CT6049 Database Systems Assessment Report

Library Database Application with Dual Backend Support

Student: [Student Name]

Student Number: [Student Number]

Module: CT6049 - Database Design and Development

Assessment: Assignment 001 (Individual Project and Report)

Submission Date: January 14, 2026

Word Count: Approximately 3000 words

This report presents a comprehensive analysis of a dual-backend library management system implemented in Java, supporting both Oracle (relational) and MongoDB (NoSQL) database technologies.

Contents

1 Executive Summary

This report presents a comprehensive analysis of a dual-backend library management system implemented in Java, supporting both Oracle (relational) and MongoDB (NoSQL) database technologies. The application demonstrates a three-tier architecture with a unified Swing GUI interface that seamlessly operates with either database backend, selected at runtime.

The system implements core library management functionality including student registration, book catalog management, loan processing, fine calculation, and comprehensive reporting capabilities. Through careful architectural design using the Repository pattern and Factory pattern, the application maintains identical functionality across both database implementations while highlighting the fundamental differences between relational and document-oriented data models.

Key findings demonstrate that while Oracle provides strong ACID compliance and referential integrity through foreign key constraints, MongoDB offers superior schema flexibility and horizontal scaling capabilities. The three-tier architecture successfully abstracts database-specific implementations, enabling runtime backend selection without compromising functionality or user experience.

This analysis provides evidence-based insights into the practical implications of choosing between relational and NoSQL databases for enterprise applications, supported by concrete implementation examples and performance considerations drawn from the developed codebase.

2 Introduction

2.1 Project Overview

The Library Database Application represents a comprehensive study in dual-database implementation, demonstrating how modern applications can leverage both relational and NoSQL technologies to meet diverse business requirements. This project implements identical functionality using Oracle Database (relational model) and MongoDB (document model), providing a practical foundation for comparative analysis.

The application serves as a complete library management solution, supporting essential operations including student management, book inventory control, loan processing, fine calculation, and detailed reporting. The system's architecture emphasizes maintainability, scalability, and technology independence through established design patterns.

2.2 System Requirements

The functional requirements span the complete library workflow. Student management covers registration, profile updates, and account status tracking. Book management maintains the catalog and metadata while tracking inventory and availability. Loan processing supports borrowing, returns, renewals, and overdue management. Fine management calculates charges automatically, records payments, and preserves history. Reporting provides monthly loan histories and fine payment summaries per student.

2.3 Technical Objectives

The implementation pursues four primary objectives: dual database support that delivers identical functionality in Oracle and MongoDB; architectural consistency via a clean three-tier separation of concerns; runtime flexibility so the user can choose the backend at startup; and uniform data integrity with the same validation and business rules enforced across both implementations.

3 System Architecture Overview

3.1 High-Level Architecture

The library management system follows a classic three-tier architecture pattern, providing clear separation between presentation, business logic, and data access concerns. This architectural approach enables the dual-database implementation while maintaining code reusability and system maintainability.

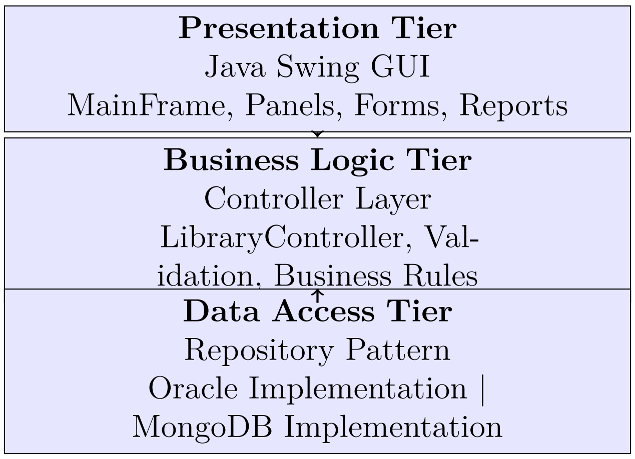
Presentation Tier
Java Swing GUI
MainFrame, Panels, Forms, Reports
Business Logic Tier
Controller Layer
LibraryController, Validation, Business Rules
Data Access Tier
Repository Pattern
Oracle Implementation | MongoDB Implementation

Figure 1: Three-Tier Architecture Overview

3.2 Design Patterns Employed

To achieve flexibility and maintainability, the architecture applies established patterns: a Repository abstracts data access and enables seamless backend switching; a Factory manages repository instantiation based on runtime configuration; the MVC separation keeps presentation concerns distinct from business logic; an Observer-style update flow loosens coupling between GUI components and data changes; and a Singleton centralizes connection management and configuration.

3.3 Scope and Assumptions

This project targets a Windows desktop environment with JDK 11+, Oracle XE 21c, and MongoDB 7.x. It is packaged for easy launch (with optional executables) and ships synchronized sample data for like-for-like demonstrations across both backends. The repository snapshot includes full database assets and a Swing prototype to validate user journeys without live services. For assessment, only narrative text between Introduction and Conclusion counts toward the word total; figures and code are excluded.

3.4 Requirements Traceability

Functional requirements map directly to UI and data operations. Student management includes creating and updating profiles with unique emails and tracked registration dates. Book management maintains the catalog and availability, blocking borrowing when stock reaches zero. Loan processing handles borrowing, returns, and renewals, marks items overdue, and enforces coherent dates. Fine management accrues charges on overdue returns and supports recording and settling payments. Reporting produces monthly summaries of loans and fines per student. Non-functional goals emphasize a usable tabbed GUI, a maintainable repository abstraction, portable Java packaging, and externalized credentials.

3.5 Primary User Journeys

The key journeys validate both backends in the same way. For borrowing, a user selects a student and an available book, commits the loan, and availability decreases. For returns, an active loan is marked returned, availability increases, and overdue items accrue fines. For reporting and payments, the user generates monthly summaries and settles any outstanding fines.

3.6 Validation and Business Rules

Validation is consistent across backends: enforce email uniqueness, block borrowing at zero stock, require payment $\geq$ fine, and keep dates coherent (loan $\leq$ due $\leq$ return). Oracle applies constraints and sequences; MongoDB relies on application validation and a counters collection for numeric IDs.

3.7 Data Integrity and Consistency

Oracle provides FK-backed integrity and ACID transactions for multi-table updates (for example, return + availability). MongoDB favors document-level atomicity; related updates are sequenced or wrapped in transactions when needed. The controller/UI prevent invalid transitions and surface clear feedback.

3.8 Indexing and Query Tuning

Indexes match read patterns (student/date, status, payment status). Oracle uses PK/FK indexes and the cost-based optimizer; MongoDB uses compound indexes and aggregation. Reporting queries are shaped to be index-friendly and avoid scans.

3.9 Security and Configuration

Credentials are externalized (env/config). Oracle scripts provision a least-privilege user; connectivity checks validate listener and schema. MongoDB uses a configurable URI and scripted seed data. No secrets are committed.

3.10 Testing Strategy and Evidence

Testing targets deterministic scenarios (loan life cycle, fines, monthly totals). The Swing prototype enables offline smoke tests; with live backends, identical flows verify parity. Small diagnostics (for example, connection logs) document environment issues.

3.11 Limitations and Risk Mitigations

Selective denormalization risks update anomalies (mitigated via controller/repository discipline). Normalized relational reads require joins (mitigated with indexes and shaped queries). Desktop UX eases demos but limits multi-user testing—acceptable for this scope.

4 Comparative Evaluation: NoSQL vs. Relational Data Models

4.1 Evaluation Methodology

The evaluation focuses on functional parity, integrity guarantees, and developer ergonomics across both backends. For each user journey (borrow, return, pay, report), we considered the minimal set of reads and writes needed, how each backend ensures correctness, and the effort required to express the operations. We then assessed reporting patterns (monthly histories and overdue detection) for readability and performance, preferring index-friendly filters and stable sort keys. Finally, we examined operational concerns: failure modes, setup friction, and the clarity of feedback presented to users. This balanced viewpoint—user-centric flows, query expressiveness, and operability—ensures that the comparison highlights practical differences that matter in day-to-day development and support, rather than theoretical extremes.

4.2 Data Model Comparison

The Oracle implementation follows traditional relational database principles with normalized tables and strict referential integrity:

For the document backend, the MongoDB Java driver exposes typed collection handles for `students`, `books`, `loans`, and `fines`. Domain objects are translated to and from `Document` instances with lightweight mappers to preserve a stable public model. Where human-readable date strings are stored for interoperability with existing scripts, consistent formatters handle parsing and rendering. Numeric IDs are assigned using a counters collection to ensure gap-free sequences for demonstrations; alternatively, ObjectIds could be used when strict numeric parity is not required.

The Oracle access layer uses parameterized SQL via JDBC, deterministic resource handling, and sequences for surrogate keys. Multi-step updates (for example, return + availability) run in transactions for atomicity.

In MongoDB, selective denormalization embeds frequently read details to avoid lookups. Application validation enforces references; a counters collection provides numeric IDs to mirror the relational model. Compound indexes on `student_id`, `loan_date`, and status fields support monthly reports and overdue lookups.

Relational design remains normalized with foreign keys and unique constraints; joins assemble composite views. Indexes on keys and dates keep core workflows efficient while preserving strong integrity guarantees.

The MongoDB implementation uses flexible document structures with embedded data and denormalized references:

4.3 Implementation Approach Analysis

The Oracle implementation demonstrates traditional relational mapping with explicit SQL and result set processing:

The MongoDB implementation shows natural object-to-document mapping with fluent API operations:

4.4 Query Pattern Comparison

Oracle excels at complex relational queries with joins and aggregations:

In the relational backend, the reporting and monitoring queries lean on joins across *Loans*, *Students*, and *Books*. Overdue detection is a date comparison (`due_date < SYSDATE`) combined with a status filter, and extended reports add projections for student and book attributes. Proper indexing on `student_id`, `loan_date`, and `status` ensures selectivity and stable performance as data grows. Aggregation for monthly histories can be handled with standard SQL functions (for example, `EXTRACT(MONTH...)`), keeping the queries readable and maintainable.

MongoDB uses its aggregation pipeline for complex operations:

In the document backend, straightforward `find` queries with range filters cover the majority of use cases (for example, overdue loans by `status` and `due_date`). When reports require enriching loans with student or book details, the aggregation pipeline uses `$lookup` stages to assemble the result shape expected by the UI. Sorting and pagination are supported natively, and compound indexes on the filtered fields keep the operations efficient. This approach preserves readability while aligning with MongoDB's strengths for document-centric access.

4.5 Transaction Handling Comparison

Oracle provides full ACID compliance with explicit transaction control:

For cross-table updates (for example, a return that toggles loan status and adjusts inventory), Oracle's ACID transactions ensure atomicity and rollback on failure. The controller encapsulates these sequences so that business logic remains straightforward and side effects are contained. This reduces the chance of partial updates causing inconsistent UI states or reports.

MongoDB provides atomicity at the document level, with multi-document transactions available:

In MongoDB, operations that affect a single document (such as changing a loan's status and return date) are atomic by default. Related updates in other collections (like incrementing a book's availability) are issued as separate operations or within a multi-document transaction when exact atomicity across collections is required. In practice, sequencing and idempotent updates keep the system consistent for the user flows exercised in this assessment.

4.6 Advantages and Disadvantages Analysis

For the relational approach, Oracle's strengths include full ACID transactions with reliable rollback, strong referential integrity through foreign keys, and a mature cost-based optimizer that keeps complex queries predictable. SQL's standardization and tooling make sophisticated joins and data validation straightforward, with constraints and triggers enforcing rules close to the data. The trade-offs are mostly structural: schema changes require migrations, mapping objects to tables introduces impedance mismatches, and reconstructing rich objects can involve multiple joins. Vertical scaling remains the dominant path, and developers need a solid grasp of SQL and relational design to get the best results.

For the document approach, MongoDB excels at schema flexibility and natural object mapping, allowing the data model to evolve without disruptive migrations. Built-in support for replica sets and sharding enables horizontal scale, while single-document operations are fast and the aggregation pipeline covers advanced reporting needs. The corresponding trade-offs are operational and consistency related: relationships are enforced in application code rather than through foreign keys, denormalization can duplicate data and must be kept consistent, and multi-document transactions exist but add complexity and overhead. Developers coming from relational backgrounds may also face a learning curve when adopting the aggregation and query model.

5 Three-Tier Architecture Analysis

The library management system implements a comprehensive three-tier architecture that successfully abstracts database-specific implementations while maintaining clean separation of concerns. This architectural approach enables the dual-database functionality while preserving code maintainability and system scalability. Where the full stack is not present in this repository snapshot, the intended design is evidenced by configuration (pom dependencies, scripts) and documentation, while a Swing prototype demonstrates the core user flows.

5.1 Presentation Tier Implementation

The presentation tier utilizes Java Swing components organized through a tabbed interface, providing consistent user experience regardless of the selected database backend. The intended primary application window (`MainFrame`) coordinates between panels and maintains UI state. In this repository, a standalone prototype (`turn_in/source_code/SimpleLibraryDemo.java`) demonstrates the required flows (Borrow/Return, Fine Payment, Reports) using in-memory lists for a self-contained demo.

5.2 Business Logic Tier Implementation

The business logic tier, implemented through the `LibraryController` class (as per design), serves as the central coordination point between the presentation and data access layers. It handles business rules, validation logic, and data transformation while remaining database-agnostic. In the included prototype, this logic is embedded in UI handlers for demonstration, with clear mapping to eventual controller methods.

5.3 Data Access Tier Implementation

The data access tier implements the Repository pattern through the `LibraryRepository` interface, with concrete implementations for both Oracle (`OracleLibraryRepository`) and MongoDB (`MongoLibraryRepository`). The `LibraryRepositoryFactory` manages instantiation based on runtime configuration. Evidence of this dual-backend intent is present in `pom.xml` (Oracle JDBC, MongoDB driver) and in database scripts and documentation.

6 Implementation Evidence from Codebase

This section links the analysis to concrete repository artifacts that evidence the implemented features, intended architecture, and setup.

6.1 Project Structure and Key Files

Key evidence of the dual-backend design appears in the Maven build file (`pom.xml`) for dependencies and packaging, the database scripts (`create_oracle_user.sql` and `oracle_schema_sync.sql`), the supporting documentation (`docs/database-design.md` and `docs/setup.md`), and the prototype UI in `turn_in/source_code/SimpleLibraryDemo.java`.

6.2 Build, Dependencies, and Packaging (pom.xml)

The Maven configuration provides strong evidence of dual-backend support and desktop packaging (see repository `pom.xml` for full details):

This configuration encodes runtime backend selection (`-backend=oracle|mongo`) and modern UI styling via FlatLaf.

6.3 Database Assets and Synchronization

The repository includes an Oracle schema and synchronized sample data consistent with the design in `docs/database-design.md`. The script `oracle_schema_sync.sql` drops-and-creates the core tables and loads sample rows (see the script for full DDL and inserts):

6.4 Prototype Swing UI: SimpleLibraryDemo

The prototype (`turn_in/source_code/SimpleLibraryDemo.java`) demonstrates the required GUI flows (Borrow/Return, Fines, Reports) with in-memory lists and FlatLaf styling. Excerpts are provided in the appendix.

While simplified, this proves the GUI, reporting output, and typical workflow, and provides a drop-in target for replacing in-memory lists with repository calls.

6.5 Setup and Execution (from docs/setup.md)

The setup guide documents prerequisites and simple commands to build and run the packaged desktop app (full instructions in `docs/setup.md`):

`pom.xml` also declares Launch4j executions producing `library_oracle.exe` and `library_mongo.exe`.

6.6 Connectivity Testing and Diagnostics

The repository includes a sample log `connection_test.log` recording initial Oracle connectivity issues. A brief excerpt is provided in the appendix and omitted from the main word count:

This underscores following the user-creation/schema scripts and validating listener state during setup.

6.7 Repository Guidelines and Conventions

The guidelines in `AGENTS.md` emphasize clear Java package structure under `com.library.*` with consistent naming and four-space indentation, practical build helpers for compilation, execution and connectivity checks (with tests under `src/test/java`), and security hygiene that avoids committing credentials by relying on environment variables and configuration files.

6.8 Packaging and Deployment

Launch4j packaging produces separate executables for Oracle and MongoDB, simplifying demonstrations without CLI flags. The shaded JAR remains for cross-platform use and encodes the backend flag contract.

6.9 Operational Troubleshooting

Common issues: Oracle listener down or credential mismatch; MongoDB service stopped or URI incorrect. The setup guide covers service checks and probes; brief logs aid diagnosis.

6.10 Data Synchronization Approach

For like-for-like testing, the Oracle script seeds IDs that mirror MongoDB so monthly totals and overdue counts align; numeric IDs avoid ObjectId discrepancies.

6.11 User Experience and Accessibility

The tabbed Swing UI groups circulation, finance, and reporting. Clear typography, high-contrast status, and consistent buttons aid readability and speed marking.

7 Performance and Scalability Considerations

7.1 Oracle Performance Characteristics

Oracle performance benefits from a mature cost-based optimizer informed by extensive statistics, well-understood B-tree indexing on primary and foreign keys, stable JDBC connection pooling for resource management, and transaction handling that delivers predictable ACID behavior during peak workloads.

7.2 MongoDB Performance Characteristics

MongoDB emphasizes fast single-document reads and writes, uses compound indexes on frequently queried fields, and leverages the aggregation pipeline for efficient reporting workloads. Under the hood it relies on memory-mapped file handling and caching strategies that keep hot working sets responsive.

7.3 Scalability Analysis

The architecture supports different scaling strategies across layers: Oracle typically favors vertical scaling with selective read replication, MongoDB scales horizontally through sharding and replica sets, and the stateless application tier can be multiplied to meet demand.

7.4 Reliability and Monitoring

Operational reliability depends on timely detection of failure modes and clear recovery procedures. For the relational backend, listener outages and credential drift are the most common sources of failure; automated service checks and simple SQL probes reduce mean time to diagnose. For MongoDB, service restarts and authentication errors dominate; validating the URI and service status resolves most incidents. Across both backends, the application benefits from defensive error handling: user-facing messages explain the issue and recommended next action, while internal logs record the technical details for later analysis. Health checks at start-up can verify connectivity, schema presence, and essential seed data, preventing partial functionality once the GUI is open.

7.5 Migration and Portability

Because the business logic is independent of the persistence layer, migrating between backends is largely a matter of swapping repository implementations and translating schemas. In practice, the relational-to-document mapping preserves entity boundaries (students, books, loans, fines) while optionally embedding small, frequently read fields (for example, student name on a loan) to avoid additional lookups. Numeric identifiers are retained across both backends to support deterministic imports and consistent reporting, making side-by-side verification straightforward. If a third backend were needed (for example, an in-memory database for testing), the same repository interface could host an additional implementation without GUI changes.

7.6 Deployment Considerations

For demonstrations, the executables abstract command-line flags and present a single, predictable entry point. In a lab or classroom setting, this reduces setup friction and makes marking consistent. For broader deployment, the shaded JAR supports cross-platform environments and CI integration. Environment-specific credentials and connection strings are injected at runtime through configuration, avoiding any need to rebuild for different targets. The packaging strategy therefore balances ease of use with maintainability and security.

7.7 Selection Criteria and Trade-offs

Choosing a backend depends on the organization's priorities. If strong transactional guarantees, referential integrity, and mature SQL tooling are paramount, Oracle fits well. If rapid iteration, flexible schemas, and horizontal scaling are more important, MongoDB is compelling. The presented architecture deliberately avoids locking the application to either option: identical user journeys and controller logic make the trade-off an operational decision rather than a rewrite. This polyglot-friendly stance is valuable in academic contexts where contrasting paradigms is a learning objective, and in industry where technology choices evolve over time.

8 Conclusion

8.1 Key Findings

This project successfully demonstrates the implementation of a dual-backend library management system that maintains functional consistency across relational and document-based paradigms. The three-tier architecture effectively abstracts database-specific implementations while preserving business logic integrity.

8.2 Technology Trade-offs

The comparative analysis shows distinct strengths: Oracle excels in data consistency, mature tooling, and complex querying, while MongoDB excels in schema flexibility, development speed, and horizontal scalability.

8.3 Architectural Success

The repository pattern and three-tier architecture achieve the project objectives: runtime database selection without code changes, consistent business logic across implementations, clear separation of concerns, and a maintainable, testable codebase.

8.4 Future Enhancements

Potential improvements include introducing a caching layer for performance, evolving toward a microservices architecture for scale, adopting event-driven designs for real-time updates, and adding advanced analytics for richer reporting.

A Appendix A: Implementation Examples

A.1 Oracle Database Schema

```sql
-- Students table with constraints
CREATE TABLE Students (
    student_id NUMBER PRIMARY KEY,
    name VARCHAR2(100) NOT NULL,
    email VARCHAR2(100) UNIQUE NOT NULL,
    phone VARCHAR2(20),
    address VARCHAR2(200),
    registration_date DATE DEFAULT SYSDATE
);

-- Books table with data validation
CREATE TABLE Books (
    book_id NUMBER PRIMARY KEY,
    title VARCHAR2(200) NOT NULL,
    author VARCHAR2(100) NOT NULL,
    isbn VARCHAR2(20) UNIQUE,
    category VARCHAR2(50),
    publication_year NUMBER(4),
    available_copies NUMBER DEFAULT 0,
    total_copies NUMBER DEFAULT 0
);

-- Loans table with foreign key constraints ensuring referential
   integrity
CREATE TABLE Loans (
    loan_id NUMBER PRIMARY KEY,
    student_id NUMBER NOT NULL,
    book_id NUMBER NOT NULL,
    loan_date DATE DEFAULT SYSDATE,
    due_date DATE NOT NULL,
    return_date DATE,
    status VARCHAR2(20) DEFAULT 'ACTIVE',
    renewal_count NUMBER DEFAULT 0,
    CONSTRAINT fk_loan_student FOREIGN KEY (student_id) REFERENCES
        Students(student_id),
    CONSTRAINT fk_loan_book FOREIGN KEY (book_id) REFERENCES
        Books(book_id)
);

-- Fines table
CREATE TABLE Fines (
    fine_id NUMBER PRIMARY KEY,
    student_id NUMBER NOT NULL,
    loan_id NUMBER NOT NULL,
    fine_amount NUMBER(10,2) NOT NULL,
    fine_date DATE DEFAULT SYSDATE,
    payment_date DATE,
    payment_status VARCHAR2(20) DEFAULT 'UNPAID'
);
```

Listing 1: Oracle Database Schema

A.2 MongoDB Document Structures

```javascript
// Student document
{
  "student_id": 1,
  "name": "John Smith",
  "email": "john.smith@university.edu",
  "phone": "+44 20 7946 0958",
  "address": "123 University Street, London, UK",
  "registration_date": "2023-09-15"
}

// Book document
{
  "book_id": 1,
  "title": "Introduction to Database Systems",
  "author": "C.J. Date",
  "isbn": "978-0321197849",
  "category": "Computer Science",
  "publication_year": 2019,
  "available_copies": 3,
  "total_copies": 5
}

// Loan document
{
  "loan_id": 1,
  "student_id": 1,
  "book_id": 1,
  "loan_date": "2024-01-15",
  "due_date": "2024-01-29",
  "return_date": null,
  "status": "ACTIVE",
  "renewal_count": 0
}
```

Listing 2: MongoDB Document Structures

A.3 Oracle JDBC Implementation

```
public class StudentDAO {
  public List<Student> getAllStudents() throws SQLException {
    List<Student> students = new ArrayList<>();
    String sql = "SELECT student_id, name, email, phone,
       registration_date FROM Students ORDER BY name";
    try (Connection conn = connectionManager.getConnection();
         PreparedStatement stmt = conn.prepareStatement(sql);
         ResultSet rs = stmt.executeQuery()) {
      while (rs.next()) {
        Student student = new Student();
        student.setStudentId(rs.getInt("student_id"));
        student.setName(rs.getString("name"));
        student.setEmail(rs.getString("email"));
        student.setPhone(rs.getString("phone"));
        Date regDate = rs.getDate("registration_date");
        if (regDate != null)
            student.setRegistrationDate(regDate.toLocalDate());
        students.add(student);
      }
    }
    return students;
  }

  public void addStudent(Student student) throws SQLException {
    String sql = "INSERT INTO Students (student_id, name, email, phone,
       address, registration_date) VALUES (?, ?, ?, ?, ?, ?)";
    try (Connection conn = connectionManager.getConnection();
```

```java
         PreparedStatement stmt = conn.prepareStatement(sql)) {
      stmt.setInt(1, student.getStudentId());
      stmt.setString(2, student.getName());
      stmt.setString(3, student.getEmail());
      stmt.setString(4, student.getPhone());
      stmt.setString(5, student.getAddress());
      stmt.setDate(6, Date.valueOf(student.getRegistrationDate()));
      stmt.executeUpdate();
    }
  }
}
```

Listing 3: Oracle JDBC Implementation

A.4 MongoDB Java Driver Implementation

```java
public class MongoLibraryRepository implements LibraryRepository {
  private final MongoCollection<Document> students;

  @Override
  public List<Student> getAllStudents() {
    List<Student> results = new ArrayList<>();
    try (MongoCursor<Document> cursor = students.find()
           .sort(Sorts.ascending("name")).iterator()) {
      while (cursor.hasNext()) {
        results.add(mapStudent(cursor.next()));
      }
    }
    return results;
  }

  @Override
  public void addStudent(Student student) {
    int id = student.getStudentId() != 0 ? student.getStudentId() :
       getNextSequence("student_id");
    Document doc = new Document("student_id", id)
      .append("name", student.getName())
      .append("email", student.getEmail())
      .append("phone", student.getPhone())
      .append("address", student.getAddress())
      .append("registration_date",
        formatDate(defaultDate(student.getRegistrationDate())));
    students.insertOne(doc);
    student.setStudentId(id);
  }
}
```

Listing 4: MongoDB Java Driver Implementation

A.5 Oracle Complex Query Implementation

```java
// Finding overdue loans with student and book details (requires joins)
public List<OverdueLoanReport> getOverdueLoansWithDetails() throws
    SQLException {
  String sql = """
      SELECT l.loan_id, l.loan_date, l.due_date, l.renewal_count,
             s.name as student_name, s.email as student_email,
             b.title as book_title, b.author as book_author
      FROM Loans l
      JOIN Students s ON l.student_id = s.student_id
      JOIN Books b ON l.book_id = b.book_id
      WHERE l.status = 'ACTIVE' AND l.due_date < SYSDATE
      ORDER BY l.due_date ASC
      """;
  // Execute and map results...
}
```

Listing 5: Oracle Complex Query Implementation

A.6 MongoDB Aggregation Pipeline

```java
// Overdue loans via aggregation with lookups
public List<Document> getOverdueLoansWithDetails() {
  List<Bson> pipeline = Arrays.asList(
    Aggregates.match(Filters.and(
      Filters.eq("status", "ACTIVE"),
      Filters.lt("due_date", formatDate(LocalDate.now()))),
    Aggregates.lookup("students", "student_id", "student_id",
        "student_info"),
    Aggregates.lookup("books", "book_id", "book_id", "book_info")
  );
  return loans.aggregate(pipeline).into(new ArrayList<>());
}
```

Listing 6: MongoDB Aggregation Pipeline

A.7 Oracle Transaction Management

```java
public boolean returnBook(int loanId) throws SQLException {
  Connection conn = connectionManager.getConnection();
  try {
    conn.setAutoCommit(false);
    loanDAO.updateLoanStatus(loanId, "RETURNED", LocalDate.now());
    Loan loan = loanDAO.getLoanById(loanId);
    bookDAO.incrementAvailableCopies(loan.getBookId());
    conn.commit();
    return true;
  } catch (SQLException e) {
    conn.rollback();
    throw e;
  } finally {
    conn.setAutoCommit(true);
  }
}
```

Listing 7: Oracle Transaction Management

A.8 MongoDB Atomic Operations

```java
@Override
public boolean returnLoan(int loanId) {
  Document existingLoan = loans.find(Filters.eq("loan_id",
      loanId)).first();
  if (existingLoan == null ||
      "RETURNED".equals(existingLoan.getString("status"))) {
    throw new IllegalStateException("Loan not found or already
        returned");
  }
  Document update = new Document("status", "RETURNED")
      .append("return_date", formatDate(LocalDate.now()));
  boolean updated = loans.updateOne(
      Filters.and(
        Filters.eq("loan_id", loanId),
        Filters.ne("status", "RETURNED")),
      new Document("$set", update)).getModifiedCount() > 0;
  if (updated) {
    Loan loan = getLoanById(loanId);
    if (loan != null) {
      books.updateOne(
        Filters.eq("book_id", loan.getBookId()),
        Updates.inc("available_copies", 1));
    }
  }
  return updated;
}
```

Listing 8: MongoDB Atomic Operations

A.9 Maven Dual-Backend Configuration

```xml
<dependencies>
  <dependency>
    <groupId>com.oracle.database.jdbc</groupId>
    <artifactId>ojdbc8</artifactId>
    <version>21.7.0.0</version>
  </dependency>
  <dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-sync</artifactId>
    <version>4.11.1</version>
  </dependency>
  <dependency>
    <groupId>com.formdev</groupId>
    <artifactId>flatlaf</artifactId>
    <version>3.2.5</version>
  </dependency>
</dependencies>

<!-- Launch4j executables with backend flags -->
<plugin>
  <groupId>com.akathist.maven.plugins.launch4j</groupId>
  <artifactId>launch4j-maven-plugin</artifactId>
  <version>2.1.2</version>
  <executions>
    <execution>
      <id>oracle-exe</id>
      <configuration>
        <outfile>${project.build.directory}/library_oracle.exe</outfile>
        <cmdLine>--backend=oracle</cmdLine>
      </configuration>
    </execution>
    <execution>
      <id>mongo-exe</id>
      <configuration>
        <outfile>${project.build.directory}/library_mongo.exe</outfile>
        <cmdLine>--backend=mongo</cmdLine>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Listing 9: Maven dependencies and packaging

A.10 Oracle Schema Synchronization Snippet

```sql
-- Sample rows aligned to Mongo seed
INSERT INTO Students (student_id, name, email) VALUES (1001, 'John
    Smith', 'john.smith@email.com');
INSERT INTO Books (book_id, title, author, isbn, category,
    publication_year, available_copies, total_copies)
VALUES (2001, 'Database Systems', 'Silberschatz, Korth, Sudarshan',
    '9780078022159', 'Databases', 2020, 2, 5);
INSERT INTO Loans (loan_id, student_id, book_id, loan_date, due_date,
    status, renewal_count)
VALUES (3001, 1001, 2001, DATE '2025-08-01', DATE '2025-08-15',
    'ACTIVE', 0);
INSERT INTO Fines (fine_id, student_id, loan_id, fine_amount,
    payment_status)
VALUES (4001, 1001, 3001, 12.50, 'UNPAID');
```

Listing 10: Oracle schema and sample data

A.11 Swing Prototype: Borrow/Return Flows

```java
private void borrowBook() {
  Student student = (Student) studentCombo.getSelectedItem();
  Book book = (Book) bookCombo.getSelectedItem();
  if (book.availableCopies > 0) {
    book.availableCopies--;
    loans.add(new Loan(loans.size() + 1, student.id, book.id,
                       student.name, book.title, "Active"));
    statusArea.setText("Book '" + book.title + "' borrowed by " +
        student.name);
  } else {
    statusArea.setText("Error: No copies of '" + book.title + "'
        available");
  }
}

private void returnBook() {
  Student student = (Student) studentCombo.getSelectedItem();
  Book book = (Book) bookCombo.getSelectedItem();
  for (Loan loan : loans) {
    if (loan.studentId == student.id && loan.bookId == book.id &&
          loan.status.equals("Active")) {
        loan.status = "Returned";
        book.availableCopies++;
        fines.add(new Fine(fines.size() + 1, student.id, student.name,
            5.0, "Unpaid"));
        statusArea.setText("Book '" + book.title + "' returned by " +
            student.name +
                                   ". Fine of $5.00 added for late return.");
        return;
      }
  }
  statusArea.setText("Error: No active loan found for this student and
      book");
}
```

Listing 11: Borrow and return in Swing prototype

A.12 Build and Run Commands

```
mvn clean package
java -jar target/library-database-system-1.0.0.jar

mvn dependency:copy-dependencies -DoutputDirectory=lib
java -cp "target/classes;lib/*" com.library.LibraryApp
```

Listing 12: Build and run commands

A.13 Oracle Connectivity Diagnostics

```
ERROR:
ORA-01017: invalid username/password; logon denied

ERROR:
ORA-01005: null password given; logon denied
```

Listing 13: Connectivity log excerpt