SYD466V1A - Assignment 3

Public Library Booking System (FMO System)

Final System Design & Implementation Plan

Version 1.0.0

Luca Novello

TABLE OF CONTENTS

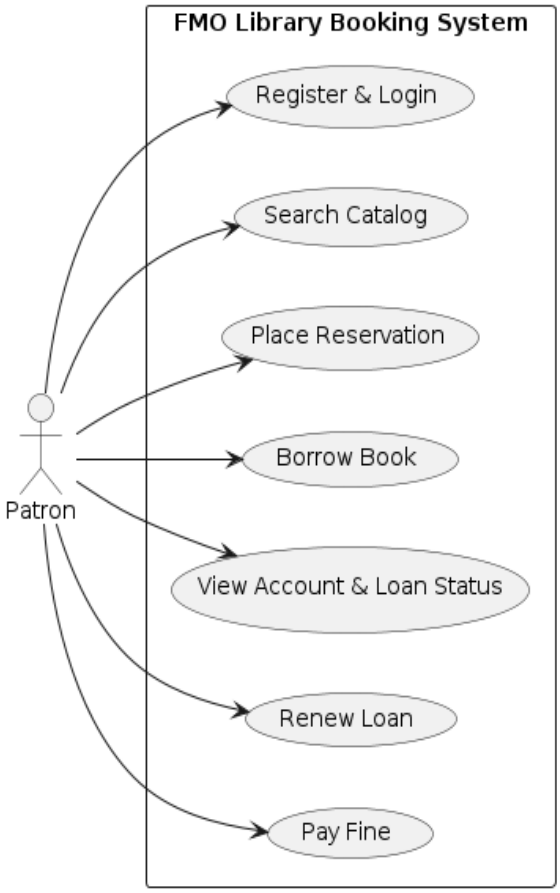
1. SELECTION OF DESIGN AND DELIVERY

Based on the vendor evaluation, Vendor A was selected due to their high scores in technical quality, cost, and relevant experience. Their proposed methodology uses Agile development with bi-weekly sprints, enabling iterative feedback from stakeholders. This method aligns well with the project's goals of reliability, adaptability, and timely delivery.

For system delivery, the team will use a three-tier architecture built with React.js (frontend), Node.js/Express (backend), and MongoDB Atlas (database), hosted on AWS. This cloud-based, modular approach ensures scalability and supports integration with services like Stripe and Google Books API.

2. FINAL SYSTEM DESIGN SPECIFICATION (SDS)

2.1 USE CASE DIAGRAM

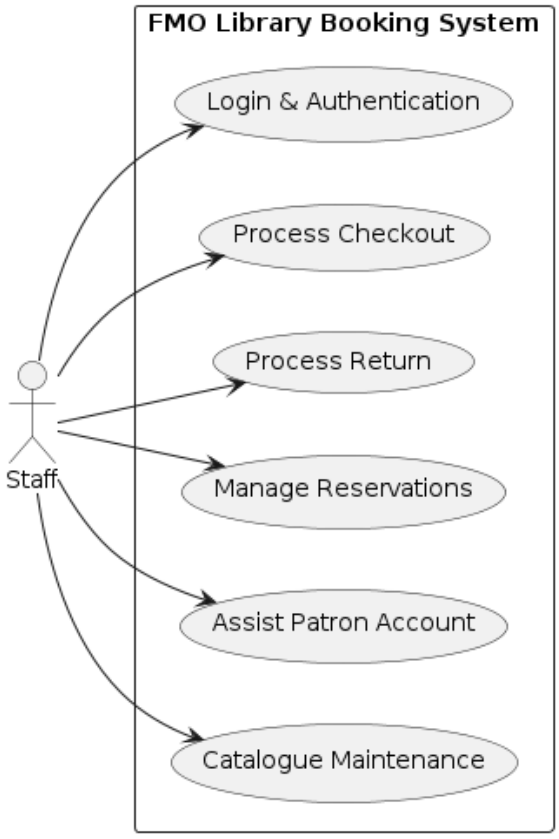


2.1.1 PATRON USE CASES:

Users will have access to:

- User account access
- Catalog search
- Placing and managing reservations
- Borrowing and renewing items
- Account and loan statuses
- Paying fines online.

These use cases reflect the system’s emphasis on improving the user experience through self-service features and digital convenience, providing a modern, accessible library interface.



2.1.2 STAFF USE CASES:

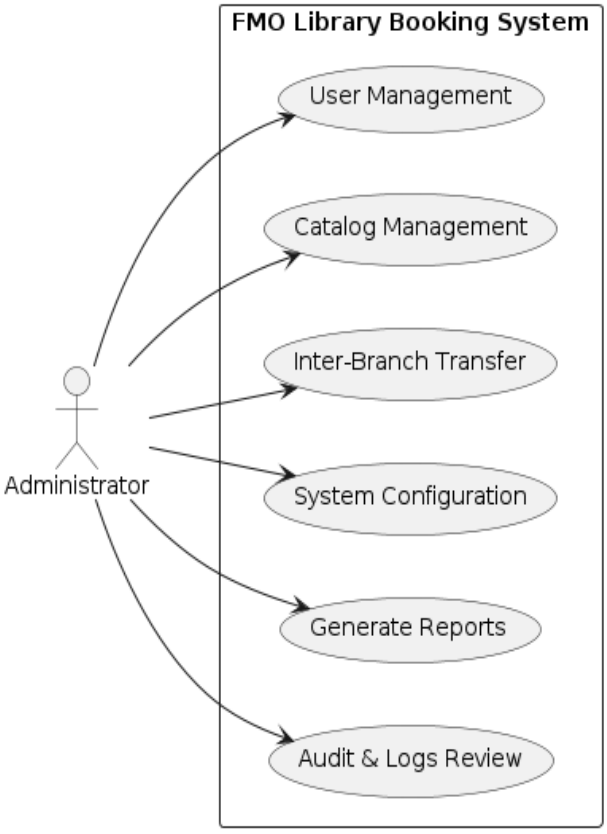
Staff will be responsible for:

- Managing book transactions
- Supporting patron accounts
- Maintaining catalog records

Staff will have access to:

- Process sales, loans and returns
- Manage reservations
- Assist patrons with accounts
- Update and maintain the catalogue

These ensure smooth day-to-day operations and support operational efficiency and reduced manual effort for routine tasks.



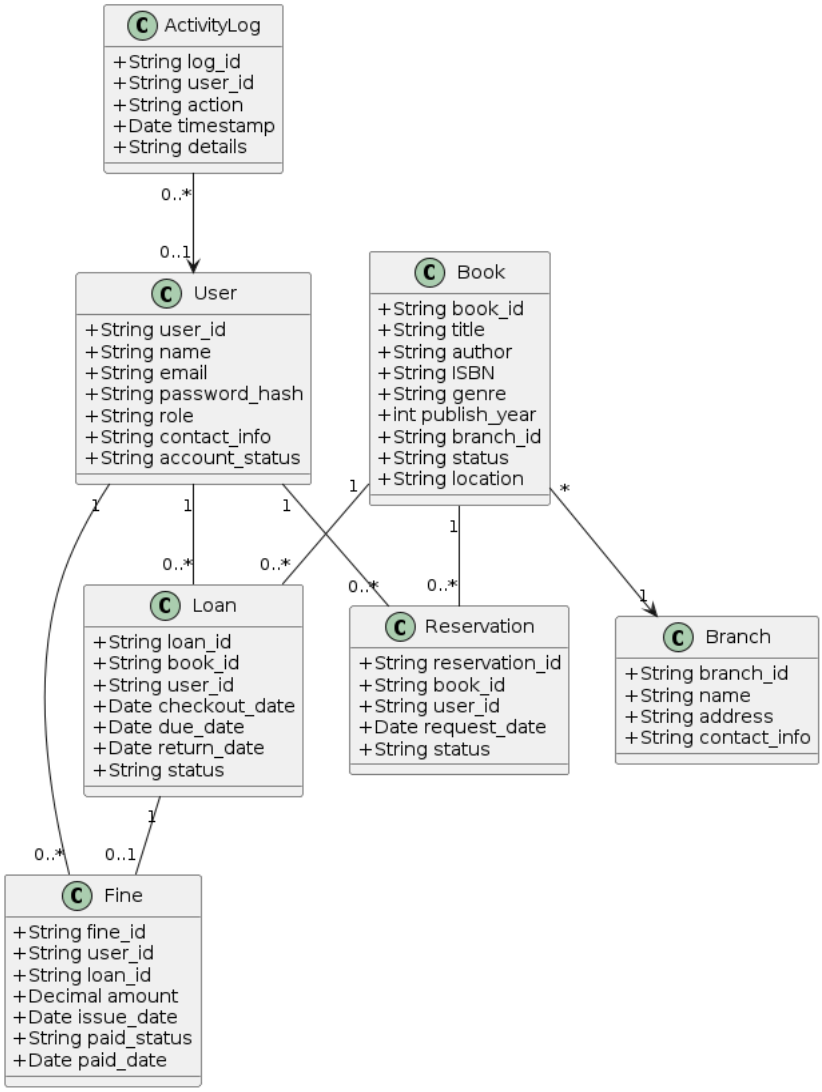
2.1.3 ADMINISTRATOR USE CASES:

Administrators will be responsible for:

- User and catalog management
- Overseeing inter-branch book transfers
- Configuring system settings
- Generating reports
- Reviewing logs

These functions support centralized administration across all library branches and provide guidance for governance, reporting, and scalability.

2.2 CLASS DIAGRAM



The **User** class holds personal and account-related information such as name, contact details, role, and authentication credentials. Each user may be associated with multiple loans, reservations, fines, and system activity logs.

The **Book** class contains metadata including title, author, ISBN, genre, and location. It is linked to a specific library branch and can participate in multiple loan and reservation records.

The **Loan** class records borrowing transactions, associating a user with a book, along with checkout, due, and return dates. It includes a status field to track the loan's progress and may optionally link to a fine if returned late.

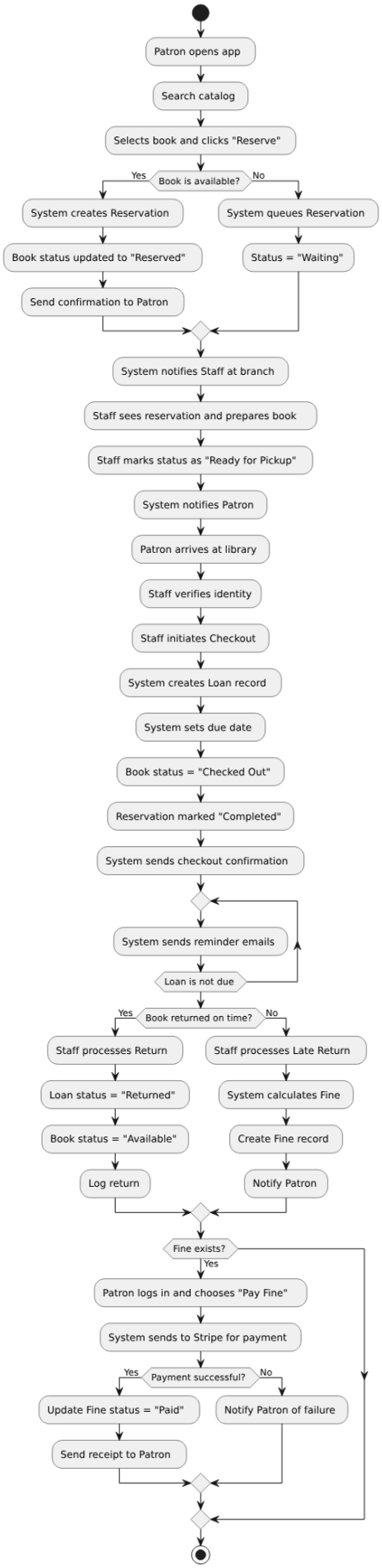
The **Reservation** class manages user requests to hold books, tracking the book and user IDs, request dates, and the current status of the hold.

The **Fine** class captures any charges issued to users, typically due to overdue returns. It stores the fine amount, issue and payment dates, and status, and references both the user and the related loan.

The **Branch** class defines library locations and includes contact information. Books are linked to branches to support multi-location inventory management.

The **ActivityLog** class logs key system actions, such as checkouts or updates. Each log entry stores a description, timestamp, and the ID of the user who performed the action, if applicable.

2.3 ACTIVITY DIAGRAM



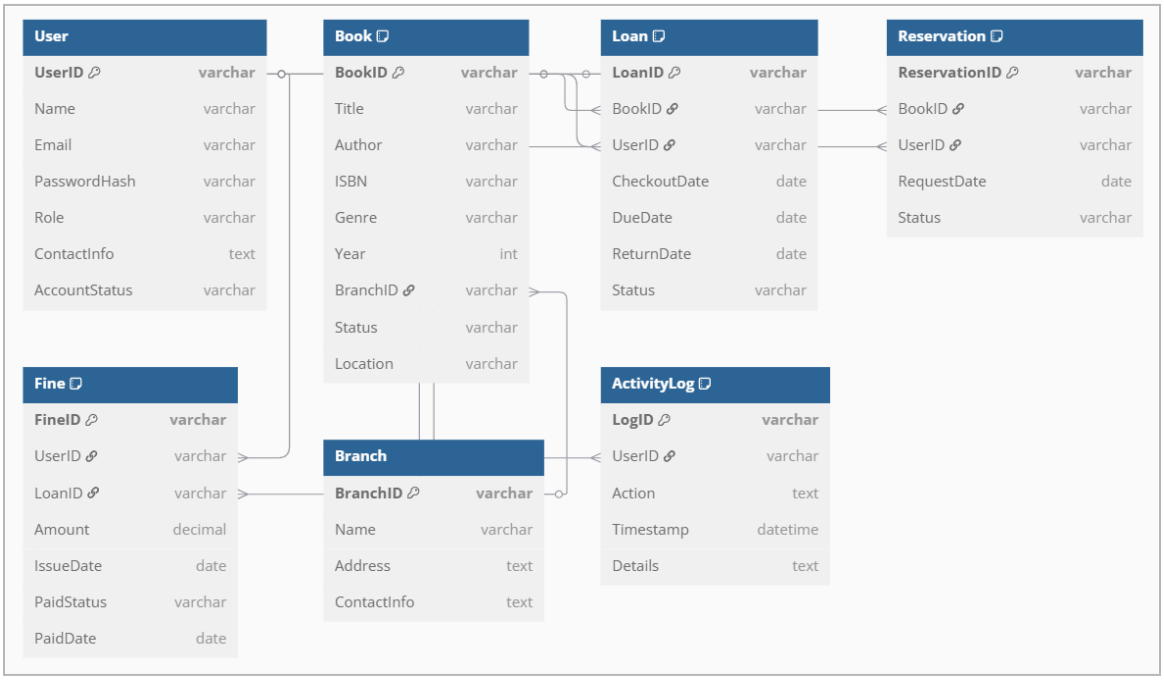
This illustrates the full workflow for reserving, borrowing, and returning a book, including optional fine payment. The following are some potential interactions between the **Patron**, **Staff**, and **System** components.

- **Patron** searches catalog and selects a book.
- **System** checks availability.
  - If available: create reservation, update status to "*Reserved*".
  - If unavailable: queue reservation, set status to "*Waiting*".
- **Staff** notified of reservation and prepare item.
- **Staff** updates status to "*Ready for pickup*"; **Patron** is notified.
- **Patron** arrives; staff verifies ID and processes checkout.
  - **Loan** record created, due date set.
  - **Book** status updated to "*Checked Out*"; reservation completed.
- **System** sends return reminders during the loan period.
- **Patron** returns book; **Staff** completes return.
  - If on time: status updated, no fine.
  - If late: fine calculated, **Fine** record created, patron notified.
- **Patron** pays fine online via Stripe.
  - **Fine** marked "*Paid*", receipt sent.
- All actions logged in **ActivityLog**; next patron (if queued) is notified.

This process supports timely reservations, automated workflows, and consistent tracking of all user and system actions.

2.4 ENTITY-RELATIONSHIP DIAGRAM (ERD)

While the backend uses MongoDB, this entity-relationship style diagram helps outline the logical structure.



Key Entities:

- **User:** Stores patrons, staff, and admins. Includes basic info, contact details, role, and account status.
    - Related to loans, reservations, fines, and activity logs.
- **Book:** Represents individual items in the catalog. Includes title, author, ISBN, location, and branch.
    - Linked to loans and reservations.
- **Loan:** Captures borrow transactions. Includes checkout and return dates, user and book references, and loan status.
    - May link to a fine if returned late.
- **Reservation:** Stores holds placed by users. Includes book, user, request date, and current status.
- **Fine:** Tracks overdue or lost item penalties. Linked to user and optionally a loan. Stores amount, issue and payment dates, and status.
- **Branch:** Represents physical library locations. Includes name, address, and contact info.
    - Each book record includes a branch ID to show current location.
- **ActivityLog:** Records system events and user actions. Stores user ID (if applicable), action taken, timestamp, and details.

3. TDD PLAN

The FMO Library System will follow a Test-Driven Development (TDD) approach to ensure functionality is verified early and consistently throughout the project. Each feature will be built using the cycle of **Red** → **Green** → **Refactor**: tests are written first, then code is implemented to pass those tests, followed by refactoring while ensuring test coverage remains intact.

3.1 TDD WORKFLOW

- **Red**: Write a failing test based on a requirement.
- **Green**: Implement minimal code to make the test pass.
- **Refactor**: Clean and improve the code while keeping all tests green.

This cycle will be applied across modules like user authentication, catalog search, reservations, checkout/return processing, and fine payment. Writing tests first helps align development with requirements, prevents regressions, and promotes modular code.

3.2 UNIT TESTING

Unit tests will target small, isolated functions and components:

- **Scope**: Core business logic such as *calculateFine()*, *reserveBook()*, and *validateLogin()*.
- **Tools**: *Jest* or *Mocha*/*Chai* for backend tests, with mocking for external services.
- **Coverage**: Focus on meaningful coverage for core features and edge cases, aiming for *80%+*.
- **Execution**: Tests will run locally and in CI pipelines to catch issues early.

3.3 INTEGRATION TESTING

Integration tests will verify that system components work together:

- **API Testing**: Simulate API calls and validate end-to-end behavior.
- **Database**: Use test or in-memory databases like *mongodb-memory-server* to verify database operations.
- **External Services**: *Stripe* and *Google Books APIs* will be mocked for repeatable testing; real sandbox tests will be used for final verification.
- **Frontend Integration**: *Cypress* or manual testing will ensure frontend and backend interact as expected.

Key flows such as *book reservation*, *login*, and *fine payment* will be tested across layers.

3.4 ACCEPTANCE TESTING

Acceptance tests will confirm the system meets stakeholder expectations:

- **Scenarios**: Derived from use cases.
- **UAT**: Library staff will test real workflows in a staging environment.
- **Automation**: Limited automation with *Cypress* or *Selenium* may be used for key flows like login, reserve, or add book.
- **Regression Testing**: Critical flows will be re-tested after major updates to ensure stability.

All test cases will be documented and tracked using simple tools like Excel.

3.5 QA PRACTICES

- **Version Control & CI**: All code will be tested automatically on push using *GitHub Actions*.
- **Code Standards**: *Linters* and peer code reviews will enforce clean, maintainable code.
- **Test Management**: Test cases and status will be tracked in a shared test matrix.
- **Tools**: *Postman* for API testing, *Jest* for unit tests, *Cypress* for E2E, and *Istanbul* for coverage reports.

Continuous testing will ensure quality is maintained throughout development. Failures will be logged and addressed before progressing to deployment.

4. DEPLOYMENT PLAN

A structured, cloud-based deployment approach ensures smooth implementation and ongoing system stability. Key components include infrastructure setup, application rollout, verification checks, and stakeholder handoff.

4.1 INFRASTRUCTURE PREPARATION

- **MongoDB Atlas**: Configure a 3-node production cluster with daily backups enabled. Network access restricted to AWS IP ranges.
- **AWS EC2 (Backend)**: Launch an Ubuntu instance with Node.js, PM2, and security group rules for HTTP/S and DB access.

- **AWS S3 & CloudFront (Frontend)**: Host static React files with global delivery and HTTPS via SSL certificates.
- **DNS & SSL**: Register subdomains and associate with AWS Certificate Manager.

4.2 APPLICATION DEPLOYMENT STEPS

Backend:

1. Clone production repository to *EC2*.
2. Install dependencies and configure .env.
3. Start application with *PM2*.
4. Verify with curl or browser endpoint.

Frontend:

1. Create new *React* application.
2. Upload files to *S3* bucket.
3. Configure *CloudFront* distribution and attach SSL.
4. Set *CORS* policy and verify API access.

4.3 PRE-LAUNCH VERIFICATION

- **Functional Tests**: Run live use case scenarios (reservation, checkout, payment).
- **Performance Checks**: Monitor page loads, test concurrent API calls.
- **Security Validation**: Confirm *HTTPS*-only access, role-based permissions, *DB* lock-down.
- **Data Integrity**: Validate test user and book data, cross-check against source records.
- **Monitoring**: Set up *CloudWatch* alarms, verify logging and alerts.
- **Backup Readiness**: Trigger manual backup and confirm success.

4.4 GO-LIVE AND HANDOFF

- **Stakeholder Review**: Final walk-through of major features with library staff.
- **Handoff**: Transfer access credentials, provide system documentation and user guides.
- **Support Plan**: Maintain active monitoring and bug response period for 1 month.

5. RECOVERY AND ROLLBACK PLAN

Ensuring system resilience through defined recovery procedures and rollback strategies minimizes risk during unexpected issues or failed updates.

5.1 RECOVERY PROCEDURES

- **Database Restoration**:
  - Use *MongoDB Atlas* point-in-time restore or latest snapshot.
  - Reconfigure application to use restored cluster if needed.
- **Backend Recovery**:
  - Launch replacement *EC2* using *AMI* backup.
  - Deploy app and redirect traffic via *DNS* or *Elastic IP*.
- **Monitoring Tools**:
  - Trigger alerts for health check failures.
  - Respond using predefined action plans.
- **Communication**:
  - Use notification template for downtime and resolution updates to staff.

5.2 ROLLBACK STRATEGY

- **Trigger Conditions**: Major bugs, app crash, data corruption, failed critical feature.
- **Steps**:
  - Revert backend via *Git tag* and *PM2* restart.
  - Restore previous frontend build to *S3*, invalidate *CloudFront* cache.
- **Database Compatibility**:
  - Avoid destructive migrations.
  - Use scripts to revert schema changes if required.
- **Testing Post-Rollback**:
  - Verify system stability and key workflows.
  - Review logs for errors.

5.3 DATA INTEGRITY MEASURES

- **Atomic Operations**: Ensure critical updates are all-or-nothing.
- **Scheduled Checks**: Validate consistency between book, loan, and user data.
- **Backup Validations**: Perform periodic test restores.
- **Manual Corrections**: Allow staff to retroactively enter data (e.g., during paper fallback).