An Academic Report on

Verification and Validation of a Design-by-Contract Layered Library Management System

Submitted To:

Dr. Raja Manzar Abbas

Submitted By:

Anil Manyam (Student Id:202483873)

Sadia Alam (Student Id: 202480704)

TABLE OF CONTENTS:

1. ABSTRACT

This project focuses on combining formal verification and practical validation techniques in building a layered Library Management System (LMS) that follows the principles of Design by Contract (DbC). The main aim is to ensure the system is correct, consistent, and robust by making sure each part of the system meets its defined contractual responsibilities. The LMS is structured in layers, which helps separate responsibilities clearly and enforce contracts at each level. Key actions like borrowing and returning books, managing borrowing limits, and keeping the inventory accurate are formally modeled using the Alloy specification language and checked for logical correctness through model checking. During runtime, the system uses PyContracts, a Python library that enforces contracts at the function level. Additionally, unit tests are created using pytest, where the expected results are based on the system specifications and used as test oracles. By combining formal methods with hands-on testing, this approach shows how software can be made more reliable. The project includes a working LMS prototype, verified Alloy models, Python code with contracts, and test outputs demonstrating that integrating verification and validation into real-world development leads to measurable improvements.

Keywords: *Library Management System (LMS), Design by Contract, Unit Testing, System Testing, Exploratory Testing, PyContracts, pytest, Python Mocking, End-to-End Testing, User Experience Testing, Mocking & Fixtures, Test Case Design.*

2. INTRODUCTION

Modern software systems are growing in complexity, making it increasingly important to ensure their correctness, reliability, and robustness especially in applications that handle real-world data and user interactions, such as Library Management Systems (LMS). While traditional software testing is useful for identifying bugs, it often falls short when it comes to proving that a system fully complies with its formal specifications. This challenge has brought greater attention to software verification and validation (V&V) techniques, particularly those grounded in formal methods and the principles of Design by Contract (DbC). This project focuses on designing, verifying, and validating an LMS that uses Design by Contract as a core methodology for ensuring software correctness. The system is built using a layered architecture, which encourages modular design and a clear separation of responsibilities. Contracts defined through preconditions, postconditions, and invariants are applied at various points in the system to make component responsibilities explicit and enforceable.

To evaluate the system's correctness and reliability, we adopt a hybrid approach that combines formal verification with practical validation techniques. Critical system behaviors, such as borrowing and returning books, are formally modeled using the Alloy specification language and verified using the Alloy Analyzer to ensure logical consistency and adherence to defined constraints. On the implementation side, PyContracts is used within the Python codebase to enforce these contracts during runtime. Additionally, we use pytest to develop unit tests based on expected behaviors drawn from the system specifications, serving as test oracles to validate the implementation under different scenarios. By combining formal methods with runtime validation, this approach aims to enhance system reliability while demonstrating how these techniques can be integrated into real-world development workflows. Through this project, we explore the practical benefits of using Design by Contract and formal modeling to build software systems that are not only functional but also safer and more dependable.

3. LITERATURE REVIEW

3.1 State of the Art in Software Verification and Witness Validation (Beyer, 2024)

The 2024 edition of the Software Verification Competition (SV-COMP), as described by Beyer, marked the 13th and largest event to date. It evaluated 76 tools 59 verifiers and 17 validators submitted by 34 teams. These tools were tested against a comprehensive benchmark suite consisting of 30,300 C programs and 587 Java programs, covering key properties like reachability, memory safety, integer overflows, and termination [1]. A significant addition this year was the introduction of a second track focused on witness validation. This included a new YAML-based witness format (version 2.0) and an updated scoring system for validators, aiming to better connect verification tools with independent result validation [1].

Beyer emphasizes SV-COMP's broader objectives: to benchmark the current capabilities of verification tools, to maintain a public and reproducible repository of benchmark tasks and tool outputs, and to set standards for result formats and evaluations. All competition data is archived publicly (e.g., via Zenodo) to promote transparency and reproducibility. Although the top-performing tools produce very few incorrect results, the report underscores that even a small number of errors makes thorough result validation essential [1]. The report argues that expanding the use of verification tools and improving validation techniques will directly enhance software reliability. SV-COMP stands alongside other formal-method competitions like VerifyThis and TermCOMP, providing an annual snapshot of the field's progress. Related initiatives such as Test-Comp (for automated test-case generation) and RERS (for reactive systems) demonstrate the complementary nature of formal verification and practical software testing. Collectively, these efforts showcase both the achievements and the challenges in bridging formal guarantees with real-world software development [1].

3.2 Fault Localization on Verification Witnesses (Beyer et al., 2024)

In a related study, Beyer et al. focus on improving the usefulness of violation witness's counterexample traces produced when a verifier detects a potential error. These witnesses are meant to be as concise and relevant as possible, enabling validators to quickly confirm the issue. However, in practice, witnesses often include unnecessary details, such as extensive loop traces, which complicate validation and waste resources [2]. To tackle this, the authors propose a technique that applies fault-localization methods to reduce witnesses to only the statements that are crucial for triggering the error. Using SMT-based tools like MaxSAT or MinUNSAT, the method identifies a minimal set of relevant program statements. This reduced witness is then easier to understand and validate. The approach was implemented in the CPAchecker framework and a new tool called Flow [2].

Their evaluation, based on 21,356 violation witnesses generated by 14 verifiers on 3,225 benchmark tasks from SV-COMP 2023, shows promising results. The reduction technique decreased witness size by about 25% on average, and more importantly, increased the number of successful validations by up to 34% [2]. Many alarms that could not previously be confirmed in time were now verifiable. This work is the first formal method to apply fault localization to verification witnesses and demonstrates how concepts from debugging can meaningfully enhance formal verification outputs [2].

3.3 Runtime Verification: 24th International Conference (Ábrahám & Abbas, 2025)

The 24th International Conference on Runtime Verification (RV 2024), edited by Ábrahám and Abbas, presents the latest advances in monitoring software behavior against formal specifications.

Held in Istanbul from October 15 to 17, 2024, the conference proceedings include 18 peer-reviewed papers spanning topics like cyber-physical systems, temporal logics, specification visualization, neural networks, and distributed systems [3]. This variety reflects a growing interest in applying runtime verification techniques to complex, real-world scenarios such as autonomous systems and machine-learning components.

The conference featured a keynote by Bonakdarpour on "Distributed Runtime Verification with Imperfect Monitors," highlighting current challenges in dealing with partial and uncertain observations during system execution [3]. The proceedings also include a tool showcase, demonstrating the field's emphasis on practical applications. Common themes among the papers include efficient stream-based monitoring, hybrid methods, and new approaches for ensuring the safety of neural-network-based controllers. Overall, RV 2024 showcases how runtime verification complements traditional static analysis by enabling real-time correctness checking in deployed systems. The papers demonstrate both theoretical contributions and applied innovations, reinforcing the field's role in improving the reliability of increasingly dynamic and data-driven software systems [3].

3.4 Native Design by Contract in Python via Class Invariants (Andrea, 2025)

In a 2025 online discussion, Andrea proposed a native approach to incorporating design-by-contract (DbC) into Python by introducing a special method called __invariant__(). The idea is that Python would automatically call this method before and after each public method invocation, thereby enforcing class invariants rules that ensure the internal consistency of an object without relying on decorators or meta classes. Andrea points out that other languages like Eiffel, D, and Ada already support DbC at the language level, and suggests that Python could benefit from a similar opt-in, debug-mode mechanism. This would be especially helpful in stateful systems such as simulations or financial applications, where ensuring the consistency of object state is critical. The discussion critiques current Python solutions, such as decorators and meta classes, for being manual, fragile, and inconsistent across projects. Unlike decorators, which must be applied explicitly to each method or class and can be error-prone, a built-in invariant mechanism would standardize and simplify enforcement, potentially improving both code robustness and tooling support. Andrea backs the idea with a preliminary proof-of-concept using a C-extension (though it's not fully functional) and refers to a more detailed discussion on Python-Dev for community feedback. Responses from the community were mixed. Some developers raised concerns about performance overhead if the interpreter checks for __invariant__() on every method call, it could slow down applications that don't use contracts. Others questioned whether Python needs a new language feature for something decorators already handle, even if imperfectly. Andrea clarifies that the aim isn't to make writing invariants easier, but to make their enforcement more reliable and explicit. This sparked a broader conversation about whether such a change fits with Python's design philosophy, and whether the benefits justify the complexity.

In summary, the proposal represents a meaningful exploration of contract-based programming in Python. Though informal and not peer-reviewed, the discussion captures both interest and skepticism within the Python community. It sheds light on the broader desire for built-in verification tools that offer runtime assurances similar to unit or system tests but integrated more deeply into the language itself.

4. SYSTEM ARCHITECTURE

This Library Management System (LMS) is designed using a three-tier layered architecture, separating the system into the presentation, service, and data layers. This separation improves clarity, maintainability, and flexibility for future updates.

4.1 Layered Architectural Model:

Presentation Layer (React.js): Provides a responsive, component-based user interface. It handles user interactions like searching for books or initiating loans.

Service Layer (Flask): Acts as the middleware, exposing a REST API. It applies business logic, validates inputs using Design by Contract (icontract), and handles communication between the frontend and database.

Data Layer (MongoDB Atlas): A cloud-based NoSQL database that stores structured collections for users, books, loans, loan history, and feedback. Its schema-less design allows easy data model evolution.

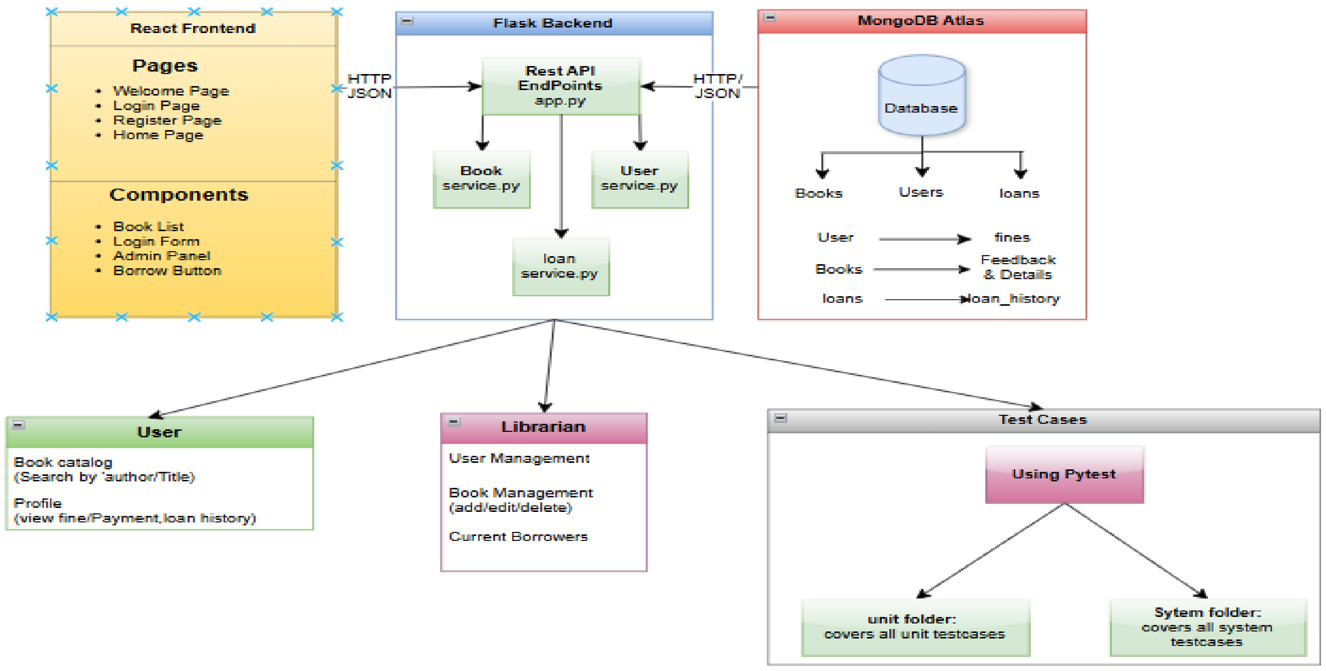
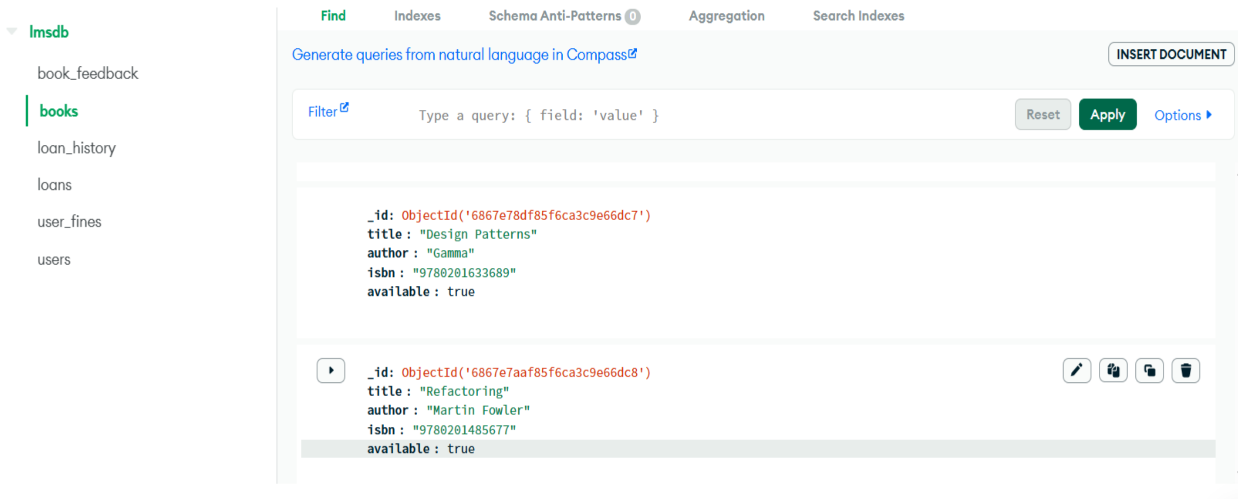


Figure 1: Layered Architecture Diagram of LMS

4.2 Data Persistence Strategy & Communication Flows: MongoDB Atlas offers flexibility through its document-oriented storage and global accessibility. Key collections include users, books, loans, loan_history, and book_feedback. Indexes on fields like username and book_id ensure fast data retrieval, while replica sets enhance fault tolerance and availability.



```
_id: ObjectId('6867e78df85f6ca3c9e66dc7')
title : "Design Patterns"
author : "Gamma"
isbn : "9780201633689"
available : true
```

```
_id: ObjectId('6867e7aaf85f6ca3c9e66dc8')
title : "Refactoring"
author : "Martin Fowler"
isbn : "9780201485677"
available : true
```

Figure 2: The above figure showcases all the collections in our database.

Finally, The React frontend interacts with Flask through RESTful APIs and retrieve the necessary information from the database based on user requests and functions.

4.3 Testing Strategy: For testing purposes as shown in figure1, MongoDB collections are mocked using Python's MagicMock, allowing for isolated, controlled testing environments that avoid altering real data. This setup ensures reliable test results and enables the development of robust test cases without database dependencies. While the project focuses on backend verification and validation, it is designed to work seamlessly with a frontend client potentially built using frameworks like React that communicates through the defined REST APIs. This layered and decoupled architecture not only supports end-to-end testing but also aligns with widely accepted web development best practices, making the LMS easier to extend and maintain over time.

5. METHODOLOGY

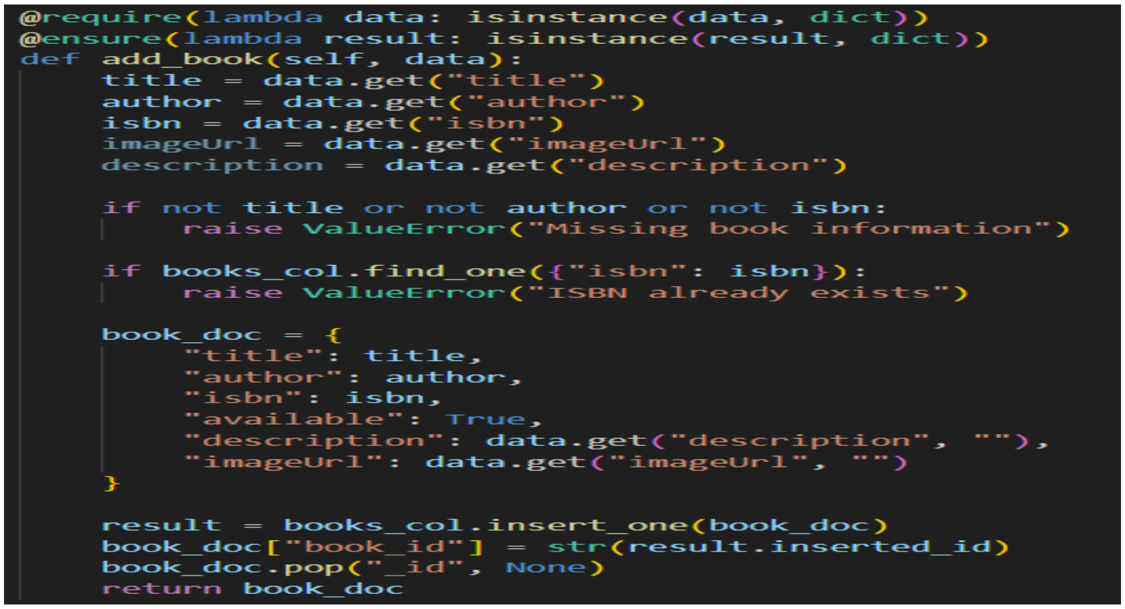
The verification and validation (V&V) strategy for the Library Management System is designed to ensure that the application works correctly both internally and from the user's perspective.

5.1 Unit Testing:

As a first step in verification and validation, Confirming that the core components like book management, user operations, and loan processing function as they should. To do this, a detailed set of unit tests is maintained under **"Backend/tests/unit/"** in codebase, where each piece of business logic is tested separately. The testing framework used is pytest, with MagicMock employed to simulate interactions with MongoDB. These tests cover essential features like user sign-up, borrowing limits, checking book availability, calculating due dates, and applying fines. For example, the below two-unit testcases one from regular user side and another from librarian/Admin side showcases the step-by-step explanation.

Test Case 1: test_add_book_success (Librarian)

Step 1 - Identify the Function to Test: The function under test is **BookService.add_book(data: dict)** showcased in figure 3, which plays a key role in allowing librarians to add new books to the library's catalog. This method first checks whether a book with the same ISBN or title already exists in the system to avoid duplication. If no existing match is found, it proceeds to insert the new book into the MongoDB collection. Since this function directly affects how the library maintains a unique and organized collection of books, any malfunction such as allowing duplicate entries or failing to insert a book could disrupt the core functionality of the system.



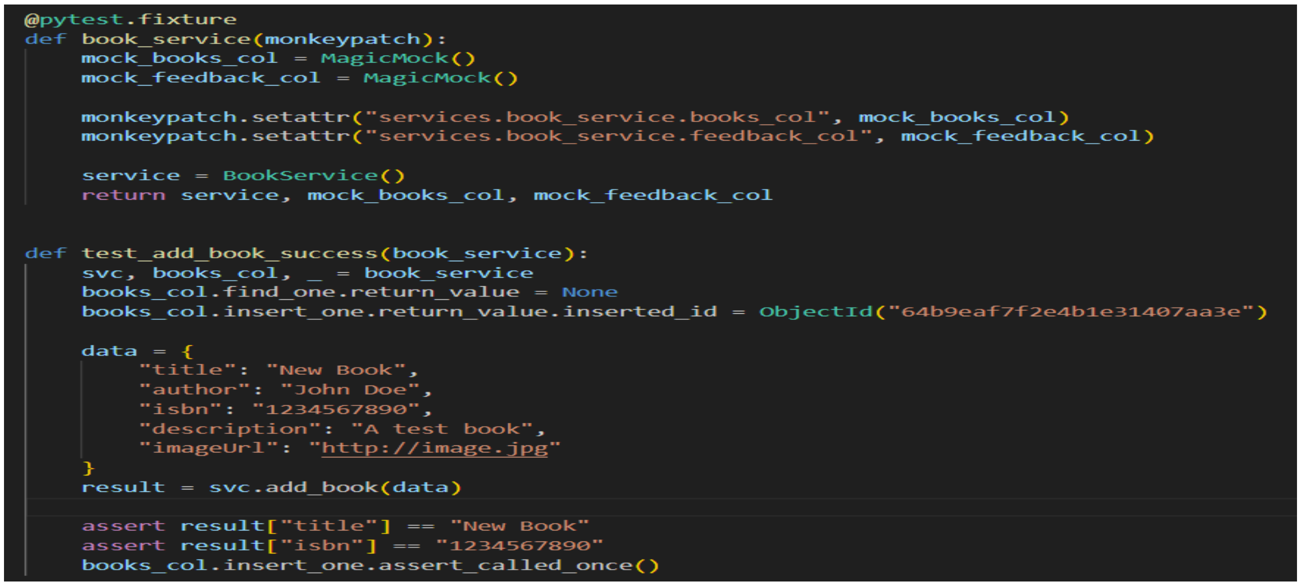
```python
@require(lambda data: isinstance(data, dict))
@ensure(lambda result: isinstance(result, dict))
def add_book(self, data):
    title = data.get("title")
    author = data.get("author")
    isbn = data.get("isbn")
    imageUrl = data.get("imageUrl")
    description = data.get("description")

    if not title or not author or not isbn:
        raise ValueError("Missing book information")

    if books_col.find_one({"isbn": isbn}):
        raise ValueError("ISBN already exists")

    book_doc = {
        "title": title,
        "author": author,
        "isbn": isbn,
        "available": True,
        "description": data.get("description", ""),
        "imageUrl": data.get("imageUrl", "")
    }

    result = books_col.insert_one(book_doc)
    book_doc["book_id"] = str(result.inserted_id)
    book_doc.pop("_id", None)
    return book_doc
```

Figure 3: Python code of add a book function by Administrator

Step 2 - Analyze & List Dependencies: This function relies primarily on the books MongoDB collection. It uses the 'find_one' operation to search for existing entries with the same ISBN or title and the 'insert_one' method to add a new book if no duplicates are found. While users collection may be part of the broader service, it's not directly involved in this particular test. To ensure the test remains predictable and doesn't interact with actual database contents, all these dependencies must be mocked and fully isolated.

Step 3 –Isolate the Function: To test the function in isolation, Python's unittest.mock.MagicMock is used to simulate the MongoDB collection. The pytest.monkeypatch utility is then used to override services.book_service.books_col and users_col with these mock objects. The mock setup is configured so that find_one returns None, simulating the absence of a duplicate book, while insert_one returns a dummy ObjectId to mimic a successful insertion. This approach keeps the test self-contained and ensures it doesn't require or affect any live MongoDB instance.

Step 4 - Write & Run the Unit Test: The test begins by defining the behavior of the mocked database operations as shown in figure 4. find_one is mocked to return None, indicating that there's no existing book with the given ISBN or title. insert_one is then mocked to simulate a successful insertion of a new book. The test proceeds by calling the add_book method with a sample valid data payload. The response is then checked to verify that the returned output contains the correct title and ISBN values. Additionally, the test asserts that insert_one was called exactly once, confirming that the book was "inserted" into the mock database as expected.



```python
@pytest.fixture
def book_service(monkeypatch):
    mock_books_col = MagicMock()
    mock_feedback_col = MagicMock()

    monkeypatch.setattr("services.book_service.books_col", mock_books_col)
    monkeypatch.setattr("services.book_service.feedback_col", mock_feedback_col)

    service = BookService()
    return service, mock_books_col, mock_feedback_col


def test_add_book_success(book_service):
    svc, books_col, _ = book_service
    books_col.find_one.return_value = None
    books_col.insert_one.return_value.inserted_id = ObjectId("64b9eaf7f2e4b1e31407aa3e")

    data = {
        "title": "New Book",
        "author": "John Doe",
        "isbn": "1234567890",
        "description": "A test book",
        "imageUrl": "http://image.jpg"
    }
    result = svc.add_book(data)

    assert result["title"] == "New Book"
    assert result["isbn"] == "1234567890"
    books_col.insert_one.assert_called_once()
```

Figure 4: The unit testcase code for adding a book by Admin using pytest

Step 5 - Automate & Repeat: This unit test is located in the file '**tests/test_book_service.py**' and can be executed individually using the command '**pytest -k test_add_book_success -v**'. It is also part of the automated test suite that runs during continuous integration, helping to ensure that future code changes in the service layer don't unintentionally break the book addition feature. By isolating this critical admin function with reliable, repeatable testing, the system maintains both stability and safety without any risk to real data.

Test Case 2: test_borrow_book_success (Regular User)

Step 1-Identifying the Function: This test is focused on **LoanService.borrow_book(user_id: str, book_id: str)** function shown in figure 5, which handles the process of a user borrowing a book from the library system. The function first validates the user's identity and eligibility, checks whether the selected book is currently available, ensures the user hasn't reached their borrowing limit, created a new loan record, and then updates the book's status to unavailable. Because this function enforces key borrowing rules and manages book availability, any failure could lead to issues like users borrowing more than allowed or checking out books that are already loaned, ultimately causing inconsistencies in the system.

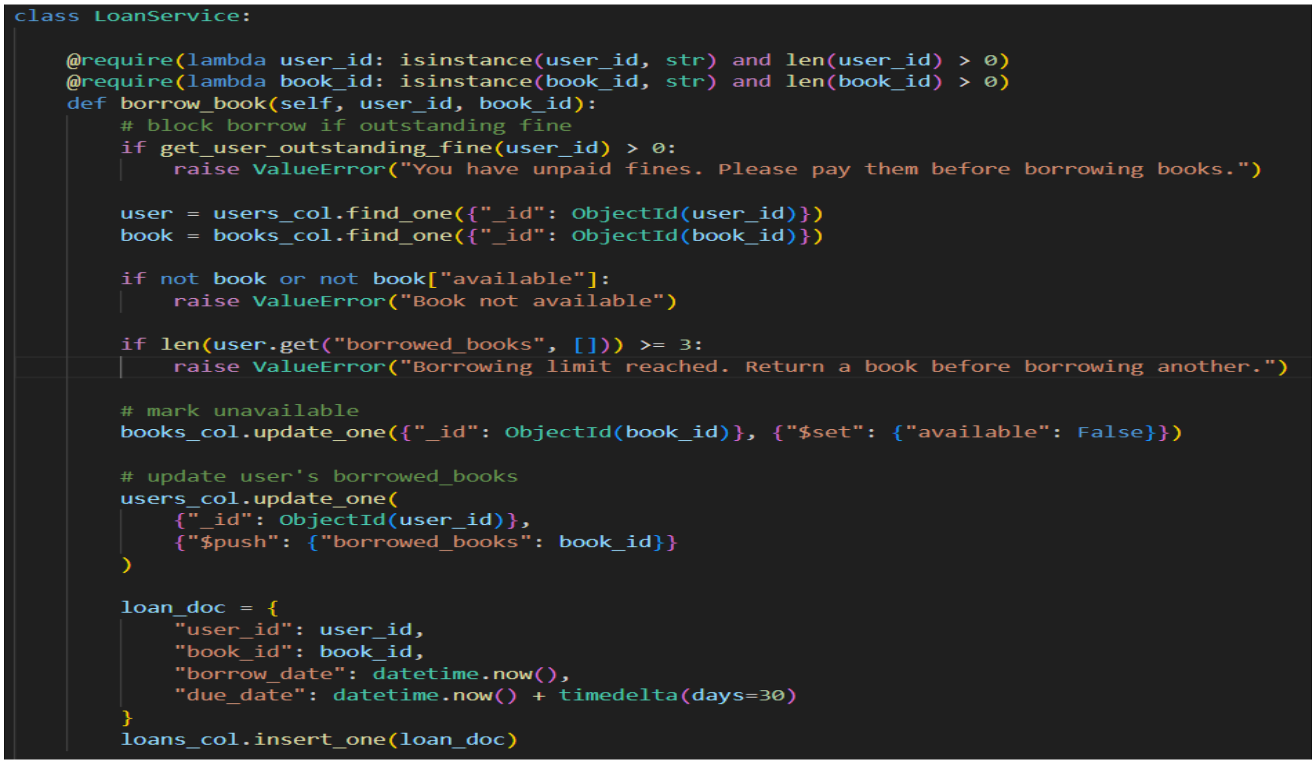
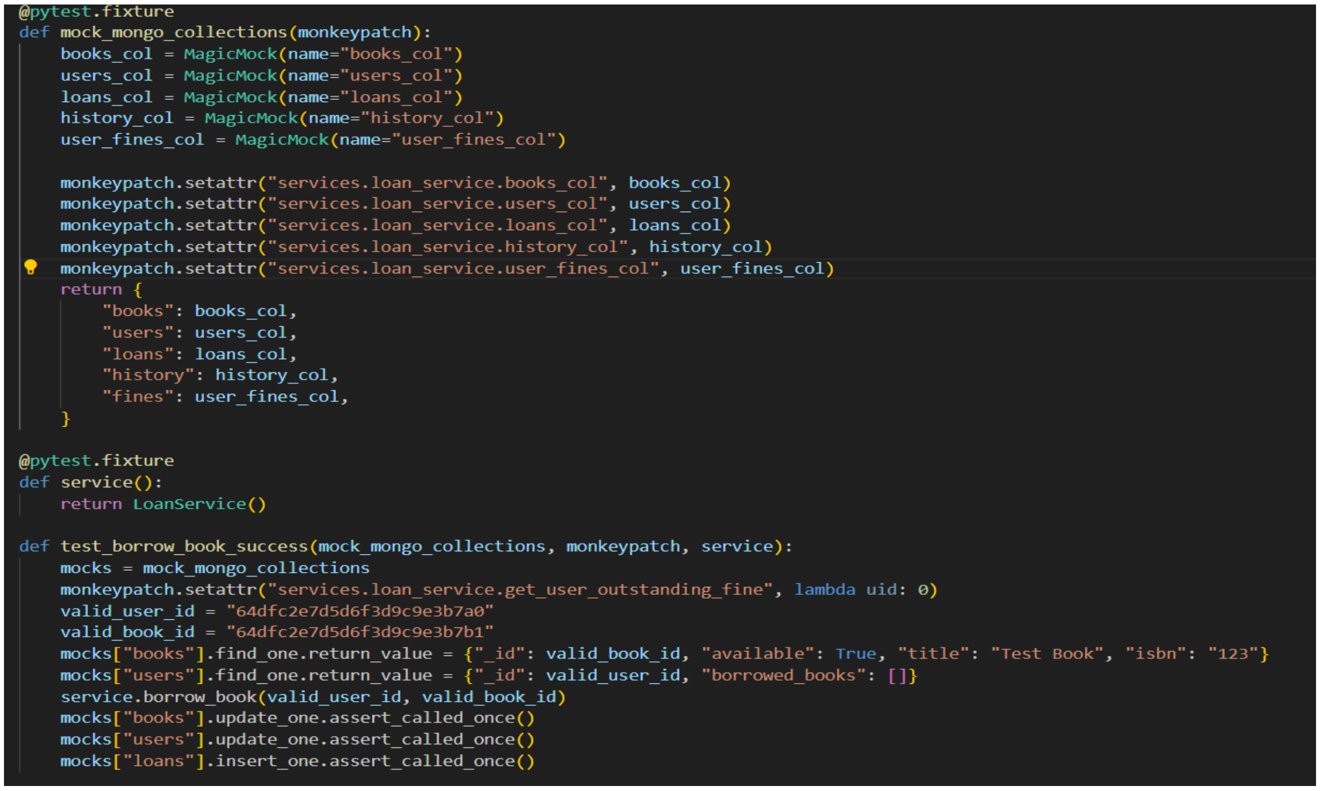

```python
class LoanService:

    @require(lambda user_id: isinstance(user_id, str) and len(user_id) > 0)
    @require(lambda book_id: isinstance(book_id, str) and len(book_id) > 0)
    def borrow_book(self, user_id, book_id):
        # block borrow if outstanding fine
        if get_user_outstanding_fine(user_id) > 0:
            raise ValueError("You have unpaid fines. Please pay them before borrowing books.")

        user = users_col.find_one({"_id": ObjectId(user_id)})
        book = books_col.find_one({"_id": ObjectId(book_id)})

        if not book or not book["available"]:
            raise ValueError("Book not available")

        if len(user.get("borrowed_books", [])) >= 3:
            raise ValueError("Borrowing limit reached. Return a book before borrowing another.")

        # mark unavailable
        books_col.update_one({"_id": ObjectId(book_id)}, {"$set": {"available": False}})

        # update user's borrowed_books
        users_col.update_one(
            {"_id": ObjectId(user_id)},
            {"$push": {"borrowed_books": book_id}}
        )

        loan_doc = {
            "user_id": user_id,
            "book_id": book_id,
            "borrow_date": datetime.now(),
            "due_date": datetime.now() + timedelta(days=30)
        }
        loans_col.insert_one(loan_doc)
```

Figure 5: The Borrow book function form Loanservice.py file

Step 2 - Analyze & List Dependencies: This function depends on three MongoDB collections: users_col, books_col, and loans_col. It uses ‘users_col.find_one’ to fetch the user and determine their type (e.g., regular or admin). It checks book availability using ‘books_col.find_one’, and determines whether the user has reached their borrowing limit with ‘loans_col.count_documents. To register the loan’, it calls ‘loans_col.insert_one’, and then updates the book’s availability through ‘books_col.update_one’. Since all of these are database interactions, they need to be mocked in the unit test to isolate and test only the business logic.

Step 3 - Isolate the Function: To isolate this function properly, MagicMock is used to simulate all three MongoDB collections users_col, books_col, and loans_col. These mocks are applied using pytest.monkeypatch, ensuring that none of the actual database operations are executed during the test. Each mock is set up to return specific values to mimic real behavior: for instance, returning user data from find_one, a book marked as available, a loan count of zero from count_documents, and dummy results for inserting a loan and updating the book’s status. This setup allows for accurate testing of the function’s logic without relying on real data.

Step 4 - Write & Run the Unit Test: The test sets up the necessary inputs and mock behaviors as shown in figure 6 and a valid regular user is returned from the user lookup, the selected book is reported as available, and the loan count is zero, indicating that the user hasn't borrowed any books yet. The insert_one method of loans_col is mocked to return a fake loan ID to simulate a successful loan creation. After calling borrow_book with the mock user and book IDs, the test checks that the result includes the correct loan ID. Additionally, the test confirms that the insert_one and update_one method were each called exactly once, ensuring that both the loan was recorded and the book's status was updated correctly.



```python
@pytest.fixture
def mock_mongo_collections(monkeypatch):
    books_col = MagicMock(name="books_col")
    users_col = MagicMock(name="users_col")
    loans_col = MagicMock(name="loans_col")
    history_col = MagicMock(name="history_col")
    user_fines_col = MagicMock(name="user_fines_col")

    monkeypatch.setattr("services.loan_service.books_col", books_col)
    monkeypatch.setattr("services.loan_service.users_col", users_col)
    monkeypatch.setattr("services.loan_service.loans_col", loans_col)
    monkeypatch.setattr("services.loan_service.history_col", history_col)
    monkeypatch.setattr("services.loan_service.user_fines_col", user_fines_col)
    return {
        "books": books_col,
        "users": users_col,
        "loans": loans_col,
        "history": history_col,
        "fines": user_fines_col,
    }

@pytest.fixture
def service():
    return LoanService()

def test_borrow_book_success(mock_mongo_collections, monkeypatch, service):
    mocks = mock_mongo_collections
    monkeypatch.setattr("services.loan_service.get_user_outstanding_fine", lambda uid: 0)
    valid_user_id = "64dfc2e7d5d6f3d9c9e3b7a0"
    valid_book_id = "64dfc2e7d5d6f3d9c9e3b7b1"
    mocks["books"].find_one.return_value = {"_id": valid_book_id, "available": True, "title": "Test Book", "isbn": "123"}
    mocks["users"].find_one.return_value = {"_id": valid_user_id, "borrowed_books": []}
    service.borrow_book(valid_user_id, valid_book_id)
    mocks["books"].update_one.assert_called_once()
    mocks["users"].update_one.assert_called_once()
    mocks["loans"].insert_one.assert_called_once()
```

Figure 6: unit testcase code for borrowing a book using pytest and Mock

Step 5 - Automate & Repeat: This unit test is stored in **tests/test_loan_service.py** and can be run using the command **pytest -k test_borrow_book_success -v**. It's also integrated into the project's automated testing workflow, playing a crucial role in verifying the correctness of the book borrowing logic. By fully mocking external dependencies, the test ensures that this core functionality continues to behave as expected even if future updates affect other parts of the system. Similarly, we have written unit testcases (around 35) for every functionality in our code base and stored in **'Backend/test/unit'** folder.

5.2 System Testing:

As a second step validation takes a broader view by checking that the system behaves properly when used through its public API. System tests, stored in **tests/system/**, are used to mimic realistic user activity by sending HTTP requests to key endpoints like /api/books, /api/users/login, /api/loans/borrow, /api/loans/overdue, and /api/admin/users. These tests are run using Flask's test_client, along with monkeypatch and complete mocking of MongoDB collections. This setup allows the system's external behavior to be tested under controlled conditions, while simulating how users would interact with it in practice. The below is the explanation one such system testcase.

Test Case: One of the system-level scenarios selected for testing is the CreateUser() functionality, where a librarian or administrator creates a new user through the '/api/admin/users' API endpoint.

Step 1: Identify Key Decision Points for CreateUser()

- User Authorization - Check whether the request is being made by a properly logged-in administrator (librarian). Only authorized users should be able to create new accounts.
- Input Data Validity - Ensure that all required fields such as username, email, and password are present in the request and meet validation criteria (e.g., proper email format).
- Username/Email Uniqueness - Verify that the username or email provided isn't already in use. The system should prevent duplicate user accounts.
- Database Insert Outcome - Confirm that the new user can be successfully inserted into the database without errors.
- Header Format - Check that the user ID is correctly included in the request header (X-User-Id) to identify the admin performing the operation.

Step 2: Identify Representative Values (Partitions)

To effectively test the CreateUser() functionality, we focused on three main decision areas and define representative value partitions for each:

User Authorization:

i. Authorized Admin - A librarian making the request with a valid X-User-Id header.
ii. Unauthorized User - A regular (non-admin) user or a request missing the required header.
iii. Invalid User ID: A request that includes a fake or non-existent user ID in the header.

Input Data Validity:

i. All Fields Valid - The request includes all required fields (username, email, password), and each one meets validation standards.
ii. Missing Fields: One or more required fields are absent for example, no password.
iii. Invalid Format: Fields are present but incorrectly formatted, such as an invalid email.

Username/Email Uniqueness:

i. Unique: The provided username and email don't exist in the system yet.
ii. Duplicate: The username or email is already associated with existing user in the database.

Step 3: Generate Test Case Specifications

Specification 1 - Successful User Creation

i. Choices & Values:
   - User Authorization: Authorized Admin
   - Input Data Validity: All Fields Valid
   - Username/Email: Unique

ii. Expected Outcome: User is created
   - API returns 201 Created
   - Response: "User created successfully"
   - User is inserted into the database
   - Admin sees confirmation in frontend

Specification 2 - Duplicate Email or Username

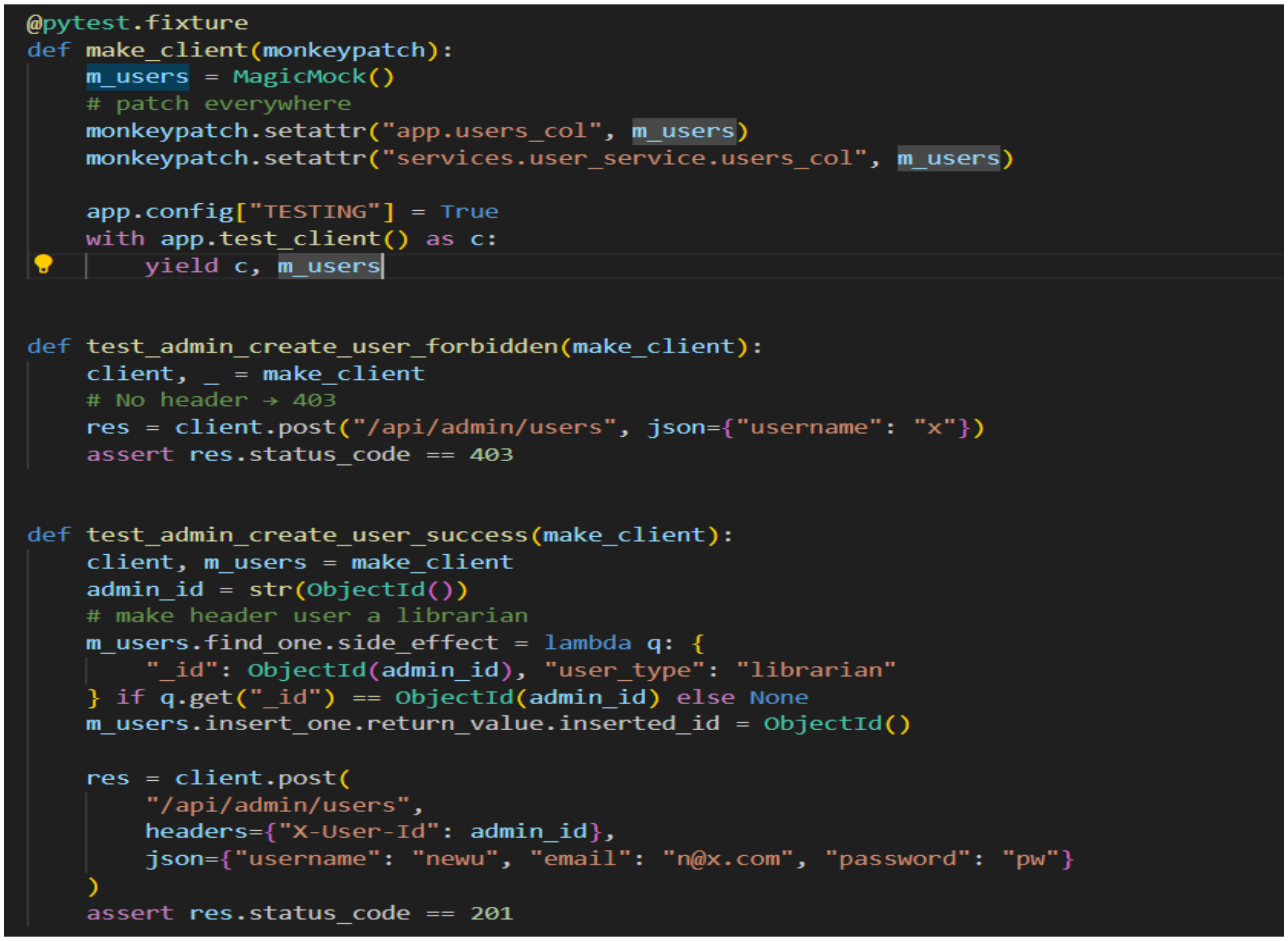
i. Choices & Values:
   - User Authorization: Authorized Admin
   - Input Data Validity: All Fields Valid
   - Username/Email: Duplicate

ii. Expected Outcome: User rejected
   - API returns 409 Conflict or 400 Bad Request
   - Error message: "Username or email already exists"
   - No duplicate user is inserted into the database

Step 4: Generate Concrete Test Cases

Concrete Test Case 1 - Successful Creation, A valid librarian with proper authorization creates a user with unique credentials. The system responds with a success message and stores the new user in the database.

Concrete Test Case 2 - Failure Due to Duplicate Email, an authorized admin attempts to create a user with an email that already exists in the database. The system blocks the operation and returns a 409 Conflict error.



```python
@pytest.fixture
def make_client(monkeypatch):
    m_users = MagicMock()
    # patch everywhere
    monkeypatch.setattr("app.users_col", m_users)
    monkeypatch.setattr("services.user_service.users_col", m_users)

    app.config["TESTING"] = True
    with app.test_client() as c:
        yield c, m_users


def test_admin_create_user_forbidden(make_client):
    client, _ = make_client
    # No header → 403
    res = client.post("/api/admin/users", json={"username": "x"})
    assert res.status_code == 403


def test_admin_create_user_success(make_client):
    client, m_users = make_client
    admin_id = str(ObjectId())
    # make header user a librarian
    m_users.find_one.side_effect = lambda q: {
        "_id": ObjectId(admin_id), "user_type": "librarian"
    } if q.get("_id") == ObjectId(admin_id) else None
    m_users.insert_one.return_value.inserted_id = ObjectId()

    res = client.post(
        "/api/admin/users",
        headers={"X-User-Id": admin_id},
        json={"username": "newu", "email": "n@x.com", "password": "pw"}
    )
    assert res.status_code == 201
```

Figure 7: System Test Case Implementation for CreateUser() API Endpoint

Similarly, we have written all the possible system testcases (around 20) in our code base and stored in **‘Backend/test/system’** folder.

5.3 Exploratory testing:

Beyond automated testing, the V&V strategy also includes plans for exploratory testing, which is performed to validate the end-to-end functionality and user experience of the Library Management System through direct interaction with the UI. Unlike scripted test cases, this testing approach simulates real user behavior logging in, borrowing and returning books, viewing details, and leaving feedback. The aim is to uncover usability issues, integration errors, and confirm that the entire system behaves as expected under realistic scenarios. Below are the documented steps with visual evidence from the actual system. For an Instance we have explained User Journey through the Library Management System from welcome page to giving feedback for a book after returning it by using some figures of our LMS.

Step 1: Visiting the Landing Page - When the application is first launched, the user is welcomed by a clean, professional landing page that introduces the Library Management System as shown in figure 8. Two prominently displayed call-to-action buttons **Login** and **Register** make it easy for both returning users and new visitors to navigate the platform. The background imagery adds an academic, trustworthy feel, reinforcing the system's purpose and tone.

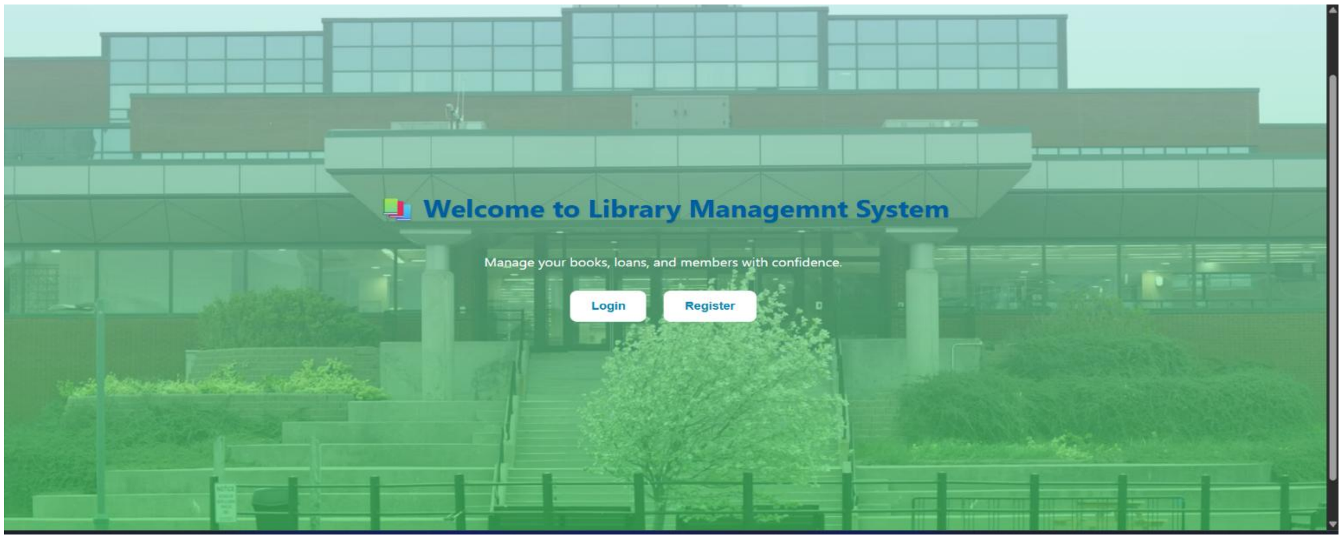
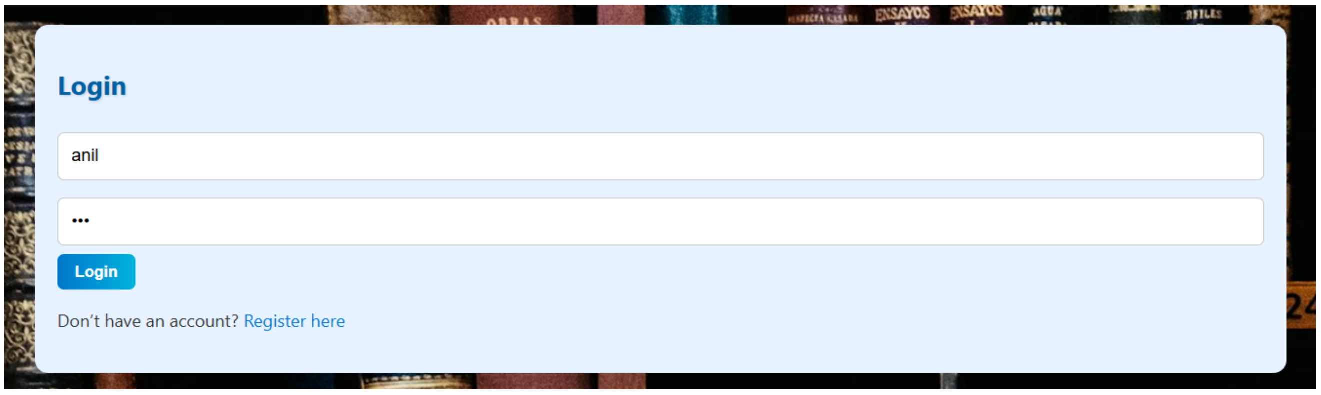

Welcome to Library Managemnt System
Manage your books, loans, and members with confidence.
Login
Register

Figure 8: Landing or Welcome Page of the Library Management System

Step 2: Logging in as a User


Login

anil

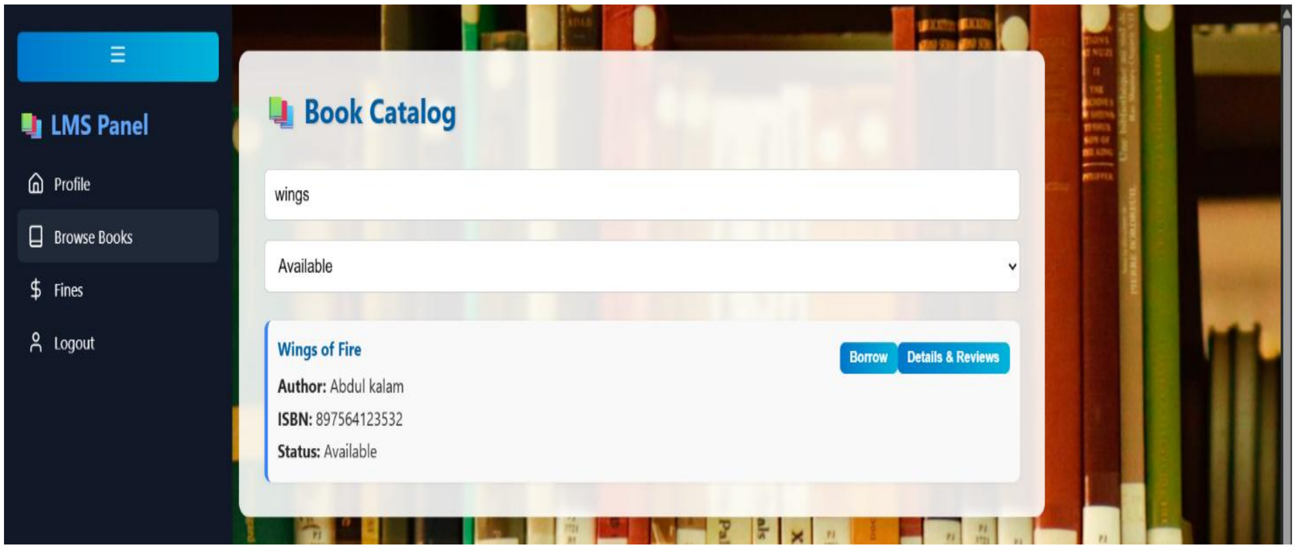
•••

Don't have an account? Register here

Figure 9: User Login Interface

By clicking the **Login** button brings the user to a straightforward and functional login interface (figure 9). Here, the user enters valid credentials (e.g., username: *anil* and the corresponding password). The design remains minimalistic, focusing on clarity and usability. Upon submitting the login form, the backend handles authentication, and if successful, the user is redirected to their personal dashboard marking a smooth transition from login to access.

Step 3: Searching for a Book - After logging in, the user is taken to the profile Catalog. A sidebar provides access to features like Profile, Browse Books, Fines, and Logout. To search for a book, the user clicks on Browse books option then user types "wings" into the search bar. The catalog updates in real time, filtering results and displaying based on author or book tittle in this case it displayed Wings of Fire among the matches as shown in figure 10. Each book listing offers options such as Borrow and Details & Reviews, allowing users to interact with content in a good way.



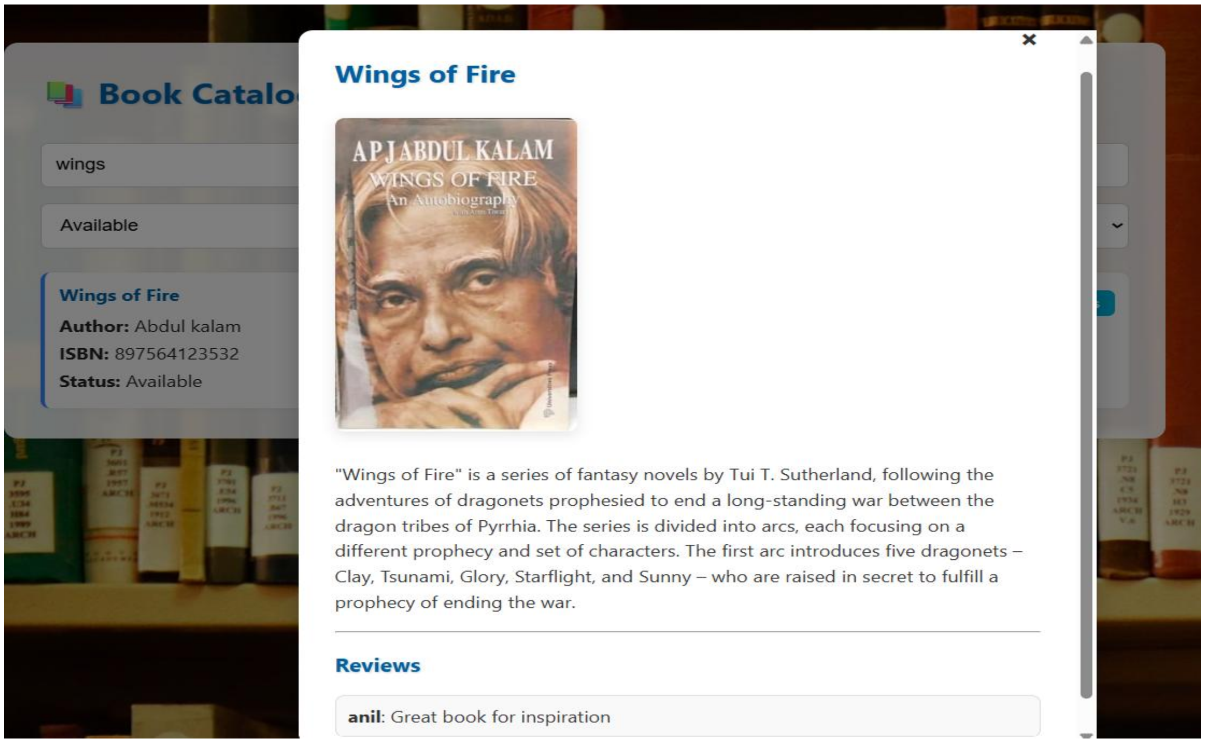
wings

Available

Wings of Fire
Author: Abdul kalam
ISBN: 897564123532
Status: Available

Figure 10: Book Catalog with Search Functionality

Step 4: Viewing Book Details and Existing Reviews - Before borrowing the book, the user decides to explore its details. Clicking on Details & Reviews opens a polished modal window showcasing the book's cover, description, and community feedback as shown in figure 11. Among the reviews, one from the same user, anil, reads: "Great book for inspiration." This interaction not only informs the user's decision but also highlights the platform's support for collaborative and informed reading choices.



Wings of Fire

Author: Abdul kalam
ISBN: 897564123532
Status: Available

Wings of Fire

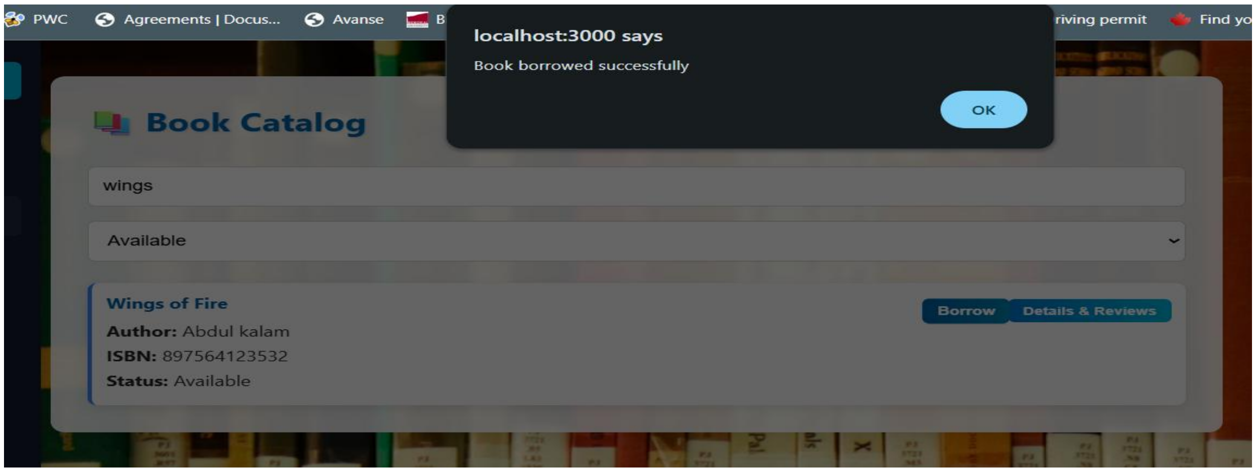
"Wings of Fire" is a series of fantasy novels by Tui T. Sutherland, following the adventures of dragonets prophesied to end a long-standing war between the dragon tribes of Pyrrhia. The series is divided into arcs, each focusing on a different prophecy and set of characters. The first arc introduces five dragonets – Clay, Tsunami, Glory, Starflight, and Sunny – who are raised in secret to fulfill a prophecy of ending the war.

Reviews

anil: Great book for inspiration

Figure 11: Book Details and User Reviews Modal

Step 5: Borrowing the Book - Convinced by the information provided, the user clicks the Borrow button. A confirmation alert appears: "Book borrowed successfully", indicating the transaction has gone through. Behind the scenes, the system updates the book's availability, adjusts the user's loan records, and reflects the change in the UI demonstrating a seamless connection between the frontend interface and backend logic.

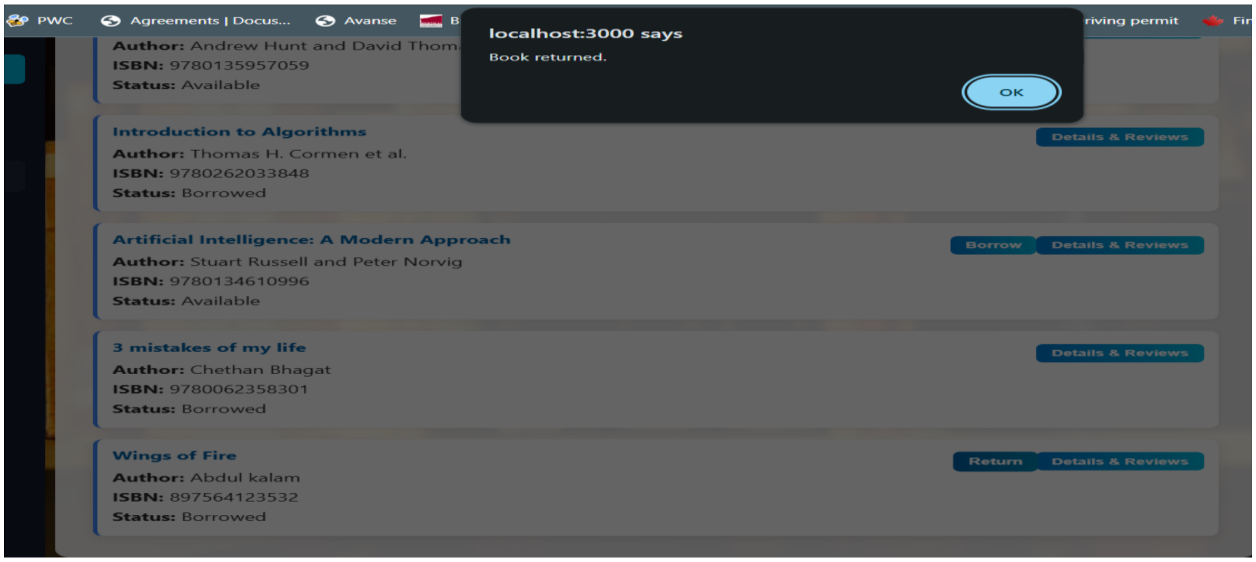


localhost:3000 says
Book borrowed successfully

Wings of Fire
Author: Abdul kalam
ISBN: 897564123532
Status: Available

Figure 12: Successful Book Borrow Confirmation

Step 6: Returning the Book - Once the book is read, the user revisits the catalog. The previously borrowed Wings of Fire now appears with a Return button instead of Borrow. Clicking it triggers another confirmation: "Book returned" as shown in figure 13. This response signals that the backend has updated the loan status, ensuring the system prevents multiple checkouts of the same book by the same user without a return in between.



localhost:3000 says

Book returned.

Author: Andrew Hunt and David Thom
ISBN: 9780135957059
Status: Available

Introduction to Algorithms
Author: Thomas H. Cormen et al.
ISBN: 9780262033848
Status: Borrowed

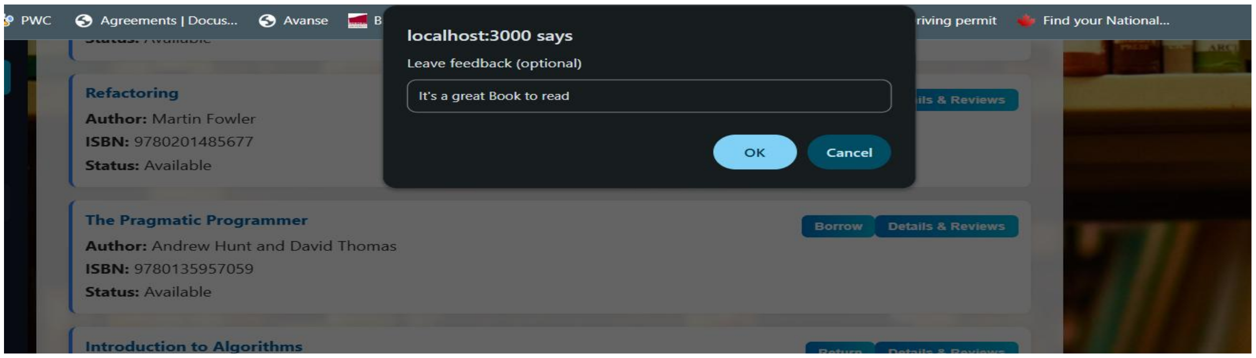
Artificial Intelligence: A Modern Approach
Author: Stuart Russell and Peter Norvig
ISBN: 9780134610996
Status: Available

3 mistakes of my life
Author: Chethan Bhagat
ISBN: 9780062358301
Status: Borrowed

Wings of Fire
Author: Abdul kalam
ISBN: 897564123532
Status: Borrowed

Figure 13: Book Return Confirmation

Step 7: Providing Feedback via Popup - Immediately after the return, the system prompts the user to leave feedback. A modal window appears, inviting optional input. The user enters feedback and submits it as in figure 14. The review is stored in the database and instantly becomes part of the Details & Reviews section. This final step closes the loop transforming a simple borrowing experience into a shared, evolving knowledge base for future readers.



localhost:3000 says

Leave feedback (optional)

It's a great Book to read

Refactoring
Author: Martin Fowler
ISBN: 9780201485677
Status: Available

The Pragmatic Programmer
Author: Andrew Hunt and David Thomas
ISBN: 9780135957059
Status: Available

Introduction to Algorithms

Figure 14: User Feedback Submission for a Book

This complete user cycle starting from login to finally submitting a review was executed smoothly without encountering any errors, broken UI elements, or functional issues. Each transition between steps was seamless, confirming that the system behaves reliably across both the front-end interfaces and back-end processes. Throughout the journey, users received clear visual cues and confirmation messages, contributing to an intuitive and positive experience.

In the same way, we performed exploratory testing across all other key functionalities within the application. By manually navigating through different user flows without relying on predefined test scripts we verified that each feature, from user management to fine payment, responded correctly under various input scenarios. This hands-on testing approach helped uncover subtle edge cases and ensured the entire system performs as expected under realistic usage conditions.

By combining unit, system, and exploratory testing, this layered verification and validation approach ensures that the Library Management System is not only functionally correct but also robust and ready for real-world use.

6. RESULTS & DISCUSSION

The Library Management System underwent thorough verification and validation through unit, system, and exploratory testing to ensure technical accuracy and a seamless user experience. Unit testing covered 35 cases, validating functions like book addition, user registration, borrowing, returns, and fines. Early issues with headers and ObjectId handling were resolved, and all tests passed, confirming reliable core logic. System testing verified end-to-end API flows, including admin user creation, loan management, fine processing, and feedback submission. Each route was tested for authorization, database accuracy, and error handling, confirming full-stack stability. Exploratory testing simulated real user behavior across key flows like login, search, borrowing, and review submission. The app consistently provided smooth interactions and accurate updates, reinforcing its usability and responsiveness. Overall, this multi-layered testing strategy validated the stability and readiness of the system.

7. CONCLUSION & FUTURE WORK:

By combining structured unit and system tests with flexible exploratory testing, we ensured coverage of both backend logic and frontend user experience. Techniques like mocking, test fixtures, and thoughtful scenario design allowed us to simulate real conditions without affecting production data. In conclusion, the Library Management System performs as intended, meeting its design goals and fulfilling all functional and non-functional requirements. It is stable, user-friendly, and ready for deployment. Continued integration of automated testing and exploratory feedback will support ongoing quality and maintainability.

While the system performed well during internal testing, future work should involve acceptance testing with real users, along with alpha and beta testing phases to gather valuable usability feedback. One limitation is that much of the testing relied on mocks and simulated data, meaning some real-world edge cases may only surface after deployment. Additionally, expanding test coverage to include UI accessibility and mobile responsiveness would further strengthen the system's reliability and user experience across different platforms.

8. REFERENCES

[1] D. Beyer, "State of the Art in Software Verification and Witness Validation: SV-COMP 2024," in *Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS vol. 14572, Apr. 2024, pp. 299–329. Springer, Cham. https://doi.org/10.1007/978-3-031-57256-2_15

[2] D. Beyer, M. Kettl, and T. Lemberger, "Fault Localization on Verification Witnesses," in *Proc. Int. Symp. on Model Checking Software (SPIN)*, LNCS vol. 14624, Apr. 2024, pp. 205–224. Springer, Cham. https://doi.org/10.1007/978-3-031-66149-5_12

[3] E. Ábrahám and H. Abbas, Eds., *Runtime Verification: 24th International Conference, RV 2024, Istanbul, Turkey, October 15–17, 2024, Proceedings*, LNCS vol. 15191. Springer, Cham, 2024. https://doi.org/10.1007/978-3-031-74234-7.

[4] Reddit user u/dwc, "Proposal: Native Design by Contract in Python via class invariants," *Reddit - r/Python*, Mar. 21, 2025. https://www.reddit.com/r/Python/comments/1jgdgob/proposal_native_design_by_contract_in_python_via/.

9. GIT REPOSITORY DETAILS:

The complete source code for this project is maintained on GitHub and is publicly accessible. The repository contains all frontend and backend code, along with necessary configuration and testing files. To run and check the code base please go through the Readme.md file.

- **GitHub Repo Link**: https://github.com/Anil-Manyam/Spring25
- **Branch Name**: verification