUserRepository.java // JSON implementation

│ │ ├── JsonBookRepository.java // JSON implementation

│ │ └── JsonRentalRepository.java // JSON implementation

├── model/

│ ├── User.java // Entity

│ ├── Book.java // Entity

│ ├── Rental.java // Entity

│ └── dto/

│ ├── LoginRequest.java // Request DTOs

│ ├── BookSearchResponse.java // Response DTOs

│ └── PagedResponse.java // Pagination wrapper

└── util/

├── JsonFileHandler.java // File I/O utilities

└── UuidGenerator.java // ID generation

**Core Classes Design**

**Entity Classes**

**User.java**

public class User {

private String id; // UUID

private String username; // Unique, case-insensitive

private String passwordHash; // BCrypt hashed

private String email; // Unique, case-insensitive

private UserRole role; // ADMIN, USER

private boolean protected; // Cannot delete if true

private boolean mustChangePassword;

// constructors, getters, setters

}

**Book.java**

public class Book {

private String id; // UUID

private String title; // Required

private String author; // Required

private String genre; // Optional

private BookStatus status; // AVAILABLE, RENTED

// constructors, getters, setters

}

**Rental.java**

public class Rental {

private String id; // UUID

private String userId; // Foreign key

private String bookId; // Foreign key

private LocalDateTime rentDate;

private RentalStatus status; // ACTIVE, CLOSED

private LocalDateTime returnDate; // Null until returned

// constructors, getters, setters

}

**Repository Interface Pattern**

**BookRepository.java (Interface)**

public interface BookRepository {

List<Book> findAll();

Optional<Book> findById(String id);

Book save(Book book);

void delete(String id);

List<Book> findByTitleContaining(String title);

List<Book> findByAuthorContaining(String author);

boolean existsByTitleAndAuthor(String title, String author);

}

**JsonBookRepository.java (Implementation)**

@Repository

public class JsonBookRepository implements BookRepository {

private final JsonFileHandler<Book> fileHandler;

public JsonBookRepository() {

this.fileHandler = new JsonFileHandler<>("data/books.json", Book.class);

}

// Implementation methods...

}

**Data Storage Strategy**

**File Locations**

* data/users.json - User accounts
* data/books.json - Book catalog
* data/rentals.json - Rental records

**JSON Structure Examples**

**users.json**

[

{

"id": "admin-uuid-here",

"username": "admin",

"passwordHash": "$2a$10$...",

"email": "admin@library.com",

"role": "ADMIN",

"protected": true,

"mustChangePassword": true

}

]

**File I/O Strategy**

* Atomic writes (temp file → rename)
* Read entire file into memory for operations
* Write entire collection back to file
* Single writer lock for thread safety

**Authentication Flow**

**Login Process**

1. User submits username/password
2. AuthService validates credentials
3. BCrypt verifies password hash
4. Session created with user ID and role
5. Return success/failure response

**Authorization Strategy**

* Session-based authentication
* Role checking in controllers via annotations
* Manual role validation in service layer

**API Design Patterns**

**REST Endpoint Structure**

* POST /auth/login - Authentication
* GET /books?q=&page=0&size=20 - Search with pagination
* POST /books/{id}/rent - Action-based endpoints
* GET /admin/export - Admin-only operations

**Response Format Standards**

**Paginated Response Envelope (Required for ALL list endpoints):**

@Data

public class PagedResponse<T> {

private List<T> content;

private int page;

private int size;

private long total;

private int totalPages;

// Standard pagination defaults

public static final int DEFAULT\_PAGE = 0;

public static final int DEFAULT\_SIZE = 20;

public static final int MAX\_SIZE = 100; // Prevent excessive requests

}

**Mandatory Usage:**

* GET /books → PagedResponse<Book>
* GET /users → PagedResponse<User>
* GET /rentals → PagedResponse<Rental>
* All search and list operations must use this format

**Error Response Format:**

{

"error": "Validation failed",

"details": ["Title cannot be blank"]

}

**Security Implementation**

**Password Security**

* BCrypt with strength 10
* Minimum 8 characters + letter/number requirement
* Force password change for default admin

**Session Management**

* HttpSession for user state
* Role-based access control
* Logout clears session

**Business Logic Implementation**

**Rental Rules**

* Check book availability before rental
* Enforce 5-rental limit per user
* Only renter or admin can return books
* Update book status automatically

**Import/Export Logic**

**Export Operation:**

* Books catalog only (JSON array format)
* Returns complete book list with current status
* Downloaded as timestamped file: library\_export\_YYYY-MM-DD.json
* Uses atomic write (temp file → rename) for data integrity

**Import Operation:**

// Import behavior specification

public class ImportService {

/\*\*

\* Import books with strict rules:

\* - Append-only operation (never replaces existing catalog)

\* - All imported books default to AVAILABLE status

\* - Skip duplicates using case-insensitive (title + author) comparison

\* - File size limit: 10MB maximum

\* - Return summary: { added: int, skipped: int, errors: String[] }

\*/

public ImportSummary importBooks(MultipartFile file) {

// Implementation with atomic write (temp → rename)

}

}

**Duplicate Detection Rules:**

* Case-insensitive comparison of title AND author fields
* Whitespace trimmed before comparison
* Example: "Clean Code" by "Robert Martin" equals "clean code" by "robert martin"

**Error Handling Strategy**

**Validation Approach**

* Controller-level input validation
* Service-level business rule validation
* Repository-level data constraints

**Exception Mapping**

* 400: Validation errors
* 401: Authentication required
* 403: Authorization denied
* 404: Resource not found
* 409: Business rule conflicts

**Testing Strategy**

**Unit Testing Focus**

* Service layer business logic
* Repository implementations with mock data
* Controller validation logic
* Authentication/authorization flows

**Mock Strategy**

* Mock repositories in service tests
* Mock external dependencies
* Test with in-memory data structures

**Configuration Requirements**

**application.properties**

# Server configuration

server.port=8080

# Session configuration

server.servlet.session.timeout=30m

# Library-specific settings

library.data.path=data/

library.rental.max-per-user=5

# Security settings

spring.security.require-ssl=false

# Logging

logging.level.com.library.management=INFO

logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %msg%n

# Swagger/OpenAPI

springdoc.api-docs.path=/v3/api-docs

springdoc.swagger-ui.path=/swagger-ui.html

**Spring Boot Annotations Usage**

* @SpringBootApplication - Main class
* @RestController - API endpoints
* @Service - Business logic
* @Repository - Data access
* @Configuration - Configuration classes