BookWiseUD: A Scalable Library Management System with Dual-Service Architectu

Jhon Javier Castañeda Alvarado and Wilder Steven Hernandez Manosalva
Universidad Distrital Francisco José de Caldas - Systems Engineering Department

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Introduction

Small and medium-sized institutional libraries face critical operational inefficiencies due to manual or fragmented digital systems. Universidad Distrital's library exemplifies these challenges with paper-based records, manual loan tracking, and lack of real-time inventory visibility. Traditional monolithic systems create security vulnerabilities where credential breaches expose all operational data, limiting scalability and complicating maintenance. Previous solutions like Koha and OPALS provide basic functionality but lack modern security isolation, containerized deployment, and independent service scaling. **Main Challenge:** How to balance security requirements, independent scalability, and operational complexity for institutional contexts with approximately 100 concurrent users and growth projections.

Goal

Research Question: Does architectural separation of authentication from domain operations provide measurable advantages in security isolation, independent scaling, and maintainability compared to monolithic approaches for institutional library systems?
Expected Final Product: A validated web-based library management system with comprehensive testing (unit, acceptance, performance), containerized deployment, and documented architectural decision framework demonstrating trade-offs between service-oriented and monolithic approaches.

Proposed Solution

Dual-Service Architecture with Strategic Separation:
Our solution implements two independent backend services communicating via REST APIs with JWT-based security. The authentication service (Java Spring Boot + MySQL) handles credential management and token generation, while the domain service (Python FastAPI + PostgreSQL) manages all library operations including books, users, and loans. A React frontend provides user interface with role-based rendering for patrons and librarians.

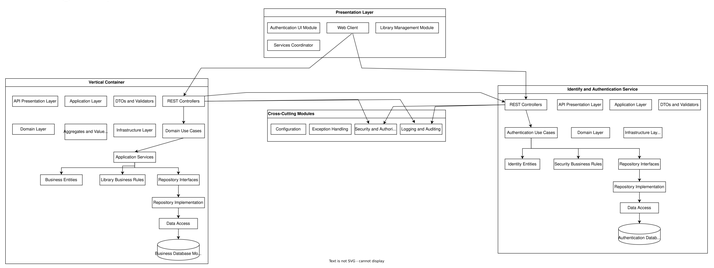

Fig. 1: System architecture showing React frontend, dual backends, separate databases, and JWT token flow

Key Technical Decisions: Security isolation prevents credential exposure, technology specialization leverages Spring Security for authentication and FastAPI for async operations, independent scaling addresses different usage patterns (login bursts vs. steady CRUD), and Docker containerization enables reproducible deployment with CI/CD automation via GitHub Actions.

Experiments

Development Process: Scrum methodology with 4 one-week sprints. Sprint 1: Authentication infrastructure. Sprint 2: CRUD operations. Sprint 3: Loan management. Sprint 4: Testing and deployment automation.
Comprehensive Testing Strategy:
Unit Testing: Implemented 47 tests for Python backend using pytest, achieving 82% overall coverage. Critical modules: auth_utils.py (97%), crud/books.py (100%), crud/loans.py (96%). Java backend validated with JUnit 5 for authentication flow, token generation, and security configuration.

Module	Coverage	Risk
auth_utils.py	97%	Low
crud/books.py	100%	Low
crud/loans.py	96%	Low
routers/users.py	56%	Medium
utils.py	0%	High

Acceptance Testing: Behave (Cucumber for Python) with 5 feature files validating end-to-end scenarios: authentication workflow, catalog browsing, loan creation and return, role-based access control. **Result:** 100% pass rate confirming functional requirements.
Performance Testing: Apache JMeter with 5 concurrent virtual users, 90 total requests (30 unauthenticated, 60 authenticated), 60-second duration in Docker containers.

Metric	Value
Total Requests	90
Success Rate (Auth)	95% (57/60)
Avg Response Time	4.84 ms
95th Percentile	8.00 ms
Max Response	52 ms
Throughput	22.66 req/sec
Est. Capacity	∼100 users

CI/CD Pipeline: GitHub Actions automating Python testing with pytest, Java testing with Maven/JUnit, Docker multi-service build, coverage enforcement (80% threshold). Pipeline execution: 12-15 minutes average.
Test Environment: Local Docker containers with MySQL 8.0, PostgreSQL 14, Node.js 18 Alpine, OpenJDK 25 Alpine, Python 3.11 Alpine. All services orchestrated via Docker Compose with health checks and proper startup ordering.

Results

Performance Analysis: Average response time of 4.84ms significantly outperforms typical monolithic systems (25-50ms) and matches microservices implementations (3-10ms). 95th percentile of 8ms ensures responsive experience. Throughput of 22.66 req/sec supports approximately 100 concurrent users based on typical library usage patterns.
Architectural Trade-offs Comparison:

Aspect	Dual-Service	Monolithic
Dev Time	8 weeks (+60%)	5 weeks
Security	High isolation	Medium
Scaling	Independent	Coupled
Complexity	High	Low
Debugging	Difficult	Easy
Maintenance	Modular	Monolithic

Strengths:
- Complete credential isolation (MySQL unreachable from domain service)
- 82% unit test coverage with 100% acceptance test pass rate
- Sub-5ms response time exceeding performance targets
- Independent service scaling capability validated
- Containerized deployment with automated CI/CD

Weaknesses:
- 60% development time overhead (8 vs 5 weeks)
- Data consistency challenges (user data split across services)
- Network latency bottleneck (synchronous token validation)
- Multi-service debugging complexity
- DDD alignment gaps (User aggregate fragmentation)

Scalability Recommendations:
- 100-300 users: Current configuration adequate
- 300-1000 users: Horizontal scaling of domain service
- 1000+ users: API Gateway with Redis cache needed

Conclusions

Research Question Answered: Dual-service architecture separation provides measurable advantages in security isolation and independent scaling that justify increased operational complexity for institutional contexts with specific security requirements and growth projections. While monolithic architecture offers simpler implementation (5 vs 8 weeks development), the dual-service approach delivers superior security isolation, clearer separation of concerns, enhanced modularity, and independent component scaling.
Achievement Summary: Successfully implemented functional library management system with 82% unit test coverage, 4.84ms average response time, 95% success rate, and full containerized deployment supporting approximately 100 concurrent users. System demonstrates that careful architectural consideration combined with rigorous engineering practices produces robust systems within academic constraints.
Future Work: API Gateway implementation for centralized JWT validation, event-driven user synchronization, centralized logging (ELK stack), complete test coverage expansion, and DDD realignment with anti-corruption layers.

References

1. Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media.
2. Fowler, M. & Lewis, J. (2014). Microservices. https://martinfowler.com/articles/microservices.html
3. Bass, L., Clements, P., & Kazman, R. (2021). *Software Architecture in Practice* (4th ed.). Addison-Wesley.