SOFTWARE TESTING PROJECT

NIDHEESH KADEM, CJ

04/28/2025

Application Introduction

Online Bookstore Application

- A web-based platform to buy and sell books online.
- Customers can register, browse books, add to cart, and checkout.
- Admins can add, update, or remove books from inventory.
- Order receipts are tracked for users and admins.

Technologies and Features

Technologies Used

- Java Servlets
- JSP (JavaServer Pages)
- JDBC + MySQL (Database)
- HTML, CSS (Front-end)

Key Features

- User Registration & Login
- Book Purchasing
- Cart Management & Checkout
- Admin Inventory Control

Types of Testing Conducted

- Black-box Testing
- White-box Testing
- GUI Testing
- Load Testing
- Mock Testing

Black-Box Testing

Main Features Tested:

- User Registration and Login
- Book Browsing
- Shopping Cart Operations
- Checkout Process
- Admin Functionality (Add/Remove Books)

Key Results

Test Results Summary:

Total Test Cases: 15
Pass: 14
Fail: 1 (Invalid Data in Registration caused server crash)

Defect Highlight

Defect Found:

Registration with Invalid Data:

Server crashes when alphabetic characters are entered in the phone field.
Missing proper input validation.
Needs exception handling for invalid inputs.

White-Box Testing

Main Features Tested:

- Constant Testing (for com.bittercode.constant)
- Model Testing (for com.bittercode.model)
- Service Layer Testing (for com.bittercode.service.impl)
- Util Testing (com.bittercode.util)
- Servlets

Constant Testing

Goal: Ensure all response codes and messages are defined correctly and retrievable.

Tests Conducted:

Getter validation for all codes and messages.
Lookup test for existing status code (e.g., 200 → SUCCESS).
Negative lookup test for non-existing status code (e.g., 999).

Results:

100% code coverage achieved.
All methods and branches tested.
No defects found.

Model Testing

Goal: Test all model classes for correct data handling. Ensure getters, setters, constructors, and special methods work correctly.

Classes Tested:
`Address, Book, Cart, User, UserRole, StoreException`

Model Testing Approach:

Getter/Setter Testing: Verified correct data retrieval and assignment.
Constructor Testing: Checked both default and parameterized constructors.
Special Logic Testing: `StoreException`: Tested different constructor behaviors and setters.
`User`: Tested setting multiple roles.

Tools Used: JUnit 5

Model Testing Results

Test Results:

- **Total Classes Tested:** 6
- **Total Methods Tested:** 58
- **Total Lines Covered:** 109
- **Code Coverage:** 100% (Instructions, Methods, Classes)

Service Layer Testing

Goal:
Test business logic of BookService and UserService.
Validate correct handling of database operations and session management.
Ensure proper exception handling and failure recovery.

Services Tested:
`BookService`
Add, Update, Delete, Fetch Books
`UserService`
Register, Login, Logout, Session Check

Testing Approach: Used **JUnit 5** and **Mockito** for unit testing and mocking.
Simulated both **happy paths** and **failure scenarios** (e.g., database connection failure, invalid inputs).

Service Layer Testing Results

Test Results:

BookService Tests: CRUD operations and edge cases tested.
UserService Tests: User registration, login, duplicate checks, session management tested.
Error Handling: Forced SQLExceptions and negative flows were properly handled without crashing.

Code Coverage Highlights:

High method coverage across service classes.
Database errors, null handling, and session validations tested.
All major business functions passed tests successfully.

Util Testing

DatabaseConfig Testing

Goal: Verify all database connection properties are loaded correctly from `application.properties`.

Challenges:

Full 100% code coverage was not possible without modifying the original file.
The **static block** in `DatabaseConfig` executes **only once** when the class is loaded.
Java does **not allow re-executing static blocks** during testing.

Current Coverage:

All static fields (`DB_HOST`, `DB_PORT`, `DB_NAME`, etc.) were successfully validated.
The **catch block** for `IOException` could not be re-triggered without corrupting configuration files.

Util Testing

DBUtil Testing

Goal: Ensure database connection creation is correct and reliable.

Challenges:

DBUtil also uses a **static block** for initializing the database connection.
To force an error in the static block, the `application.properties` would have to be corrupted.
Corrupting the file **breaks the application memory** and prevents dependent services (`BookService`, `UserService`) from working.

Current Coverage:

Tested `getConnection()` method for valid connection handling.
Exception paths inside the static block could not be tested without breaking the entire app.

Servlet Testing Overview

Purpose of Servlets in This Project

- Servlets were used to **handle user requests** and **control the flow** of the online bookstore application.
- They manage important tasks like **user login**, **adding and updating books**, **managing the shopping cart**, and **processing payments**.
- Servlets allow the application to **respond dynamically** based on user actions and maintain **session data** like shopping cart contents and user authentication.
- They also **handle errors** in a centralized way, making the system more reliable and user-friendly.

Challenge Faced During Testing

- A defensive code check `if (storeException != null)` was **logically unreachable**.
- In Java, casting a non-null object cannot result in a null, making that branch impossible to hit through testing.
- We decided to **leave that branch uncovered**, to maintain clean, realistic code without unnecessary changes.

Testing Results and Key Takeaways

What We Achieved

- **All user roles** (customer, seller, visitor) were fully tested.
- **Session management, form input handling, and business rules** were validated.
- **Exception handling and error messaging** were tested for both system and user-level errors.

Key Testing Strategy

- Designed tests for **success flows, edge cases, and error conditions**.
- Used **Mockito** and **static mocking** to simulate system behavior.
- Focused on **realistic application behavior** instead of blindly covering unreachable code.

Load Testing

Testflows:

1. Admin
    a. Login
    b. Add/Update/Remove Books
2. User
    a. Login/Register/Add book to cart/Purchase Book

Load Testing

Results:

- Admin login flow
- Admin Create Book
  - Low usage good, ramped up slows down significantly

In Progress:

- Customer flows
  - Difficulties caused by captcha, hard to run virtual users
- Admin Update/Remove Book
  - Difficulties with book ID

GUI Testing

Main Features Tested:

- Features Covered
    a. Unlogged in navigation
    b. User Registration
    c. Base user/Admin login
    d. Book Creation
    e. Book Deletion
    f. Book modification
    g. Logged in navigation (Both admin and base user)
    h. Interaction between admin and base users (adding books, modifying books, removing books)
    i. Book ordering

GUI Testing

Faults Discovered:

- Major faults:
  a. Not possible to order a book created by an admin
  b. Can checkout with more than total supply of books
  c. No address and payment verification
     i. Only address and card number need to be input (sort of)
- Minor Faults:
  a. Sign-in navigation has issues when navigating from home page
  b. Can create account without accepting terms and conditions
  c. Entering phone number with '-' causes internal server error during registration
  d. Books are not cleared from "available books" when out of stock

GUI Testing

Successes:

- All navigation paths tested, most verified
- Most possible actions by customer tested
- Most actions by admin tested
- Many major faults identified

Challenges:

- Various alerts popping up from google regarding password leaks (weak passwords used for testing) caused tests to struggle at unpredictable times

Thank You!