Bookstore Management System

Test-Driven Development Project

Advanced Techniques and Tools for Software Development

9-Credit Exam

Avan Dave

Student ID: 7096644

avan.dave@edu.unifi.it

Submitted to: Prof. Lorenzo Bettini

Contents

1 Introduction

This report documents the design, implementation, and testing of a Bookstore Management System developed following Test-Driven Development (TDD) methodologies. The project was completed for the Advanced Techniques and Tools for Software Development course (9 credits).

1.1 Project Objectives

The objective was to build a functional web application that manages books and categories while strictly adhering to TDD principles throughout the entire development lifecycle. The system provides two interfaces: a REST API for programmatic access and a web-based user interface for interactive management.

Key requirements included:

- test coverage across unit, integration, and end-to-end levels
- 100% mutation testing threshold for service and controller layers
- Continuous integration pipeline with automated quality gates
- Zero technical debt as measured by static analysis tools
- Full reproducibility of builds and test executions

1.2 TDD Methodology Applied

The project follows the Red-Green-Refactor cycle at every level of development. Tests were written before implementation code, ensuring that each feature was validated against explicit behavioral specifications. This approach resulted in a multi-level test suite spanning three distinct layers:

Unit tests verify individual components in isolation using mock dependencies. These tests focus on business logic within services and HTTP request handling within controllers.

Integration tests validate interactions between application layers using real database instances provided by Testcontainers. These tests ensure that JPA mappings, repository queries, and transaction boundaries function correctly.

End-to-end tests exercise the complete application stack, including the MySQL database, Spring Boot runtime, and either the REST API or web UI. Selenium WebDriver tests verify user workflows through the browser, while REST-assured tests validate API contracts.

1.3 Technology Stack

The application uses Spring Boot 3.5.5 with Java 17 as the foundation. Data persistence is handled by Spring Data JPA with Hibernate, backed by MySQL 5.7 in production and containerized MySQL instances during testing. The web interface is built with Thymeleaf templates and custom CSS.

Testing infrastructure includes JUnit 5, Mockito, AssertJ for fluent assertions, HTMLUnit for HTML verification, Selenium WebDriver for browser automation, and REST-assured for API testing. Maven orchestrates the build process with multiple profiles for different testing scenarios.

Quality assurance tools include JaCoCo for code coverage measurement, PITest for mutation testing, SonarCloud for static analysis, and Coveralls for coverage reporting. GitHub Actions provides continuous integration with automated execution of the test suite on every commit.

2 Application Design and Architecture

The system implements a layered architecture with separation between domain models, data access, business logic, and presentation layers.

2.1 Domain Model

The domain consists of two entities: **Book** and **Category**. Books have title, author, ISBN, publication date, and availability status. Categories have a name and organize books into groups.

The relationship is **bidirectional many-to-one**. Each book optionally references one category, while each category maintains a collection of books. Helper methods `addBook()` and `removeBook()` in the Category class maintain consistency on both sides of the relationship.

Category deletion constraint: A category cannot be deleted if it contains books. This is enforced at the service layer through `hasBooks()` and at the controller layer before deletion. The REST API returns HTTP 400 with an error message, the web UI displays a flash error.

Uncategorized books: Books can exist without a category. The repository query `findByCategoryIsNull()` retrieves these books, displayed in the category management interface.

2.2 Layered Architecture

Repository Layer: Spring Data JPA repositories provide CRUD operations. `BookRepository` includes `findByCategoryIsNull()`. Transaction management is delegated to Spring.

Service Layer: `BookServiceImpl` handles book operations and manages category relationships. `CategoryServiceImpl` implements the deletion constraint. The `hasBooks()` method uses `@Transactional(readOnly = true)` to access lazy-loaded collections.

Controller Layer: REST controllers expose JSON endpoints under `/api/*` with HTTP status codes. Web controllers serve Thymeleaf templates with form submissions and redirects.

Presentation Layer: Thymeleaf templates render HTML with custom CSS. Forms use method overriding (`_method=put`) for updates. Flash attributes carry error mes-

sages across redirects.

2.3 Technology Stack

- **Spring Boot 3.5.5** with Java 17
- **Spring Data JPA** with Hibernate
- **MySQL 5.7**
- **Thymeleaf** for HTML templates
- **JUnit 5** with Mockito
- **Testcontainers** for integration tests
- **Selenium WebDriver** for browser tests
- **REST-assured** for API tests
- **HTMLUnit** for HTML verification
- **JaCoCo** for coverage
- **PITest** for mutation testing
- **SonarCloud** for static analysis
- **Maven** for build
- **GitHub Actions** for CI
- **Docker** for deployment

2.4 Design Decisions

Dual interface: REST API under `/api/*` and web UI using the same service layer. Both interfaces access identical business logic.

Constructor injection: All controllers and services receive dependencies through constructors, enabling mock substitution in tests.

Repository abstraction: Services access data through repositories, not direct database queries.

Static factory method: `Book.withTitle(String)` creates book instances with readable syntax in tests.

Layer boundaries: Controllers do not access repositories. Services do not handle HTTP. Each layer tests independently.

3 Testing Strategy and Implementation

The project implements a testing pyramid with three distinct levels, following Test-Driven Development principles throughout. The code was developed using the Red-Green-Refactor cycle: tests written first, watched to fail, implementation added to pass, then refactored for quality.

3.1 Test Organization

Tests are organized into three separate source folders enabling independent execution:

- `src/test/java` - Unit tests (suffix: `*Test.java`)
- `src/it/java` - Integration tests (suffix: `*IT.java`)
- `src/e2e/java` - End-to-end tests (suffix: `*E2ETest.java`)

Maven Surefire executes unit tests during the test phase. Maven Failsafe executes integration tests during the integration-test phase. E2E tests run only when the `e2e-tests` profile is activated. This separation enables fast feedback during development and comprehensive validation before commits.

3.2 Unit Testing

Unit tests verify individual components in isolation without database connections. All external dependencies are replaced with mocks using Mockito.

Model tests validate domain logic including bidirectional relationship management through helper methods like `addBook()` and `removeBook()`, ensuring relationship consistency and constraint validation.

Service tests use `@ExtendWith(MockitoExtension.class)` with `@Mock` repositories and `@InjectMocks` services. Tests verify business rules such as the category deletion constraint (`hasBooks()`) and book update operations including category reassignment.

Controller tests employ two approaches: Plain JUnit tests verify business logic delegation with mocked services. `@WebMvcTest` with MockMvc validates HTTP-level behavior including status codes, redirects, model attributes, and view names without starting the full application server.

HTML verification uses HTMLUnit with WebClient to validate Thymeleaf template rendering, checking element presence, form fields, links, and dynamic content without browser automation overhead.

3.3 Integration Testing

Integration tests verify component interactions using real MySQL databases provided by Testcontainers. The container lifecycle is managed automatically through JUnit Jupiter extensions.

Testcontainers configuration: Integration test classes declare a static `MySQLContainer` field annotated with `@Container`. The `@DynamicPropertySource` method configures Spring datasource properties from the container's JDBC URL after readiness verification, preventing race conditions.

Repository tests use `@DataJpaTest` with `@AutoConfigureTestDatabase(replace = Replace.NONE)` to override in-memory databases. Tests verify JPA mappings, custom queries like `findByCategoryIsNull()`, and cascade behaviors using `TestEntityManager` for explicit flush and clear operations.

Web controller integration tests combine `@SpringBootTest` with `@AutoConfigureMockMvc`. MockMvc performs HTTP requests against controllers backed by real services and repositories, verifying request-response cycles including database persistence and transaction management.

REST controller integration tests use `@SpringBootTest(webEnvironment = RANDOM_PORT)` with REST-assured. The `@LocalServerPort` annotation injects the dynamic port. Tests verify JSON responses, HTTP status codes, and database state after operations.

All integration tests use `@Transactional` where appropriate for test isolation through automatic rollback. For deletion operations, explicit `flush()` and `clear()` calls force persistence and cache invalidation before assertions.

3.4 End-to-End Testing

E2E tests validate user workflows through both web UI and REST API with the full application stack running. The `e2e-tests` Maven profile orchestrates infrastructure startup.

Infrastructure orchestration: The docker-maven-plugin starts a MySQL container in the pre-integration-test phase. The spring-boot-maven-plugin starts the application with datasource properties configured for containerized MySQL. Port conflicts are avoided through the build-helper-maven-plugin, which reserves an available port and passes it to both the application and tests via system properties.

Web UI tests use Selenium WebDriver with headless Chrome. WebDriverManager automatically downloads and configures ChromeDriver binaries. Tests read the server port dynamically and verify complete user workflows: navigation, form filling, submission, and result verification. The `@AfterEach` method ensures proper WebDriver cleanup.

REST API tests use REST-assured configured with dynamic port allocation. Tests perform CRUD operations using the fluent `given().when().then()` pattern, verifying JSON responses, HTTP status codes, and business constraints like the category deletion restriction.

3.5 Test Data Management

Test data creation follows consistent patterns across all levels. The Book entity provides a static factory method `withTitle(String)` for readable test syntax. Integration and E2E tests use fixture methods to create valid test entities, encapsulating common setup logic and reducing duplication.

3.6 Mutation Testing

PITest performs mutation testing on service and controller layers with a 100% mutation threshold. The `mutation-testing` Maven profile activates PITest targeting only `com.attsw.bookstore.service.*` and `com.attsw.bookstore.web.*` classes. Repository classes are excluded from mutation testing. Integration and E2E tests are excluded for performance.

The configuration uses all default mutators while excluding generated methods like `toString()` and `hashCode()`. The 100% threshold ensures every mutation is killed by at least one test, validating test effectiveness beyond simple code coverage.

3.7 Coverage Measurement

JaCoCo measures code coverage for unit tests only, as specified in course requirements. The plugin instruments bytecode during the test phase and generates HTML and XML reports.Coverage excludes the main application class and model entities.

The Maven build enforces coverage thresholds as quality gates. Coverage data uploads to Coveralls during CI builds for historical tracking, while SonarCloud performs static analysis on pull requests with automated quality gates.

4 Challenges and Solutions

This section documents key technical challenges encountered during development and their implemented solutions.

4.1 Testcontainers MySQL Startup Timing

Integration tests occasionally failed with connection refused errors during initial execution. The MySQL container was not fully initialized before Spring Boot attempted database connection, causing intermittent failures in the test suite.

Solution: The Testcontainers MySQL container configuration includes explicit startup verification. Spring Boot's datasource initialization waits for the container's JDBC URL to become available. The `@DynamicPropertySource` method configures properties only after the container reports readiness, ensuring the database is fully accessible before application context initialization and eliminating race conditions.

4.2 Integration Test Transaction Isolation

Early integration tests experienced data contamination where one test's database modifications affected subsequent tests. This caused non-deterministic failures depending on execution order, violating test independence principles.

Solution: Integration tests apply `@Transactional` at the class level, enabling Spring's automatic transaction rollback after each test method. This ensures complete database state isolation between tests. For tests verifying deletion operations, explicit `entityManager.flush()` and `entityManager.clear()` calls force persistence and clear the cache before assertions, ensuring accurate state verification.

4.3 Category Deletion Constraint Enforcement

Implementing the business rule that categories with associated books cannot be deleted required coordination across multiple layers. Initial attempts using database foreign key constraints caused unintended cascading deletion behavior.

Solution: The constraint is enforced programmatically in the service layer. The `CategoryServiceImpl.hasBooks()` method uses `@Transactional(readOnly = true)` to efficiently load the category with its lazy-loaded book collection and check if books exist. Controllers validate this condition before deletion. The REST API returns HTTP 400 with a descriptive JSON error message, while the web controller uses `RedirectAttributes` to display user-friendly flash messages containing the category name and book count.

4.4 JVM Forked Process Timeout

During E2E test execution, the Maven Failsafe plugin occasionally threw timeout errors when terminating the forked JVM process. The default timeout was insufficient for Selenium WebDriver cleanup and Spring Boot application shutdown sequences.

Solution: The Failsafe plugin configuration was extended with a 240-second timeout to accommodate WebDriver browser closure, Testcontainers cleanup, and Spring Boot shutdown hooks. The `exit` shutdown mode ensures forceful termination if graceful shutdown exceeds the timeout, preventing hanging CI builds.

5 Coverage and Quality Metrics

5.1 Code Coverage Results

JaCoCo measures code coverage for unit tests only, as specified in course requirements. The coverage analysis excludes the main application class and model entities.

The project achieves the following coverage metrics:

Class	File	Coverage
BookRestController	BookRestController.java	100%
CategoryRestController	CategoryRestController.java	100%
CategoryWebController	CategoryWebController.java	100%
BookstoreWebController	BookstoreWebController.java	100%
CategoryServiceImpl	CategoryServiceImpl.java	100%
BookServiceImpl	BookServiceImpl.java	100%

Table 1: JaCoCo 100% Coverage Classes (Service & Web Layers)

Coverage reports are generated in HTML and XML formats during the test phase. The XML report is uploaded to **Coveralls** for tracking coverage trends over time. The Coveralls badge in the README displays current coverage percentage.

5.2 Mutation Testing Results

PITest performs mutation testing on service and controller layers with a **100% mutation score threshold**. The analysis generated and killed 55 mutations across 6 mutator types:

Mutator	Generated	Killed
VoidMethodCallMutator	14	14 (100%)
BooleanTrueReturnValsMutator	1	1 (100%)
NullReturnValsMutator	13	13 (100%)
RemoveConditionalMutator	7	7 (100%)
EmptyObjectReturnValsMutator	19	19 (100%)
BooleanFalseReturnValsMutator	1	1 (100%)
Total	**55**	**55 (100%)**

Table 2: PITest Mutation Analysis Results

Key statistics:

- Line coverage for mutated classes: 107/107 (100%)
- Test strength: 100%

- Mutations with no coverage: 0
- Tests executed: 62 (1.13 tests per mutation)
- Total execution time: 1 minute 9 seconds

All 55 mutations were killed, with zero survived mutations.

5.3 Test Execution Metrics

The complete test suite includes:

- **Unit Tests:** Model, service, and controller tests with mocks
- **Integration Tests:** Repository and full-stack web/REST tests with Testcontainers
- **E2E Tests:** Selenium web UI tests and REST-assured API tests

5.4 Continuous Quality Monitoring

Coveralls tracks coverage trends across commits. **SonarCloud** analyzes new code on pull requests and enforces quality gates. **GitHub Actions artifacts** preserve test reports for every build.

6 Reproducibility Guide

This section provides step-by-step instructions to reproduce all testing scenarios and validate the project's quality metrics.

6.1 Prerequisites

Before running any tests, ensure the following tools are installed:

- **Java 17+** - Required for Spring Boot 3.5.5
- **Maven 3.9+** - Build automation and dependency management
- **Docker** - Container runtime for Testcontainers and Docker-based testing
- **Eclipse IDE** - Development environment with Maven integration

6.2 Project Setup

Clone the repository and import into Eclipse:

```
git clone https://github.com/AvanAvi/book-mgmt.git
cd book-mgmt
```

Import in Eclipse: `File → Import → Existing Maven Projects → Select book-mgmt folder`

6.3 Environment Preparation

Before each test run, ensure a clean environment to avoid port conflicts and container issues:

```
# Terminate Java processes
pkill -9 java

# Clean Docker environment
docker stop $(docker ps -aq)
docker rm $(docker ps -aq)

# Verify required ports are available
lsof -i :9090  # Application port
lsof -i :3308  # MySQL port (for Docker profile)
```

Both `lsof` commands should return empty results, confirming ports are available.

6.4 Test Execution Methods

6.4.1 Method 1: Maven Command Line

Execute tests directly via Maven wrapper for automated CI/CD-style verification.

Unit Tests Only:

```
./mvnw clean test
```

Unit + Integration Tests:

```
./mvnw clean verify
```

Testcontainers automatically manages MySQL container lifecycle.

End-to-End Tests:

```
./mvnw clean verify -Pe2e-tests
```

Includes full application startup, browser automation with Selenium, and automatic cleanup.

6.4.2 Method 2: Eclipse IDE Execution

Run tests directly from Eclipse for debugging and development workflows.

Unit Tests:

1. Navigate to `src/test/java`
2. Right-click on `com.attsw.bookstore` package
3. Select `Run As → JUnit Test`

Integration Tests:

1. Navigate to `src/it/java`
2. Right-click on `com.attsw.bookstore` package
3. Select `Run As → JUnit Test`

Docker must be running; Testcontainers handles MySQL automatically.

6.4.3 Method 3: Docker Compose Environment

Full application deployment with E2E testing against production-like environment.

Step 1 - Start Application Stack:

```
docker-compose up --build
```

Wait for log message: `"Started BookstoreManagementTddApplication"`

Services running:

- MySQL 5.7 database (internal networking)
- Spring Boot application: `http://localhost:9090`

Step 2 - Execute E2E Tests in Eclipse:

1. Navigate to `src/e2e/java`
2. Right-click on `BookWebE2ETest.java`
3. Select `Run As → JUnit Test`

Step 3 - Cleanup:

```
# Press Ctrl+C in docker-compose terminal, then:
docker-compose down -v
```

6.4.4 Method 4: Maven Docker Plugin

Manual control of MySQL container with local Spring Boot execution.

Step 1 - Verify Port Availability:

```
lsof -i :9090
lsof -i :3308
```

Step 2 - Start MySQL Container:

```
./mvnw docker:start -Pdocker
```

Step 3 - Start Spring Boot Application WITHOUT DevTools auto-restart

```
./mvnw spring-boot:run -Dspring-boot.run.profiles=local \
  -Dspring-boot.devtools.restart.enabled=false
```

We activate the `local` profile so that the dedicated `application-local.properties` file is picked up; this file contains all settings required for the manually-started MySQL container (port, credentials, DDL mode, etc.). DevTools auto-restart is disabled to prevent application interruption during E2E test execution.

Step 4 - Run E2E Tests: Execute `CategoryWebE2ETest.java` from Eclipse.

Step 5 - Shutdown:

```
# Stop Spring Boot with Ctrl+C, then:
./mvnw docker:stop -Pdocker
```

7 Conclusion

7.1 Technical Decisions and Trade-offs

MySQL 5.7 over MySQL 8.0: Selected for superior containerized environment stability and compatibility, despite newer version availability.

Testcontainers over H2: Used Testcontainers for all integration tests to maintain consistency and avoid dual database setup complexities, ensuring production-like testing conditions.

Unit Testing Approach: Excluded database dependencies from unit tests, using mocks instead to maintain simplicity and focus on business logic isolation.

Test Isolation Strategy: Implemented separate Testcontainers instances per test class via `@DynamicPropertySource` to prevent test pollution, despite increased execution time.

Database Persistence: Used `create-drop` mode in tests to ensure clean state for each test run, prioritizing reliability over speed.

7.2 Addressing Warnings in Test Output

404 Warnings: Intentional from E2E test cases validating error handling; occur exactly twice to demonstrate proper rejection of invalid requests. These warnings are harmless and do not affect test execution.

Timing Considerations: Incorporated strategic waits in test configuration to prevent race conditions and ensure service availability.