**Library Management System**

**Development Sequence**

**w/ SCHEDULE**

**Phase 1: Project Foundation (30 minutes)**

**Step 1.1: Initialize Spring Boot Project**

# Using Spring Initializr or IntelliJ

- Project: Maven

- Language: Java 17

- Spring Boot: 3.x

- Dependencies: Spring Web, Spring Boot DevTools

- Package: com.library.management

**Step 1.2: Create Basic Package Structure**

Step 1.2: Create Basic Package Structure

src/main/java/com/example/librarymanagementsystem/

├── LibraryManagementSystemApplication.java

├── config/ # SecurityConfig, SwaggerConfig

├── controller/ # AuthController, BookController, UserController, AdminController

├── model/ # domain entities

│ ├── dto/ # PagedResponse, LoginRequest, RegisterRequest, BookSearchResponse, ImportSummary

│ └── enums/ # UserRole, BookStatus, RentalStatus

├── repository/ # interfaces

│ └── impl/ # JsonUserRepository, JsonBookRepository, JsonRentalRepository

├── service/ # AuthService, BookService, UserService, RentalService

└── util/ # JsonFileHandler, UuidGenerator

**Step 1.3: Add Required Dependencies**

<!-- Add to pom.xml -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-crypto</artifactId>

</dependency>

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

<version>2.0.0</version>

</dependency>

**Phase 2: Core Data Models (45 minutes)**

**Step 2.1: Create Enums**

// UserRole.java

public enum UserRole { ADMIN, USER }

// BookStatus.java

public enum BookStatus { AVAILABLE, RENTED }

// RentalStatus.java

public enum RentalStatus { ACTIVE, CLOSED }

**Step 2.2: Create Entity Classes**

1. **User.java** - Complete entity with all fields
2. **Book.java** - Complete entity with all fields
3. **Rental.java** - Complete entity with all fields

**Step 2.3: Create DTO Classes**

1. **LoginRequest.java** - username, password
2. **RegisterRequest.java** - username, email, password
3. **PagedResponse.java** - Generic pagination wrapper
4. **BookSearchResponse.java** - Book search results
5. **ImportSummary.java** - Import operation results

**Phase 3: File I/O Foundation (60 minutes)**

**Step 3.1: Create Utility Classes**

// JsonFileHandler.java - Generic JSON file operations

public class JsonFileHandler<T> {

public List<T> readFromFile(String filePath, Class<T> clazz);

public void writeToFile(String filePath, List<T> data);

public void ensureFileExists(String filePath);

}

// UuidGenerator.java - ID generation utility

public class UuidGenerator {

public static String generate();

}

**Step 3.2: Create Data Directory**

# Create in project root

mkdir data

touch data/users.json

touch data/books.json

touch data/rentals.json

**Step 3.3: Initialize Default Data**

// Bootstrap default admin user in users.json

[

{

"id": "admin-uuid-001",

"username": "admin",

"passwordHash": "$2a$10$...", // BCrypt hash of "admin123"

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

"role": "ADMIN",

"protected": true,

"mustChangePassword": true

}

]

**Phase 4: Repository Layer (90 minutes)**

**Step 4.1: Create Repository Interfaces**

// UserRepository.java

public interface UserRepository {

List<User> findAll();

Optional<User> findById(String id);

Optional<User> findByUsername(String username);

Optional<User> findByEmail(String email);

User save(User user);

void delete(String id);

boolean existsByUsername(String username);

boolean existsByEmail(String email);

}

**Repeat for:** BookRepository.java, RentalRepository.java

**Step 4.2: Implement JSON Repository Classes**

1. **JsonUserRepository.java** - All CRUD operations
2. **JsonBookRepository.java** - All CRUD + search operations
3. **JsonRentalRepository.java** - All CRUD + user/book filtering

**Step 4.3: Test Repository Layer**

// Quick manual test in main() method

public static void main(String[] args) {

UserRepository userRepo = new JsonUserRepository();

List<User> users = userRepo.findAll();

System.out.println("Users loaded: " + users.size());

}

**Phase 5: Service Layer (2 hours)**

**Step 5.1: Create AuthService**

@Service

public class AuthService {

private final UserRepository userRepository;

private final BCryptPasswordEncoder passwordEncoder;

// Constructor injection

// login(), register(), validateUser() methods

}

**Step 5.2: Create BookService**

@Service

public class BookService {

private final BookRepository bookRepository;

private final RentalRepository rentalRepository;

// All book CRUD operations

// Search functionality with pagination

// Business rule validation

}

**Step 5.3: Create UserService & RentalService**

* **UserService** - User management operations
* **RentalService** - Rental/return operations with business rules

**Step 5.4: Test Service Layer**

Write basic unit tests for each service method using Mockito

**Phase 6: Basic Controllers (90 minutes)**

**Step 6.1: Create AuthController**

@RestController

@RequestMapping("/auth")

public class AuthController {

private final AuthService authService;

@PostMapping("/login")

@PostMapping("/register")

@PostMapping("/logout")

}

**Step 6.2: Create BookController**

@RestController

@RequestMapping("/books")

public class BookController {

private final BookService bookService;

@GetMapping // Search with pagination

@PostMapping("/{id}/rent")

@PostMapping("/{id}/return")

}

**Step 6.3: Test Basic Endpoints**

Use Postman or curl to test:

* User registration
* User login
* Book listing
* Basic rental operations

**Phase 7: Authentication & Security (2 hours)**

**Step 7.1: Add Session Management**

@RestController

public class AuthController {

@PostMapping("/auth/login")

public ResponseEntity<?> login(@RequestBody LoginRequest request,

HttpServletRequest httpRequest) {

// Validate credentials

// Create session

// Return success response

}

}

**Step 7.2: Add Role-Based Access Control**

// Create authentication helper

@Component

public class AuthHelper {

public User getCurrentUser(HttpServletRequest request);

public boolean isAdmin(HttpServletRequest request);

public boolean canAccessResource(String userId, HttpServletRequest request);

}

**Step 7.3: Secure All Endpoints**

Add authentication checks to all controllers using AuthHelper

**Phase 8: Admin Features (90 minutes)**

**Step 8.1: Create AdminController**

@RestController

@RequestMapping("/admin")

public class AdminController {

@GetMapping("/export")

public ResponseEntity<byte[]> exportBooks();

@PostMapping("/import")

public ResponseEntity<ImportSummary> importBooks(@RequestParam("file") MultipartFile file);

}

**Step 8.2: Create UserController**

@RestController

@RequestMapping("/users")

public class UserController {

@GetMapping("/me") // User's own profile

@PutMapping("/me") // Edit own profile

@GetMapping("") // Admin: list all users

@PutMapping("/{id}") // Admin: edit user

@DeleteMapping("/{id}") // Admin: delete user

@PostMapping("/{id}/promote") // Admin: change role

}

**Step 8.3: Implement Import/Export Logic**

* Export: Convert books to JSON, return as downloadable file
* Import: Parse uploaded JSON, validate, merge with existing data

**Phase 9: Error Handling & Validation (60 minutes)**

**Step 9.1: Create Global Exception Handler**

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ValidationException.class)

public ResponseEntity<?> handleValidation(ValidationException e);

@ExceptionHandler(UnauthorizedException.class)

public ResponseEntity<?> handleUnauthorized(UnauthorizedException e);

// Handle all major exception types

}

**Step 9.2: Add Input Validation**

* Controller-level validation using annotations
* Service-level business rule validation
* Custom validators for complex rules

**Step 9.3: Standardize Error Responses**

Create consistent error response format across all endpoints

**Phase 10: Documentation & Testing (90 minutes)**

**Step 10.1: Configure Swagger**

@Configuration

@OpenAPIDefinition(info = @Info(title = "Library Management API", version = "1.0"))

public class SwaggerConfig {

// Swagger configuration

}

**Step 10.2: Add API Documentation**

* Document all endpoints with @Operation annotations
* Add request/response examples
* Document authentication requirements

**Step 10.3: Create Unit Tests**

@ExtendWith(MockitoExtension.class)

class BookServiceTest {

@Mock private BookRepository bookRepository;

@InjectMocks private BookService bookService;

// Test all service methods

}

**Step 10.4: Integration Testing**

Test complete workflows:

* User registration → login → rent book → return book
* Admin import/export operations
* Authentication and authorization flows

**Phase 11: Final Polish (60 minutes)**

**Step 11.1: Add Pagination Everywhere**

Ensure all list endpoints return paginated responses

**Step 11.2: Performance Optimization**

* Add caching where appropriate
* Optimize file I/O operations
* Add proper logging

**Step 11.3: Final Testing**

* Test all user scenarios from BRD
* Verify all business rules from Scope Freeze
* Test error conditions and edge cases

**Daily Breakdown Suggestion**

**Day 1 (6-8 hours):**

* Phases 1-4: Foundation, models, file I/O, repositories
* Get data persistence working

**Day 2 (6-8 hours):**

* Phases 5-7: Services, controllers, authentication
* Get basic API working with auth

**Day 3 (4-6 hours):**

* Phases 8-11: Admin features, error handling, testing, polish
* Complete system ready for demo

**Key Success Checkpoints**

**End of Day 1:** Can save/load users and books from JSON files **End of Day 2:** Can register, login, and perform basic book operations  
**End of Day 3:** Complete system with import/export and full authentication

**Emergency Fallback:** If behind schedule, skip user management features and focus on core book catalog + basic auth