

Entity, Service, Repository Architecture - Detailed Guide

Overview

This document provides a comprehensive explanation of how Entity, Service, and Repository layers work together in the Library Management System, including validation mechanisms.

PART 1: ENTITY LAYER

What is an Entity?

An **Entity** is a Java class that represents a table in the database. It maps to relational database tables and holds data with business meaning. Each entity represents a core concept in the system.

Entities in Your Project

1. User Entity - Represents a library member

```
@Entity                                     // Marks this class as a JPA entity
@Table(name = "users")                      // Maps to "users" table
@Data                                         // Lombok: auto-generates getters/setters
@NoArgsConstructor                           // Lombok: generates no-arg constructor
public class User {
```



```
    @Id                                         // Primary key
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Auto-increment ID
    private Long id;
```



```
    @Column(name = "username", length = 50, unique = true, nullable = false)
    // Constraints:
    // - unique = true: No two users can have same username (database constraint)
    // - nullable = false: Username is required (NOT NULL)
    // - length = 50: Max 50 characters
    private String username;
```



```
    @Column(name = "password", length = 255, nullable = false)
    // Password stored as encrypted string
    private String password;
```



```
    @Column(name = "full_name", length = 100, nullable = false)
    private String fullName;
```

```

@Column(name = "email", length = 100, unique = true, nullable = false)
// Email must be unique and required
private String email;

@ManyToMany(fetch = FetchType.EAGER)
// A User can have MULTIPLE roles (e.g., both ADMIN and USER)
// A Role can be assigned to MULTIPLE users
// fetch = FetchType.EAGER: Load roles immediately with user

@JoinTable(
    name = "user_roles", // Junction table name
    joinColumns = @JoinColumn(name = "user_id"),
    inverseJoinColumns = @JoinColumn(name = "role_id")
)
// Junction table structure:
// user_roles (user_id, role_id) - links users to their roles
private Set<Role> roles = new HashSet<>();
}

```

Database Table Created:

```

users (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(50) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    full_name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL
)

```

2. Role Entity - Represents user roles/permissions

```

@Entity
@Table(name = "roles")
@Data
@NoArgsConstructor
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name", length = 50, unique = true, nullable = false)
    // Role name must be unique: Can only have one "ADMIN" role, etc.
    private String name; // Examples: "ADMIN", "USER"
}

```

Database Table:

```
roles (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50) UNIQUE NOT NULL
)

-- Example data:
-- 1, ADMIN
-- 2, USER
```

Junction Table for Many-to-Many:

```
user_roles (
    user_id BIGINT,
    role_id BIGINT,
    PRIMARY KEY (user_id, role_id),
    FOREIGN KEY (user_id) REFERENCES users(id),
    FOREIGN KEY (role_id) REFERENCES roles(id)
)

-- Example:
-- 1, 1  (User 1 has role 1 = ADMIN)
-- 1, 2  (User 1 also has role 2 = USER)
-- 2, 2  (User 2 has role 2 = USER)
```

3. Book Entity - Represents a library book

```
@Entity
@Table(name = "books")
@Data
@NoArgsConstructor
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String title;           // Book title (required)

    @Column(nullable = false)
    private String author;          // Author name (required)

    @Column(unique = true, nullable = false)
    private String isbn;            // Unique ISBN code (required, unique)
```

```

@Column(nullable = false)
private String genre;           // Book genre/category (required)

@Column(nullable = false)
private Integer quantity;      // Number of copies available (required)

@Column(nullable = false)
private Boolean isAvailable;   // Can this book be borrowed? (required, default true)
}

```

Database Table:

```

books (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    author VARCHAR(255) NOT NULL,
    isbn VARCHAR(255) UNIQUE NOT NULL,
    genre VARCHAR(255) NOT NULL,
    quantity INT NOT NULL,
    is_available BOOLEAN NOT NULL
)

```

4. BorrowRecord Entity - Represents a borrowing transaction

```

@Entity
@Table(name = "borrow_records")
@Data
@NoArgsConstructor
public class BorrowRecord {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.EAGER)
    // Many BorrowRecords + One User
    // A user can borrow multiple books (multiple records)
    // Each borrow record belongs to ONE user

    @JoinColumn(name = "user_id", nullable = false)
    // Foreign key to users table
    private User user;

    @ManyToOne(fetch = FetchType.EAGER)

```

```

// Many BorrowRecords + One Book
// A book can be referenced in multiple borrow records (borrowed multiple times)
// Each borrow record references ONE book

@JoinColumn(name = "book_id", nullable = false)
// Foreign key to books table
private Book book;

@Column(name = "borrow_date", nullable = false)
private LocalDate borrowDate; // When was it borrowed? (required)

@Column(name = "return_deadline", nullable = false)
private LocalDate returnDeadline; // By when should it be returned? (required)

@Column(name = "actual_return_date")
private LocalDate actualReturnDate; // When was it actually returned? (optional)

@Column(name = "status", length = 20, nullable = false)
private String status; // BORROWED, RETURNED, OVERDUE, RETURNED_LATE (required)
}

```

Database Table:

```

borrow_records (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    user_id BIGINT NOT NULL,
    book_id BIGINT NOT NULL,
    borrow_date DATE NOT NULL,
    return_deadline DATE NOT NULL,
    actual_return_date DATE,
    status VARCHAR(20) NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id),
    FOREIGN KEY (book_id) REFERENCES books(id)
)

```

PART 2: REPOSITORY LAYER

What is a Repository?

A **Repository** is an interface that handles database operations (CRUD - Create, Read, Update, Delete). It abstracts the database layer, so services don't need to know SQL or database details.

Spring Data JPA automatically provides implementations for these interfaces.

Repositories in Your Project

1. UserRepository - Data access for Users

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // JpaRepository provides:
    // - save(User): Insert or update
    // - findById(Long): Get user by ID
    // - findAll(): Get all users
    // - delete(User): Delete user
    // - deleteById(Long): Delete by ID
    // - count(): How many users exist

    // Custom queries specific to User:

    Optional<User> findByUsername(String username);
    // Finds user by username
    // Optional: Could be empty if user doesn't exist
    // Usage: repo.findByUsername("john").orElseThrow(...)

    boolean existsByUsername(String username);
    // Check if username already exists (for validation)
    // Returns: true if exists, false otherwise

    boolean existsByEmail(String email);
    // Check if email already exists (for validation)
}
```

How Spring Data JPA generates these: - Method name: `findByUsername`
- `find` = SELECT query - `By` = WHERE clause - `Username` = field to match -
Spring JPA generates: `SELECT * FROM users WHERE username = ?`

2. BookRepository - Data access for Books

```
@Repository
public interface BookRepository extends JpaRepository<Book, Long> {
    // Inherited from JpaRepository:
    // - save(Book), findById(Long), findAll(), delete(), etc.

    // Custom queries:

    Optional<Book> findByIsbn(String isbn);
    // Find book by ISBN (book identifier)
    // SELECT * FROM books WHERE isbn = ?
```

```

List<Book> findByIsAvailable(Boolean isAvailable);
// Find all available (or unavailable) books
// SELECT * FROM books WHERE is_available = true

List<Book> findByTitleContainingIgnoreCaseOrAuthorContainingIgnoreCase(
    String title, String author);
// Search books by title OR author (case-insensitive)
// SELECT * FROM books
// WHERE LOWER(title) LIKE CONCAT('%', ?, '%')
// OR LOWER(author) LIKE CONCAT('%', ?, '%')
}

```

3. BorrowRepository - Data access for Borrow Records

```

@Repository
public interface BorrowRepository extends JpaRepository<BorrowRecord, Long> {
    // Inherited: save(), findById(), findAll(), delete(), etc.

    // Custom queries:

    List<BorrowRecord> findByUser(User user);
    // Get all borrow records for a specific user
    // SELECT * FROM borrow_records WHERE user_id = ?

    List<BorrowRecord> findByStatus(String status);
    // Find records with specific status (BORROWED, OVERDUE, etc.)
    // SELECT * FROM borrow_records WHERE status = ?

    List<BorrowRecord> findByUserOrderByBorrowDateDesc(User user);
    // Get user's borrow history, sorted by date (newest first)
    // SELECT * FROM borrow_records
    // WHERE user_id = ?
    // ORDER BY borrow_date DESC
}

```

4. RoleRepository - Data access for Roles

```

@Repository
public interface RoleRepository extends JpaRepository<Role, Long> {
    // Inherited: save(), findById(), findAll(), delete(), etc.

    Optional<Role> findByName(String name);
    // Find role by name (ADMIN, USER, etc.)
}

```

```
// SELECT * FROM roles WHERE name = ?  
}
```

PART 3: SERVICE LAYER

What is a Service?

A **Service** contains **business logic**. It:

- Uses repositories to access/modify data
- Enforces business rules
- Validates data before saving
- Handles transactions
- Orchestrates complex operations across multiple entities

Services in Your Project

1. BookService - Business logic for Books

```
@Service  
@RequiredArgsConstructor  
public class BookService {  
  
    private final BookRepository bookRepository; // Dependency injection  
  
    // READ Operations  
  
    public List<Book> getAllBooks() {  
        // Get all books from database  
        return bookRepository.findAll(); // SELECT * FROM books  
    }  
  
    public Optional<Book> getBookById(Long id) {  
        // Get single book by ID  
        return bookRepository.findById(id); // SELECT * FROM books WHERE id = ?  
    }  
  
    public List<Book> getAvailableBooks() {  
        // Get only books that can be borrowed  
        return bookRepository.findByIsAvailable(true);  
        // SELECT * FROM books WHERE is_available = true  
    }  
  
    public List<Book> searchBooks(String keyword) {  
        // Search books by title or author (case-insensitive)  
        return bookRepository.findByTitleContainingIgnoreCaseOrAuthorContainingIgnoreCase(  
            keyword, keyword);  
    }  
}
```

```

// WRITE Operations with @Transactional
// @Transactional = Auto commit on success, rollback on error

@Transactional
public Book saveBook(Book book) {
    // Save new book to database
    // Validation: Database constraints ensure:
    // - title is not null
    // - author is not null
    // - isbn is unique
    // - isbn is not null
    // - genre is not null
    // - quantity is not null

    return bookRepository.save(book);
    // INSERT INTO books (...) VALUES (...)
    // OR UPDATE books SET ... WHERE id = ?
}

@Transactional
public Book updateBook(Long id, Book bookDetails) {
    // Update existing book
    Book book = bookRepository.findById(id)
        .orElseThrow(() -> new RuntimeException("Book not found"));

    // Update all fields
    book.setTitle(bookDetails.getTitle());
    book.setAuthor(bookDetails.getAuthor());
    book.setIsbn(bookDetails.getIsbn());
    book.setGenre(bookDetails.getGenre());
    book.setQuantity(bookDetails.getQuantity());
    book.setIsAvailable(bookDetails.getIsAvailable());

    return bookRepository.save(book);
}

@Transactional
public void deleteBook(Long id) {
    // Delete book from database
    bookRepository.deleteById(id);
    // DELETE FROM books WHERE id = ?
}

@Transactional
public void updateAvailability(Long bookId, boolean isAvailable) {
    // Update if book is available (used by BorrowService)
}

```

```

        Book book = bookRepository.findById(bookId)
            .orElseThrow(() -> new RuntimeException("Book not found"));
        book.setIsAvailable(isAvailable);
        bookRepository.save(book);
    }
}

```

2. BorrowService - Business logic for Borrowing

```

@Service
@RequiredArgsConstructor
public class BorrowService {

    private final BorrowRepository borrowRepository;
    private final BookService bookService; // Dependency injection

    @Transactional
    public BorrowRecord borrowBook(User user, Book book) {
        // Business logic: Check if book is available
        if (!book.getIsAvailable()) {
            throw new RuntimeException("Book is not available");
        }

        // Mark book as unavailable
        book.setIsAvailable(false);
        bookService.saveBook(book); // Update in database

        // Create borrow record
        LocalDate borrowDate = LocalDate.now();
        LocalDate returnDeadline = borrowDate.plusDays(14); // 14-day loan period

        BorrowRecord borrowRecord = new BorrowRecord(
            user,
            book,
            borrowDate,
            returnDeadline
        );
        borrowRecord.setStatus("BORROWED");

        return borrowRepository.save(borrowRecord);
        // INSERT INTO borrow_records (...) VALUES ...
    }

    @Transactional

```

```

public BorrowRecord returnBook(Long borrowRecordId) {
    // Fetch the borrow record
    BorrowRecord record = borrowRepository.findById(borrowRecordId)
        .orElseThrow(() -> new RuntimeException("Borrow record not found"));

    // Set actual return date
    record.setActualReturnDate(LocalDate.now());

    // Check if returned late (after deadline)
    if (LocalDate.now().isAfter(record.getReturnDeadline())) {
        record.setStatus("RETURNED_LATE");
    } else {
        record.setStatus("RETURNED");
    }

    // Make book available again
    Book book = record.getBook();
    book.setIsAvailable(true);
    bookService.saveBook(book);

    return borrowRepository.save(record);
    // UPDATE borrow_records SET ... WHERE id = ?
}
}

public List<BorrowRecord> getUserBorrowHistory(User user) {
    // Get all borrow records for a user (sorted by date)
    return borrowRepository.findByUserOrderByBorrowDateDesc(user);
    // SELECT * FROM borrow_records
    // WHERE user_id = ?
    // ORDER BY borrow_date DESC
}

public List<BorrowRecord> getActiveBorrows() {
    // Get all books currently borrowed (not returned)
    return borrowRepository.findByStatus("BORROWED");
    // SELECT * FROM borrow_records WHERE status = 'BORROWED'
}

@Transactional
public void updateOverdueRecords() {
    // Business logic: Check if any borrowed books are overdue
    List<BorrowRecord> activeRecords = borrowRepository.findByStatus("BORROWED");
    LocalDate today = LocalDate.now();

    for (BorrowRecord record : activeRecords) {
        if (today.isAfter(record.getReturnDeadline())) {

```

```

        record.setStatus("OVERDUE"); // Mark as overdue if past deadline
        borrowRepository.save(record);
    }
}
}
}

```

3. CustomUserDetailsService - Spring Security Integration

```

@Service
@RequiredArgsConstructor
public class CustomUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {

        // Fetch user from database
        User user = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException(
                "User not found: " + username));

        // Convert roles to authorities (Spring Security format)
        Set<GrantedAuthority> authorities = user.getRoles().stream()
            .map(role -> new SimpleGrantedAuthority("ROLE_" + role.getName()))
            .collect(Collectors.toSet());
        // Example:
        // Role "ADMIN" -> Authority "ROLE_ADMIN"
        // Role "USER" -> Authority "ROLE_USER"

        // Return Spring Security UserDetails object
        return org.springframework.security.core.userdetails.User.builder()
            .username(user.getUsername())
            .password(user.getPassword())
            .authorities(authorities)
            .build();
    }
}

```

PART 4: HOW THEY WORK TOGETHER

Example Flow: User Borrows a Book

```
Step 1: User clicks "Borrow Book" button in web interface
↓
Step 2: HTTP Request reaches BorrowController
    BorrowController.borrowBook(bookId, userId)
    ↓
Step 3: Controller calls BorrowService
    BorrowService.borrowBook(user, book)
    ↓
Step 4: Service validates business logic
    - Check if book.isAvailable == true
    - If false, throw error
    ↓
Step 5: Service updates Book entity via BookService
    BorrowService calls BookService.saveBook(book)
    → BookService calls BookRepository.save(book)
    → BookRepository executes SQL: UPDATE books SET is_available = false WHERE id = ?
    → Book updated in database
    ↓
Step 6: Service creates BorrowRecord entity
    BorrowRecord {
        user: user,
        book: book,
        borrowDate: 2024-01-15,
        returnDeadline: 2024-01-29 (14 days later),
        status: "BORROWED"
    }
    ↓
Step 7: Service saves BorrowRecord via BorrowRepository
    BorrowService calls BorrowRepository.save(borrowRecord)
    → BorrowRepository executes SQL:
        INSERT INTO borrow_records (user_id, book_id, borrow_date, return_deadline, stat
        VALUES (?, ?, '2024-01-15', '2024-01-29', 'BORROWED')
    → Record saved in database
    ↓
Step 8: Service returns BorrowRecord to Controller
    ↓
Step 9: Controller returns response to user
    "Book borrowed successfully! Return by 2024-01-29"
```

Example Flow: Search Books

```
User enters "Java" in search box
↓
```

```

BookController.searchBooks("Java")
    ↓
BookService.searchBooks("Java")
    ↓
BookRepository.findByTitleContainingIgnoreCaseOrAuthorContainingIgnoreCase("Java", "Java")
    ↓
Generated SQL:
SELECT * FROM books
WHERE LOWER(title) LIKE '%java%'
OR LOWER(author) LIKE '%java%'
    ↓
Results: [Book 1: Java Programming, Book 2: Learn Java, ...]
    ↓
Return to view, display books

```

PART 5: VALIDATION IN THE PROJECT

1. Database Constraints (Enforced by Database)

```

// User Entity
@Column(name = "username", unique = true, nullable = false)
// No two users can have same username
// Username is required

@Column(name = "email", unique = true, nullable = false)
// No two users can have same email
// Email is required

// Book Entity
@Column(unique = true, nullable = false)
private String isbn;
// ISBN must be unique
// ISBN is required

@Column(nullable = false)
private String title;
// Title is required

// BorrowRecord Entity
@JoinColumn(name = "user_id", nullable = false)
private User user;
// Borrow record must have a user
// Cannot delete user while borrowing

@JoinColumn(name = "book_id", nullable = false)

```

```

private Book book;
// Borrow record must have a book

2. Business Logic Validation (Enforced by Service)

// BorrowService.borrowBook()
if (!book.getIsAvailable()) {
    throw new RuntimeException("Book is not available");
    // Cannot borrow books that are already borrowed
}

// BorrowService.returnBook()
BorrowRecord record = borrowRepository.findById(borrowRecordId)
    .orElseThrow(() -> new RuntimeException("Borrow record not found"));
// Cannot return a non-existent borrow record

// BookService.updateBook()
Book book = bookRepository.findById(id)
    .orElseThrow(() -> new RuntimeException("Book not found"));
// Cannot update non-existent book

```

3. Transactional Integrity (@Transactional)

```

@Transactional
public BorrowRecord borrowBook(User user, Book book) {
    // If error occurs at any point:
    // - Set book.isAvailable = false
    // - Create borrow record
    // If either step fails:
    // - ROLLBACK: Undo both operations
    // - Database remains consistent

    book.setIsAvailable(false);           // Step 1
    bookService.saveBook(book);

    BorrowRecord borrowRecord = new BorrowRecord(...); // Step 2
    return borrowRepository.save(borrowRecord);

    // If step 2 fails, step 1 is rolled back
    // If both succeed, COMMIT changes
}

```

4. Authentication & Authorization (Spring Security)

```

// CustomUserDetailsService loads user with roles
public UserDetails loadUserByUsername(String username) {
    User user = userRepository.findByUsername(username)

```

```

        .orElseThrow(() -> new UsernameNotFoundException(...));

    Set<GrantedAuthority> authorities = user.getRoles().stream()
        .map(role -> new SimpleGrantedAuthority("ROLE_" + role.getName()))
        .collect(Collectors.toSet());
    // User cannot have roles they don't have in database
    // Authentication enforced by Spring Security
}

```

PART 6: VALIDATION SUMMARY TABLE

Validation Type	Location	How It Works	Example
NOT NULL	Database	Column has nullable = false	username cannot be null
UNIQUE	Database	Column has unique = true	Two users can't have same username
PRIMARY KEY	Database	@Id with auto-increment	Each user has unique ID
FOREIGN KEY	Database	@JoinColumn ensures reference exists	Borrow record must reference valid user
Business Logic	Service	Runtime checks in service methods	Cannot borrow unavailable book
Transaction	Service	@Transactional ensures ACID	All-or-nothing operations
Security	Spring Security	UserDetailsService validates auth	User must be logged in
Data Type	Java/Database	Column types enforced	quantity is Integer, not String

PART 7: DEPENDENCY INJECTION EXAMPLE

How Services Get Repositories

```

@Service
@RequiredArgsConstructor
public class BookService {
    // Lombok annotation

```

```

private final BookRepository bookRepository;
// Lombok generates constructor:
// public BookService(BookRepository bookRepository) {
//     this.bookRepository = bookRepository;
// }

// Spring automatically:
// 1. Finds BookRepository interface
// 2. Creates proxy implementation via Spring Data JPA
// 3. Injects it via constructor
}

// At startup, Spring creates:
@Bean
public BookRepository bookRepository() {
    // Automatically generated implementation
    return new BookRepositoryImpl(); // (simplified)
}

// Then injects into BookService:
@Bean
public BookService bookService(BookRepository bookRepository) {
    return new BookService(bookRepository);
}

```

PART 8: COMPLETE OPERATION EXAMPLE

Register New User Flow

1. User fills registration form and submits


```

username: "john_doe"
password: "password123"
email: "john@example.com"

```
2. AuthController.register(User) receives request
3. Check if user exists:


```

UserRepository.existsByUsername("john_doe") → false (OK)
UserRepository.existsByEmail("john@example.com") → false (OK)

```
4. Create User entity:


```

User user = new User("john_doe", "password123", "John Doe", "john@example.com")

```
5. Save to database via UserRepository:


```

UserRepository.save(user)

```

6. Database INSERT:

```
INSERT INTO users (username, password, full_name, email)
VALUES ('john_doe', 'hashed_password_123', 'John Doe', 'john@example.com')
```
7. Assign default ROLE_USER:

```
RoleRepository.findByName("USER") → Role(id=2, name="USER")
User.getRoles().add(role)
UserRepository.save(user) → Updates user_roles junction table
```
8. Success: Redirect to login page

Login Flow

1. User submits login form:

```
username: "john_doe"
password: "password123"
```
 2. Spring Security calls CustomUserDetailsService:

```
loadUserByUsername("john_doe")
```
 3. UserRepository.findByUsername("john_doe") → User object
 4. Get roles: User.getRoles() → [Role(name="USER")]
 5. Convert to authorities: "ROLE_USER"
 6. Create Spring UserDetails:

```
username: "john_doe"
password: "hashed_password_123"
authorities: [ROLE_USER]
```
 7. Spring Security compares passwords:
 - User entered: "password123"
 - Database has: "hashed_password_123"
 - Compare hashes: Match
 8. Create session, redirect to home
-

Summary

Layer	Responsibility	Example
Entity	Data structure & mapping to tables	<code>User.java users table</code>
Repository	CRUD operations & database access	<code>UserRepository.findByUsername(...)</code>
Service	Business logic & validation	<code>BorrowService.borrowBook()</code> checks availability
Together	Controller → Service → Repository → Database	Complete data flow

Each layer has a specific responsibility, and together they form a clean, maintainable architecture.