Library Management System

Streamline the process of managing books, users and borrowing events

Presented by

Ridwan Galib, Ridaa Bhatti, Inaaya Rana, Ryan Edward, Oluwadamilare Sunmola, Zenidy Le, Sameer Vashisth

Objective

- Design and document a web-based Library Management System that supports core library operations borrowing, returning, and inventory updates.
- Ensure this system is **reliable**, **secure**, and **intuitive**, with performance targets such as fast transactions and high uptime while maintaining data privacy

- Use the Incremental Process Model so new features can be delivered and tested in manageable stages.
- Use a client server architecture to separate library logic and data management on centralized servers while the web client will focus on displaying information and sending user requests.

Cost Estimation

Hardware Costs

Cloud Server: $360
Backup Storage: $90
Developer/Testing Machines: $1400

Total: $1850

Software Costs

Payment Gateway: $120
Notification/Email API: $60
Reporting/Analytics: $100
Database, Framework, IDE: Free

Total: $280

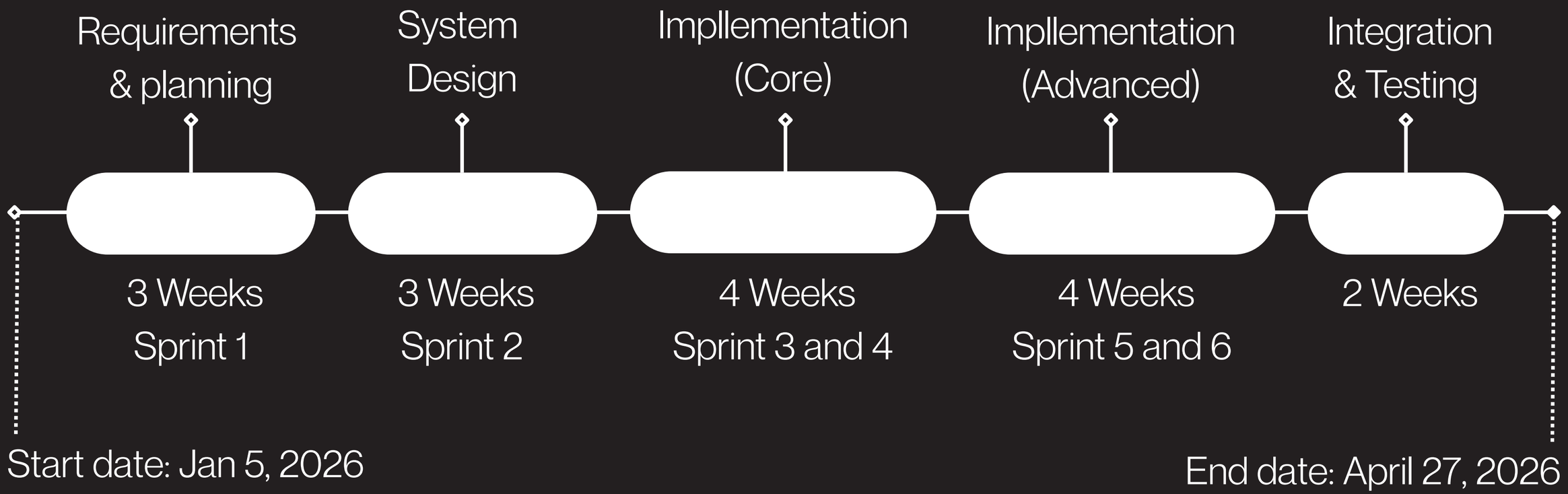
Labor Costs

Developer Labor: $81,270
Staff Training: $450
Admin Training: $150
Maintenence Tech Onboarding: $300

Total: $82,170

Total Estimated Cost: ~$84,000

Project Timeline



Functional Requirements

Specify what the system must do

- Secure user authentication
- Role-based permissions
- Book search (title, author, ISBN, or category)
- Borrow and return books
- Automatic book availability updates
- Management of book records
- Inventory update log

Non-Functional Requirements

Specify how the system must operate

Product

The system shall maintain 99% uptime during operational hours.

Organizational

The system shall be developed using a Client-Server architecture pattern.

External

The system shall prevent unauthorized data modification.

Challenges Faced

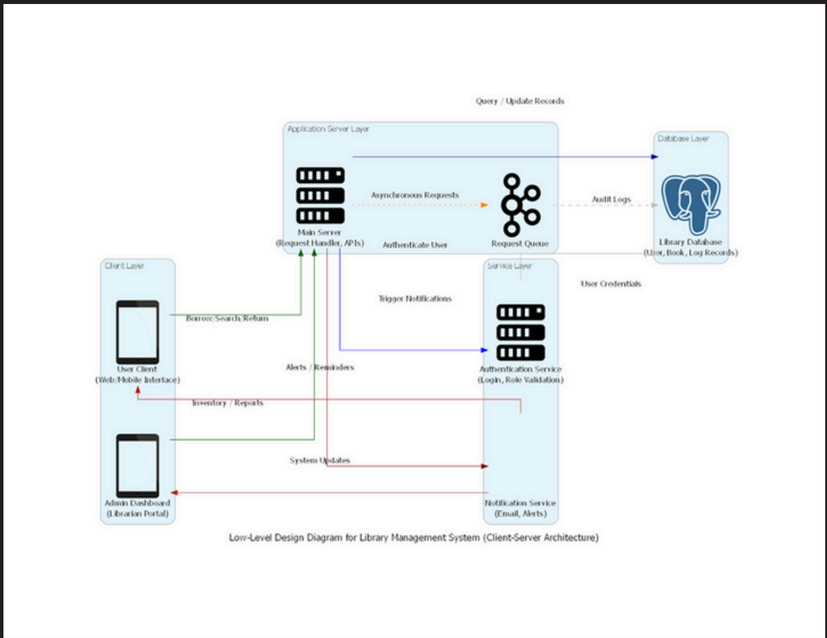
API Integration

API Integration took a lot of time to figure out. Once we connected the API to our project, we were able to advance this project further

Connecting Frontend to Backend

A lot of time was spent debugging in order to get both the frontend and backend connected.

Architectural Design



Low-Level Design Diagram for Library Management System (Client-Server Architecture)

Description

The Architectural Design diagram illustrates the high-level structure of the system, outlining its main components and their interactions. For this project, the Client-Server model is employed

Client

Web Browser/Application: The user interface through which members and librarians interact with the system. It sends requests to the server and displays responses.

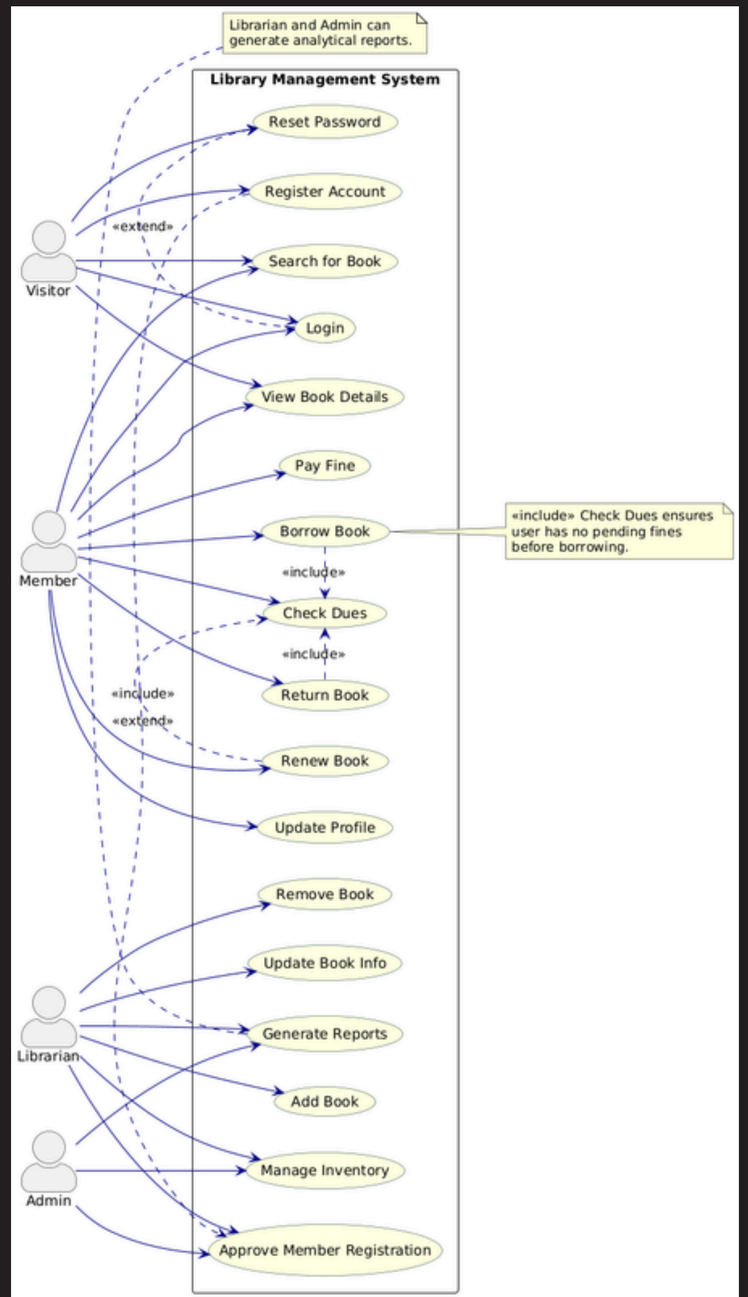
Server

Web Server: Handles HTTP requests from clients and directs them to the appropriate application components.
Application Server: Contains the business logic of the system, processing requests related to user management, book inventory, borrowing, and returns.
Database Server: Stores all persistent data, including user accounts, book records, and borrowing information.

Interactions

Clients send requests to the Web Server.
The Web Server forwards requests to the Application Server.
The Application Server interacts with the Database Server to retrieve or store data.
Responses are sent back through the Application Server and Web Server to the Client.

Use Case Diagram



Visitor/Member

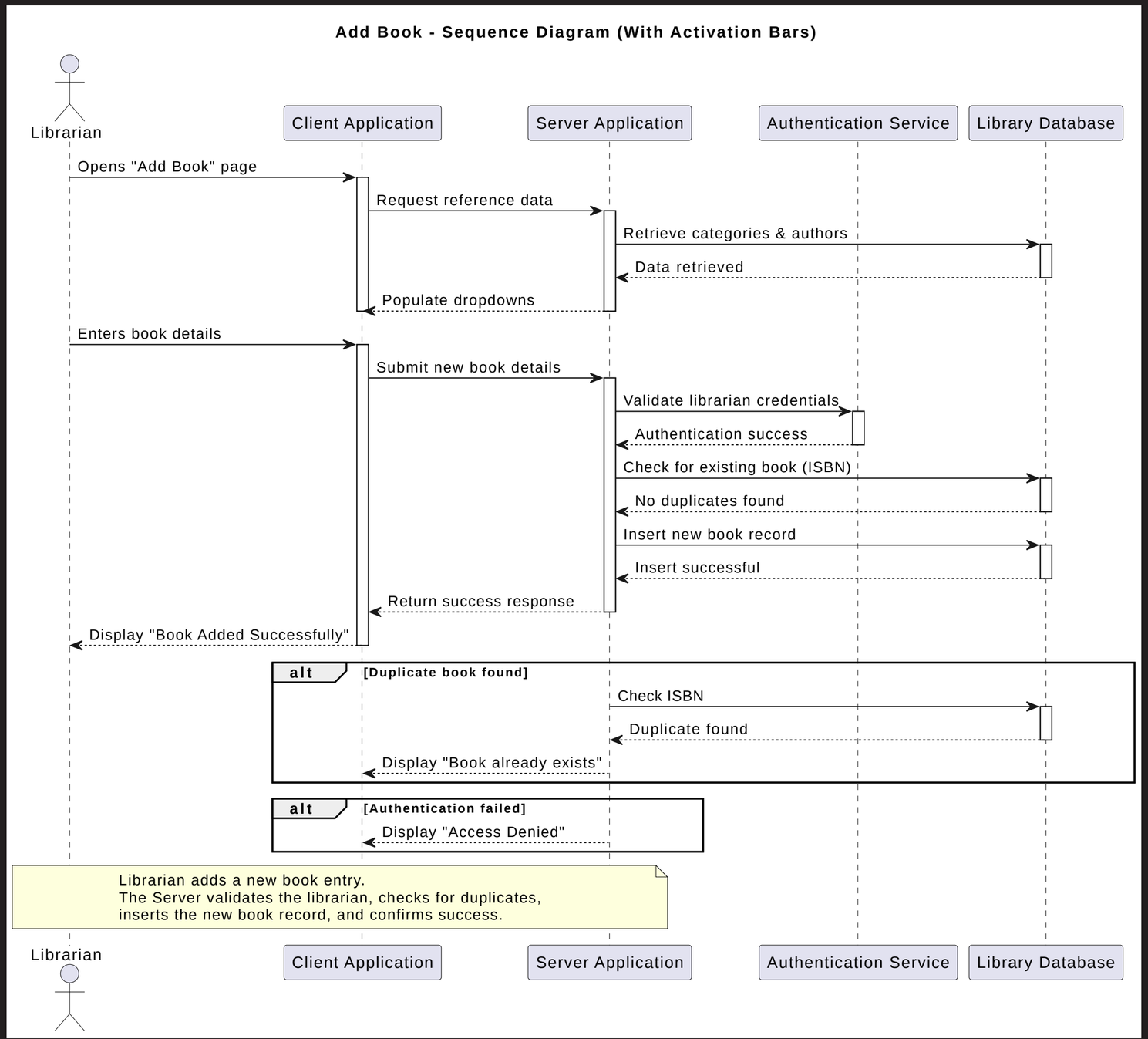
- Search for a book
- Visit book details
- Register an account
- Login
- Reset Password

Member

- Pay fines
- Borrow book
- Check Dues
- Return Book
- Renew Book

Admin/Librarian

- Add/Remove a book
- Update book inventory
- Generate reports
- Approve member registration



Sequence Diagram - Add Book

Operation allows the librarian to upload a new book to the library, the system authenticates the user, checks International Standard Book Number for duplicates, then inserts into the database

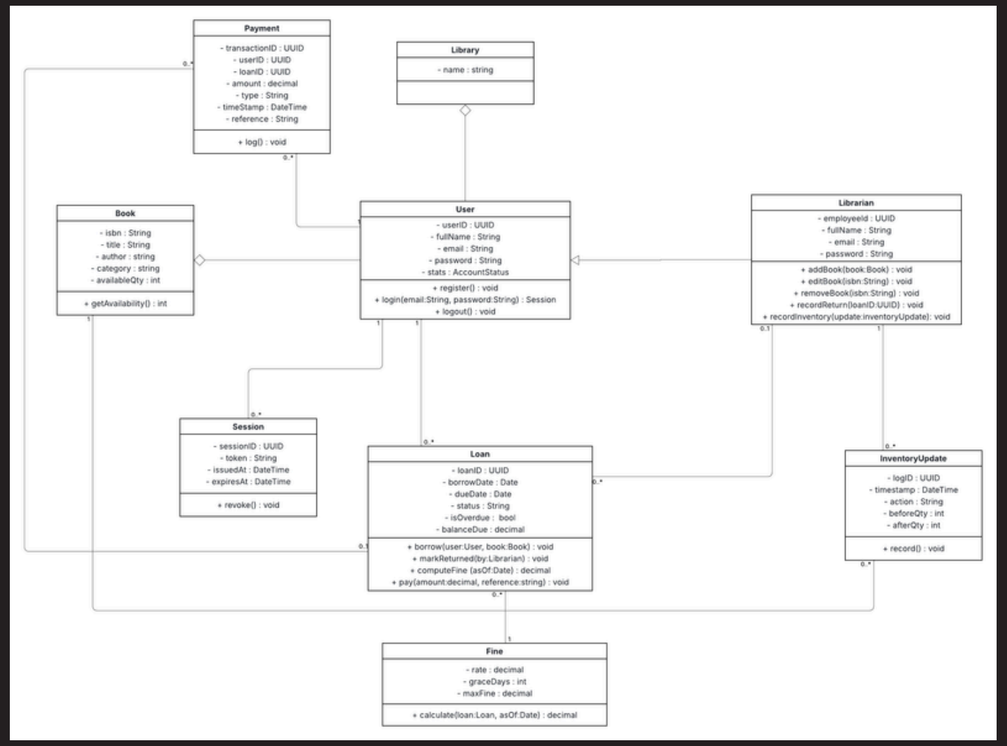
Actors

- Librarian
- Client App
- Server
- Authentication service
- Library Database

Features

- Authentication
- Duplication prevention
- Database write
- feedback

Class Diagram



Description

The Class Diagram models the core domain of the Library Management System. It displays all of the classes that the system needs, and will also show how the will interact with each other to show the core workflow.

- User
- Library
- Payment
- Book
- Librarian
- InventoryUpdate
- Fine
- Session

Each class has methods to interact with each other. For example user has login(email, password) and that creates a session with a unique sessionID.

Each class has attributes, for example, user has userID, fullName, email, password, stats. These attributes can be modified using methods that each class has.

Implementation Code: Loan Class

```python
from datetime import datetime, timedelta

class Loan:
    DAILY_FINE = 0.50
    GRACE_DAYS = 2
    MAX_FINE = 20.00

    def __init__(self, loan_id, user_id, book, borrow_date=None):
        self.loan_id = loan_id
        self.user_id = user_id
        self.book = book
        self.borrow_date = borrow_date or datetime.now()
        self.return_date = None

    def mark_returned(self):
        self.return_date = datetime.now()
        self.book.mark_as_returned()

    def compute_fine(self):
        due_date = self.borrow_date + timedelta(days=14 + self.GRACE_DAYS)
        if not self.return_date or self.return_date <= due_date:
            return 0.0
        overdue_days = (self.return_date - due_date).days
        return min(overdue_days * self.DAILY_FINE, self.MAX_FINE)

```

What this code implements

- Implements the loan class from our class diagram
- Handles borrowing, returning, and fine calculation
- Computes overdue fines using project rules
- Direclty interacts with the book class

Why It Is Important

- Controls to the libarary's borrowing workflow
- Connecsts users to books to payments/fines
- Ensures policy enforcement (due dates, penalties)

Unit Test: Loan Fine Calculation

```python
import unittest
from datetime import timedelta

class TestLoan(unittest.TestCase):
    def test_fine_no_overdue(self):
        loan = Loan(1, 10, Book(1, "1984", "Orwell", "123", "Fiction"))
        loan.return_date = loan.borrow_date + timedelta(days=10)
        self.assertEqual(loan.compute_fine(), 0.0)

    def test_fine_overdue(self):
        loan = Loan(1, 10, Book(1, "1984", "Orwell", "123", "Fiction"))
        loan.return_date = loan.borrow_date + timedelta(days=30)
        self.assertGreater(loan.compute_fine(), 0.0)

    def test_fine_max_cap(self):
        loan = Loan(1, 10, Book(1, "1984", "Orwell", "123", "Fiction"))
        loan.return_date = loan.borrow_date + timedelta(days=300)
        self.assertEqual(loan.compute_fine(), Loan.MAX_FINE)

```

What these Tests Validate

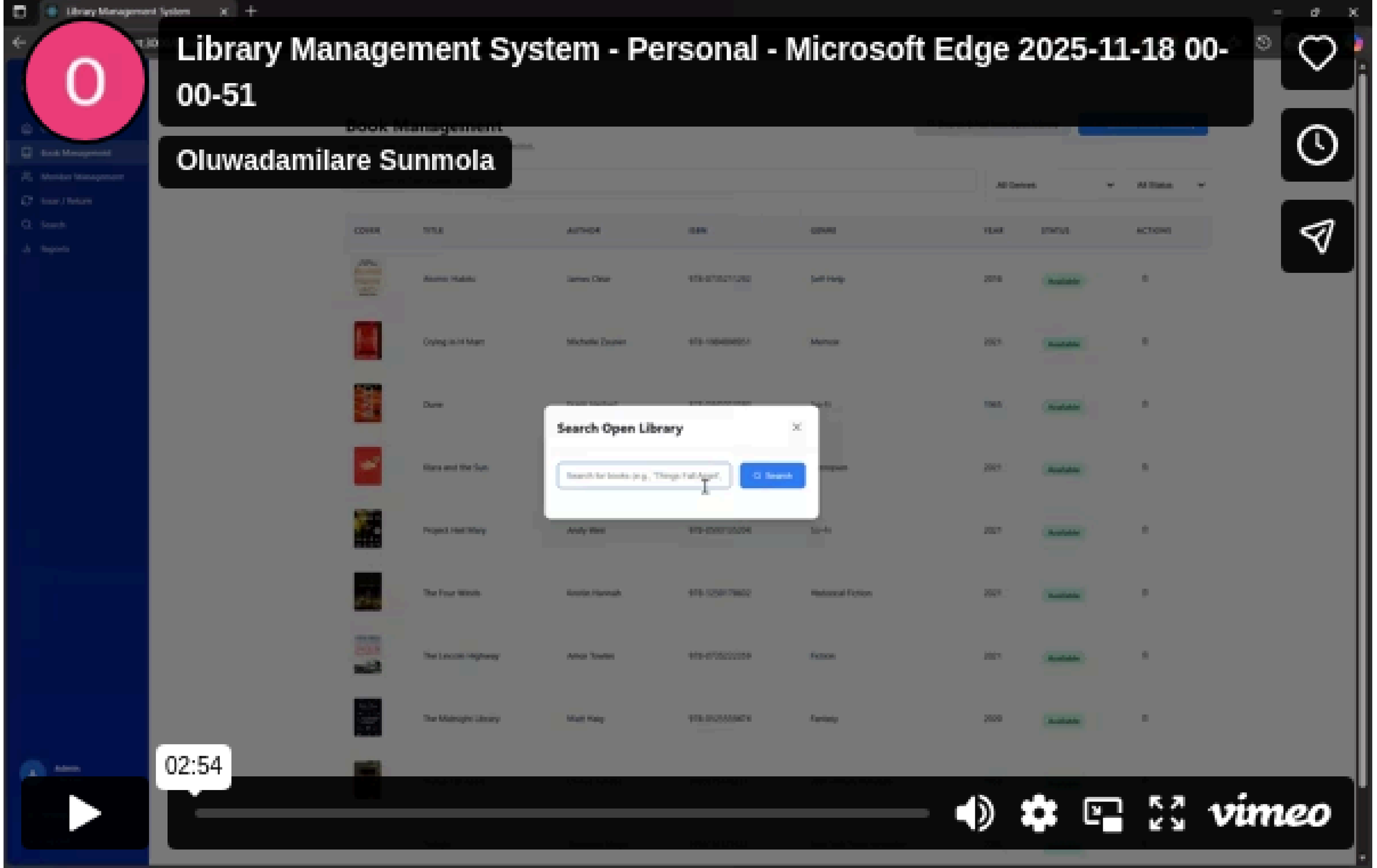
- Returning early or on time. (Fine is 0)
- Overdue returns (Fine is positive)
- Extremely late returns (fine is capped)
- Ensures the implementation handles invalid/edge cases

Why Unit Tests Matter

- Prevents incorrect fine calculation
- Ensures system reliability for borrowing and returning
- Automatically detects regressions during development
- Validates time based logic

Examples

- Negative fine amounts
- Incorrect overdue calculations
- If the book return date is earlier than the borrow date
- If fine exceeds the max allowed amount
- Loan without a book object
- Book marked as returned twice


Library Management System - Personal - Microsoft Edge 2025-11-18 00-00-51
Oluwadamilare Sunmola
Search Open Library
02:54
vimeo