Advanced Techniques and Tools for Software Development

Library Management System - Project Report

Student: Avan Dave

Matricola: 7096644

Program: Laurea Magistrale in Informatica : Resilient and Secure Cyber Physical Systems (LM-18)

Email: avan.dave@edu.unifi.it

Submitted to: Prof. Lorenzo Bettini

Course: Advanced Techniques and Tools for Software Development

Academic Year: 2024-2025

University: Università degli Studi di Firenze

July 10, 2025

Contents

1 Introduction

The **Library Management System** is a web application designed to manage library operations including book cataloguing, member registration, and borrowing/returning workflows. This project was developed to fulfill the 9-credit requirements for the "Advanced Techniques and Tools for Software Development" course.

1.1 Domain Model

The domain model consists of **two core entities** with a bidirectional relationship:

- **Book entity** and **Member entity** connected through a one-to-many relationship
- A member can borrow multiple books, but each book can only be borrowed by one member at a time
- This model enables library management operations while maintaining referential integrity through JPA annotations and database constraints

1.2 Technology Stack

The application was built using:

- **Java 17 LTS** - Programming language
- **Maven** - Build automation
- **Spring Boot 3.4.5** - Primary framework
- **Dual database support** - H2 for development and MySQL for production
- **Docker** - Containerization
- **GitHub Actions** - Continuous integration

The system provides both **RESTful API endpoints** and a **web interface** implemented with Thymeleaf template engine.

1.3 Quality Standards Achieved

The development process adhered to course methodologies:

- **Test-Driven Development** throughout the codebase
- **100% code coverage** with zero surviving mutants in mutation testing
- **Zero technical debt** through SonarCloud analysis
- **CI/CD pipelines** with automated quality gates

2 Applied Development Techniques and Quality Assurance

2.1 Test-Driven Development (TDD) and Testing Strategy

TDD served as the guiding principle for functionality development throughout the project. The implementation follows a testing strategy adhering to the **testing pyramid methodology**, with proper test distribution across different granularity levels.

2.1.1 Testing Pyramid Implementation:

- **Unit Tests:** Multiple test classes with individual test methods forming the foundation layer
- **Integration Tests:** Several test classes with methods testing component interactions
- **End-to-End Tests:** Test classes validating complete workflows
- **UI Tests:** Test classes ensuring web interface functionality
- **BDD Tests:** Cucumber feature files with scenarios providing behavior specification

Test readability is enhanced through strategic use of:

- **AssertJ** for assertions
- **Mockito** for mocking capabilities
- **TestContainers** for integration testing environments

Each test class follows the **Given-When-Then pattern**, ensuring clear test structure and maintainability. **BDD tests** specifically use **Cucumber feature files** with Gherkin syntax, implementing the Given-When-Then approach through natural language scenarios.

2.2 Mocking Strategy

Mockito is employed in unit tests to achieve component isolation and eliminate external dependencies. Mocking is strategically applied at service layer boundaries, where repository interfaces are mocked to test business logic independently of data persistence concerns.

Example implementation in BookServiceTest.java:

```
@Mock
private BookRepository bookRepository;
@Mock
private MemberRepository memberRepository;
@InjectMocks
private BookService bookService;
```

Listing 1: Mockito Example

This approach enables testing of borrowing logic without requiring actual database operations, ensuring unit tests execute rapidly while maintaining coverage of business rule validation and exception handling scenarios.

2.3 Code Coverage (100%)

The project achieves **100% line and branch coverage** across all business logic components, as verified by JaCoCo and validated through SonarCloud analysis. The coverage report is publicly available via Coveralls integration.

2.3.1 Specific Coverage Achievements:

- **Line Coverage:** 188/188 lines (100%)
- **Instruction Coverage:** 682/682 instructions (100%)
- **Branch Coverage:** 4/4 branches (100%)
- **Method Coverage:** 55/55 methods (100%)

2.3.2 Coverage Exclusions:

- **LibraryManagementSystemApplication.java** - Spring Boot bootstrap class containing only framework initialization
- **Entity classes** (Book.java, Member.java) - JPA data models with generated getters/setters
- **DTO classes** (BookDto.java, MemberDto.java) - Data transfer objects with minimal business logic

These exclusions focus coverage measurement on components containing actual business logic rather than boilerplate code.

2.4 Mutation Testing

PITest is configured to achieve mutation testing with **zero surviving mutants**, ensuring test quality validation beyond simple coverage metrics. The mutation testing strategy focuses on critical business logic components to maintain reasonable build times while maximizing test effectiveness.

2.4.1 Configuration Results:

- **Total Mutations:** 63 mutants generated
- **Killed Mutations:** 63 mutants (100%)
- **Surviving Mutations:** 0 mutants (0%)
- **Mutation Coverage:** 100%

Mutation testing is targeted specifically on **controller and service classes**, excluding entity and DTO classes to focus on business logic validation. The PITest configuration uses default mutators including conditional boundary mutations, arithmetic operator mutations, and return value mutations.

2.5 Build Automation and Modularity

Build automation is implemented using **Maven** with plugin configuration supporting multiple test execution phases. The project follows standard Maven directory structure with clear separation of concerns across architectural layers.

2.5.1 Modular Project Structure:

- **Presentation Layer:** controller package with separate REST and web controllers
- **Business Logic Layer:** service package containing transactional business operations
- **Data Access Layer:** repository package with Spring Data JPA interfaces
- **Domain Layer:** entity package with JPA entity definitions
- **Transfer Layer:** dto package with data transfer objects

Maven configuration includes specialized plugin management for test execution separation, coverage analysis, quality reporting, and mutation testing, ensuring build automation with appropriate phase binding.

2.6 Continuous Integration (CI) with GitHub Actions

The CI pipeline is configured in **GitHub Actions** with the following workflow stages:

1. **Repository Checkout** - Full repository checkout with complete Git history for SonarCloud analysis
2. **Java Environment Setup** - JDK 17 installation with Maven dependency caching
3. **Unit Test Execution** - Surefire plugin execution with JaCoCo coverage reporting
4. **Integration Test Execution** - Failsafe plugin with TestContainers database integration
5. **Quality Analysis** - SonarCloud analysis with quality gate validation
6. **Coverage Reporting** - Coveralls integration with badge updates

The pipeline ensures validation of code quality, test coverage, and build integrity on every push and pull request. **Build failures prevent merging**, maintaining code quality standards throughout the development lifecycle.

Note: The `maven.yml` file was designed deliberately to support the **red-green-refactor** principle. Builds tagged with "red" that have failed are **supposed to fail by design** - representing the initial failing test phase of TDD. The "green" and "refactor" tagged builds are designed to pass, demonstrating the successful implementation and code improvement phases. Some builds appear without tags as they were minor fixes or modifications outside the formal TDD cycle.

2.7 Code Quality with SonarCloud

The project achieves **"clean" status** on SonarCloud with zero bugs, vulnerabilities, code smells, and technical debt. SonarCloud analysis confirms 100% code coverage and validates adherence to industry-standard quality metrics.

2.7.1 Quality Gate Results:

- **Bugs:** 0 (Grade A)
- **Vulnerabilities:** 0 (Grade A)
- **Code Smells:** 0 (Grade A)
- **Technical Debt:** 0 minutes
- **Maintainability Rating:** A
- **Reliability Rating:** A
- **Security Rating:** A

No SonarCloud rules were deactivated or marked as "won't fix," maintaining adherence to quality standards without exceptions.

2.8 Containerization with Docker

Docker integration is implemented primarily through TestContainers for integration test environments, ensuring consistent and isolated test execution. Docker containers automatically start before integration test phases and shut down afterward, both for local Maven builds and CI pipeline execution.

2.8.1 TestContainers Configuration Provides:

- **MySQL Container** - Automated database setup for integration tests with custom initialization
- **Chrome Browser Container** - Selenium WebDriver testing with headless browser automation
- **Network Isolation** - Dedicated container networking preventing test interference
- **Resource Management** - Automatic cleanup preventing resource leaks

3 Design and Implementation Choices

3.1 Presentation Layer Design

The design implements **dual-interface support** through:

- **RESTful API controllers** for programmatic access
- **Web interface controllers** for UI interaction

This enables both machine-to-machine communication and user-friendly web access without code duplication or architectural compromise.

3.2 Data Access Strategy

The implementation utilizes **Spring Data JPA repositories** with:

- Custom query methods for complex operations
- Automatic query generation for standard CRUD operations
- Bidirectional associations with proper cascade configurations
- Lazy loading to prevent N+1 query issues

3.3 Web Interface Architecture

Thymeleaf serves as the template engine for server-side rendering, enabling:

- Dynamic content generation with strong type safety
- Spring Boot integration
- Bootstrap design patterns for responsive, mobile-first user experience
- Form validation and error handling

3.4 Database Configuration Strategy

Dual-database support is implemented with:

- **H2** for rapid development cycles
- **MySQL** for production deployment
- **Spring profiles** for easy database switching without code modifications
- Consistent development-to-production workflows

3.5 Web UI Testing Strategy

The web user interface receives testing at multiple levels:

- **Unit tests** for controller logic using MockMvc
- **Integration tests** for complete request-response cycles
- **End-to-end tests** using Selenium WebDriver for complete user journey validation

This multi-layered testing approach ensures UI reliability across different interaction patterns.

4 Instructions for Project Reproduction

4.1 Prerequisites

- **Git 2.30+**
- **Java 17 LTS**
- **Apache Maven 3.8+**
- **Docker and Docker Compose**

4.2 Setup Steps

1. Clone the repository:

```
git clone https://github.com/AvanAvi/library-management.git
```

2. Navigate to project directory:

```
cd library-management
```

3. Execute complete build and test suite:

```
mvn clean verify
```

This single command executes all unit tests, integration tests, and end-to-end tests while automatically managing required Docker containers through TestContainers integration. The build process generates coverage reports, performs mutation testing, and validates code quality standards.

4.3 Alternative Execution Options

Local Development

Run the application locally using H2 in-memory database:

```
./mvnw spring-boot:run
```

Access at: http://localhost:8081

Docker

Run the application using Docker with MySQL database:

```
./mvnw clean package -DskipTests
# Start MySQL first
docker-compose up -d mysql-db
sleep 30
# Then start the application
docker-compose up library-app
```

Access at: http://localhost:8090

Unit Tests Only

```
mvn clean test
```

I am presenting this part of the report in first person to convey my personal development experience and the lessons learned throughout the project implementation.

5 Challenges Encountered and Solutions

5.1 Challenge 1: JWT Authentication Implementation and Scope Management

I initially planned to include **JWT-based authentication** in the library management system to provide realistic security measures. The implementation involved Spring Security configuration with custom JWT filters, token generation utilities, and role-based access control for API endpoints.

However, during development, I encountered additional complexity that extended beyond the project's core scope as defined by the exam requirements. The JWT implementation required extensive security configuration testing, token lifecycle management, and user session handling that diverted focus from the primary objectives of demonstrating TDD practices and testing strategies.

Solution: My personal development philosophy is to either implement features properly using industry standards or omit them entirely rather than create inadequate solutions for the sake of completion. I decided to **remove the authentication system completely** to maintain project focus on the core technical requirements.

Previous project reference: [https://github.com/AvanAvi/libraryapp1]

5.2 Challenge 2: Mutation Testing Threshold Achievement

Achieving **100% mutation testing coverage** proved to be the most challenging aspect of the project. I consistently reached 90% and above mutation coverage but struggled to eliminate the final surviving mutants without compromising code quality principles.

The challenge was not technical knowledge but rather adapting my development approach to balance mutation testing requirements with clean code practices. I found myself tempted to write additional code specifically to kill surviving mutants, which would violate the **DRY principle**, or implementing overly complex solutions that contradicted the **KISS principle**.

Solution: Finding the middle ground between comprehensive test coverage and maintainable code required careful consideration of each surviving mutant to determine whether additional testing was genuinely valuable or merely satisfying an arbitrary metric. This experience provided valuable learning about balancing testing thoroughness with practical development constraints.

5.3 Challenge 3: Eclipse IDE Adaptation and Workflow Preferences

Adapting to **Eclipse IDE** presented workflow adjustments due to the IDE's established conventions and interface paradigms. While Eclipse provides comprehensive features including integrated PITest execution through run configurations and Maven goal execution through the IDE interface, I found these approaches less intuitive compared to command-line workflows.

Solution: For efficiency and personal preference, I transitioned to using `./mvnw` **commands via terminal** for most build operations including `mvn clean verify`, PITest execution, and test suite runs.

This approach provided more direct control over build processes and clearer output formatting while maintaining Eclipse compatibility for development tasks.

6 Key Insights and Reflections

To avoid overcomplicating the project or imposing additional constraints, I maintained a **mixed programming paradigm approach** - implementing object-oriented patterns where feasible and functional programming concepts where appropriate. This was a personal developmental choice rather than a project metric consideration. Honestly, a **100% object-oriented approach** would have been more beneficial, potentially achieving better DRY compliance and improved modularity, but I was deliberate in not applying any constraints beyond those originally stated in the course requirements.

7 References

The primary source for learning throughout this project remained the course textbook: ***Test-Driven-Development, Build Automation, Continuous Integration***

For specific technologies and frameworks for which I deliberately wanted deeper insight, I referred to their official documentation websites:

- **TestContainers Documentation:** https://www.testcontainers.org/
- **JaCoCo Documentation:** https://www.jacoco.org/jacoco/
- **PITest Documentation:** http://pitest.org/
- **SonarCloud Documentation:** https://sonarcloud.io/