Fatima Mahnoor, Vilka Sergei, Vu Trung

Luku Library - Book reservation System

Software Engineering Project

Product Documentation

Metropolia University of Applied Sciences
Information and communication technologies
Software Engineering
Software project
07.05.2025

1. Introduction

The University Library Platform is a modern, efficient, and accessible digital solution designed to enhance how students and faculty interact with library resources. Its core objective is to streamline the process of locating, reserving, and returning books while promoting resource availability and academic success.

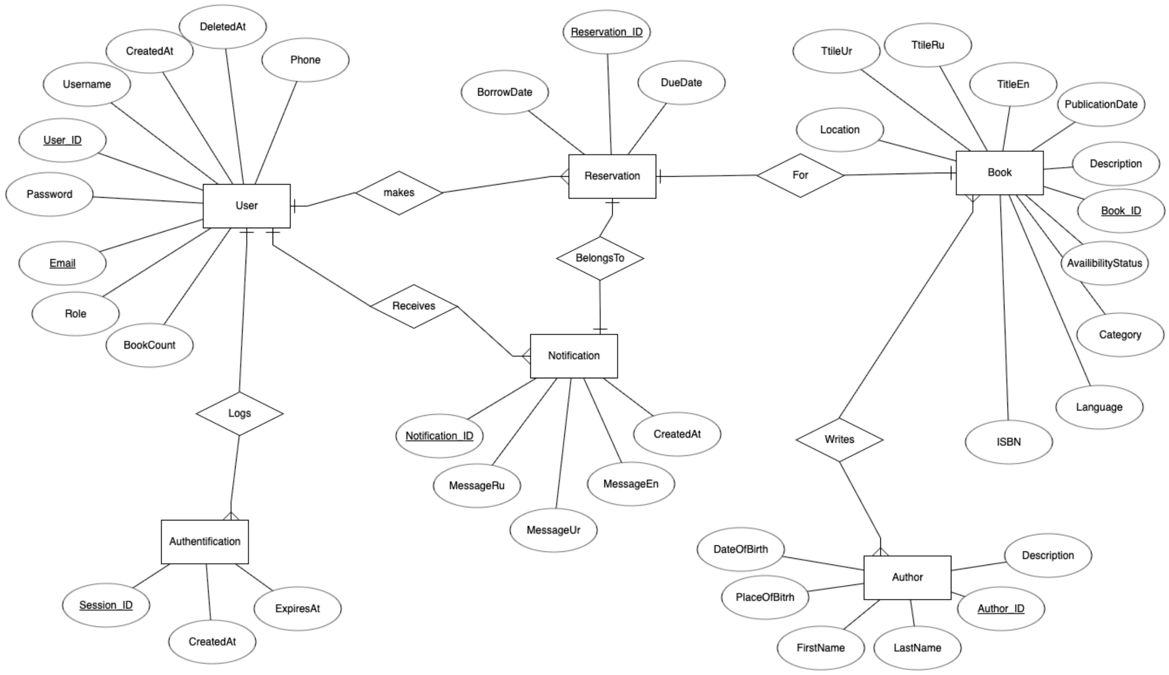
With the integration of intuitive design, real-time functionality, and multilingual support, this platform aims to bridge traditional library services with the expectations of a tech-savvy academic community. Developed using robust tools like JavaFX, MariaDB, and Docker, the platform ensures scalability, reliability, and ease of use across devices and operating systems.

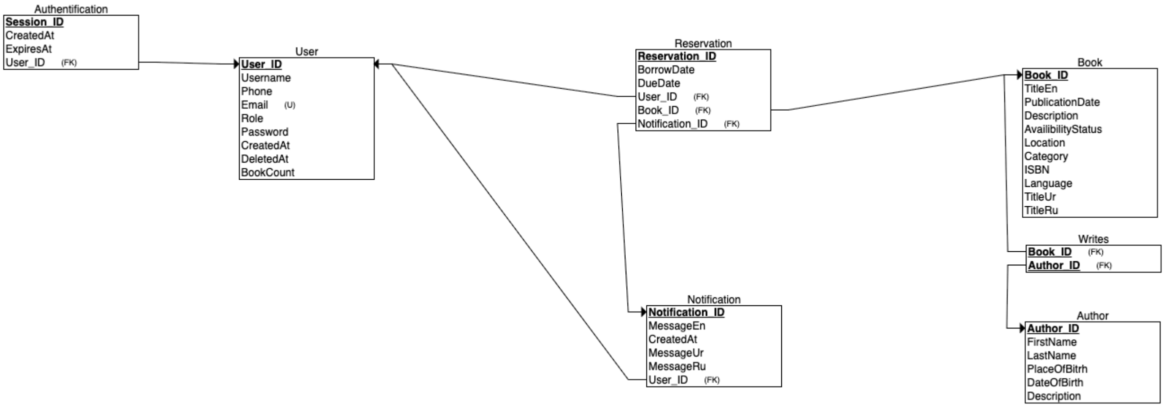
2. Design

The platform's architecture is modular and scalable, balancing simplicity in user experience with technical depth in functionality. A layered structure separates the presentation, logic, and data layers to maintain clear boundaries and enhance maintainability.

2.1 ER Diagrams

This section visualizes the database structure and entity relationships. Key entities include Users, Books, Reservations, and Notifications, each supporting multilingual data and real-time availability tracking.





Relationships and Their Descriptions

1. User makes Reservation (1-to-Many)

• A user can make multiple reservations, but each reservation is made by only one user.

2. User Receives Notification (1-to-Many)

• A user can receive multiple notifications, but each notification belongs to one user.

3. Reservation is For Book (1-to-1)

• A reservation is made for a specific book.

• One reservation can be made for one book.

4. Notification BelongsTo Reservation (1-to-1)

• A notification is directly linked to a specific reservation.

5. Author Writes Book (Many-to-Many)

• An author can write multiple books.

• One book can be written by several authors

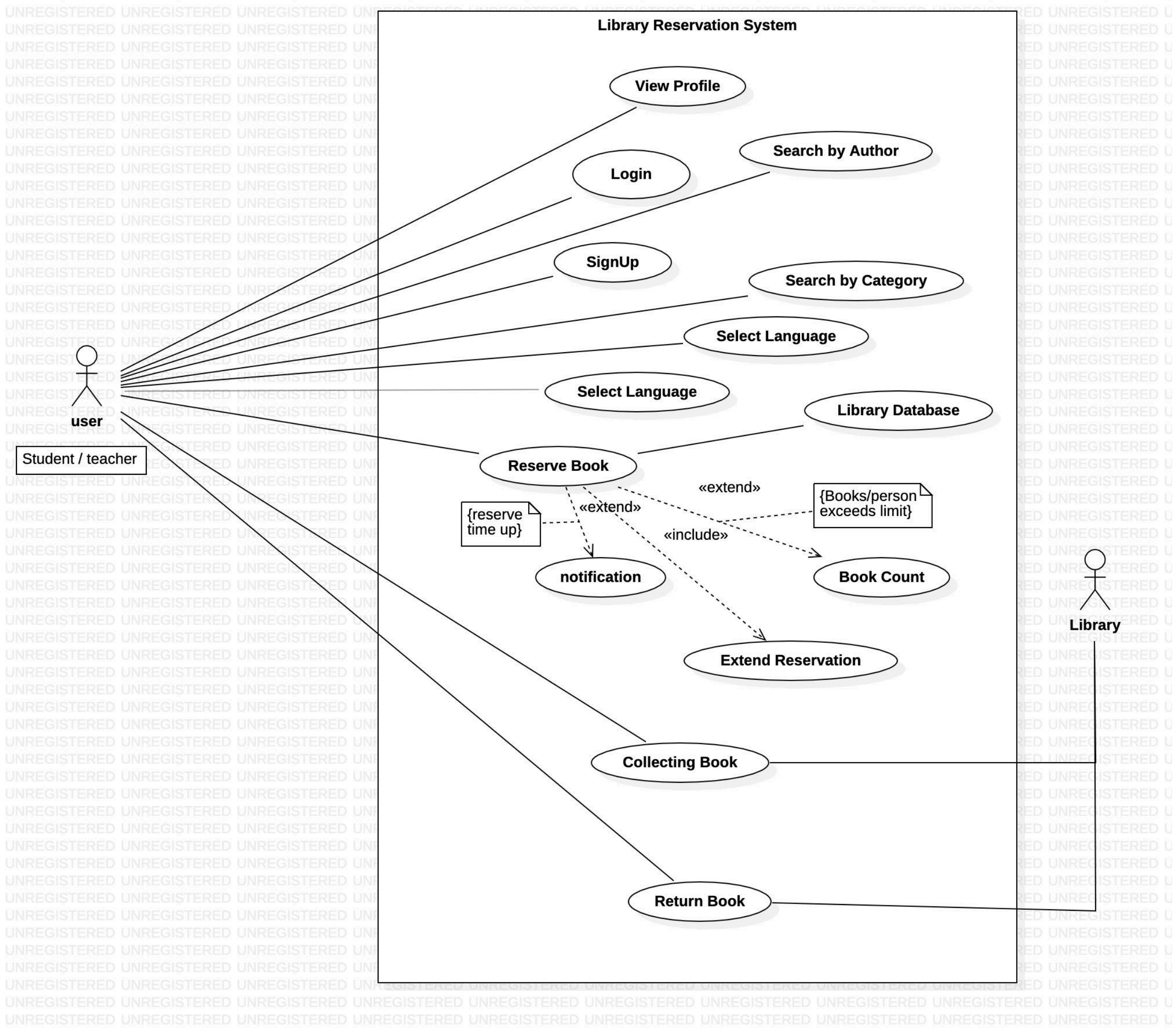
6. User Logs Authentication (1-to-Many)

• A user can have multiple authentication sessions (logins), each tracked separately.

2.2 UML Diagrams

These diagrams reflect system behaviors, interactions, and core processes like user login, book search/reservation, and notification delivery.

2.2.1 Use Case Diagram



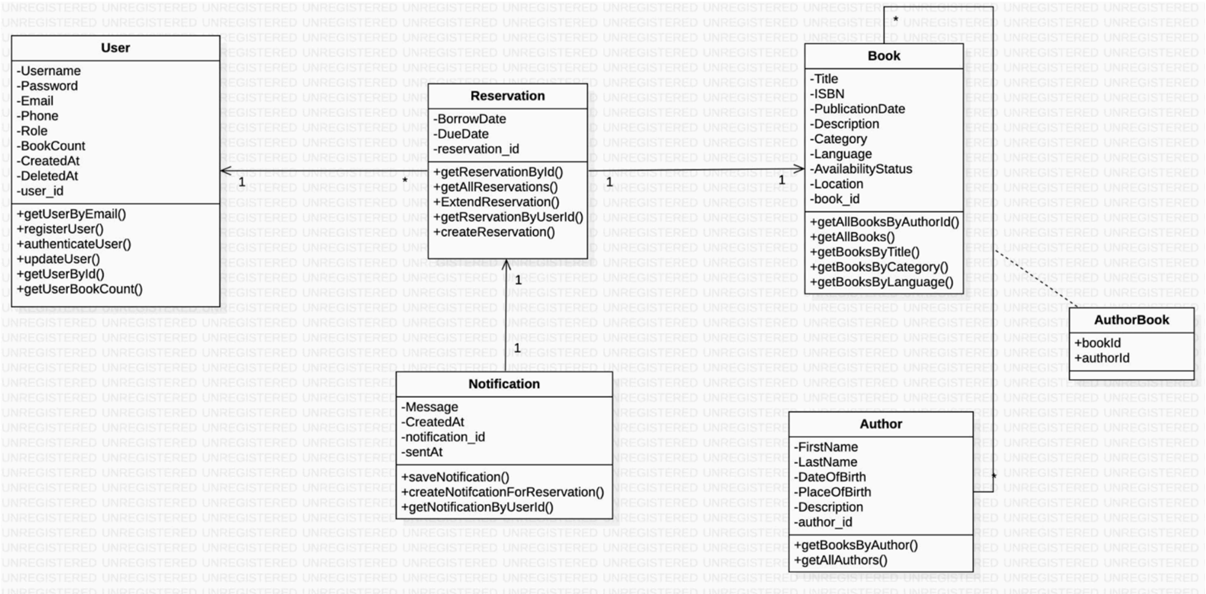
This diagram illustrates how users and the system interact:

- **Actors**:
  - *User*: Can sign up, log in, search, reserve, and return books.
  - *Library*: Physical location interacting with the system for book collection and returns.
- **Core Use Cases**:
  - *Authentication*: Sign Up and Login allow secure access.
  - *Search Operations*: Books can be searched by title, author, language, or category.
  - *Book Management*: Reserve, extend, collect, and return books.

    - *Profile Viewing*: Users can view personal info and history.

- **System Extensions and Inclusions**:
    - *Extend Reservation* is included in *Reserve Book*.
    - *Book Count* and *Notification* are extended by *Reserve Book*, ensuring constraints and reminders are handled.

2.2.2 Class Diagram

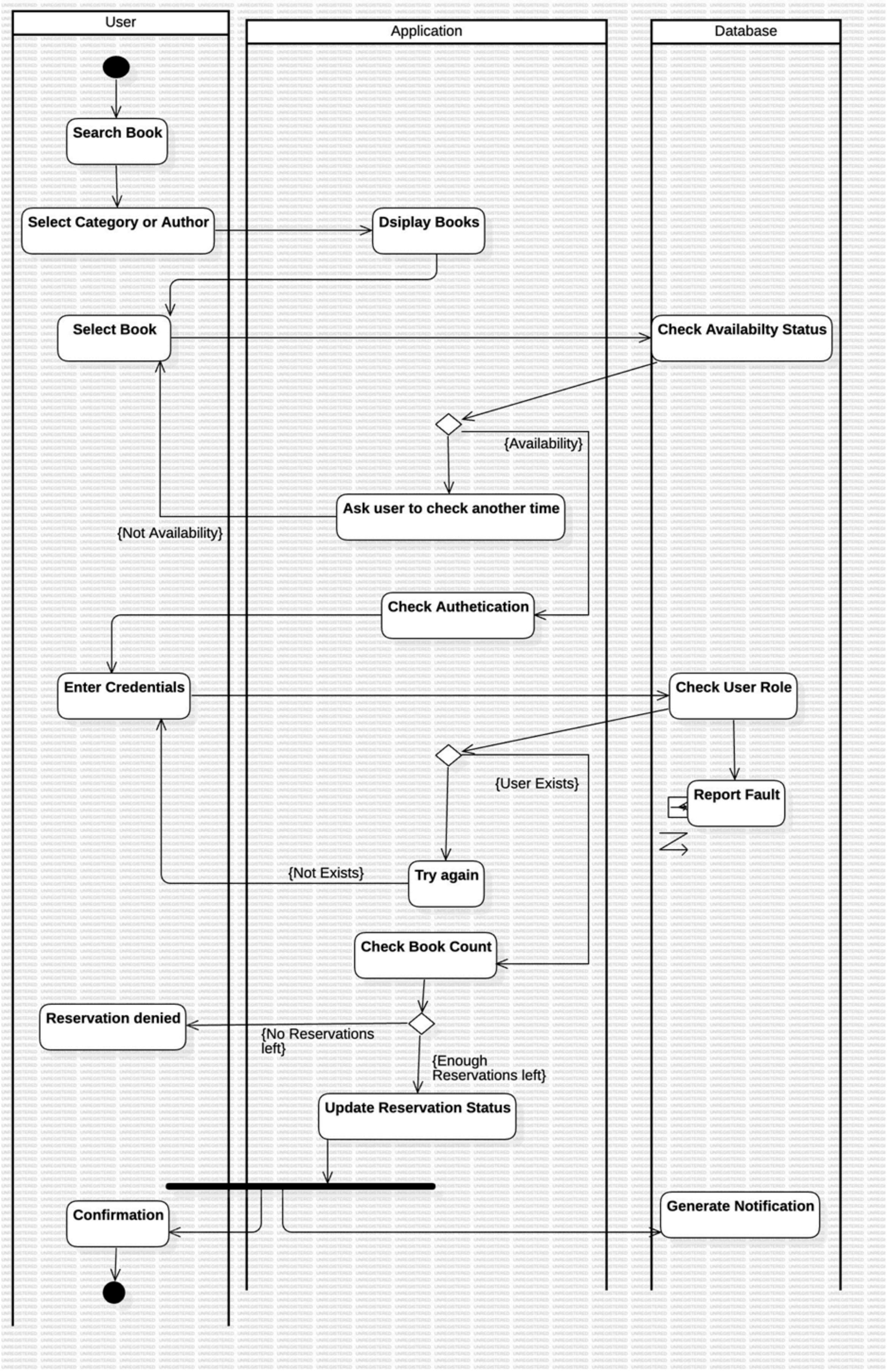


The class diagram defines the relationships between key entities:

- **User** ↔ **Reservation**: One user can make multiple reservations (1:N).
- **Reservation** ↔ **Book**: Each reservation is linked to one book (1:1).
- **Reservation** ↔ **Notification**: A notification is generated for every reservation (1:1).
- **Book** ↔ **Author**: A many-to-many relationship exists, as books can have multiple authors and vice versa.
- **Book** ↔ **Reservation**: A book can be linked to only one reservation at a time (1:1).

This structure ensures referential integrity and supports all core functionalities.

2.2.3 Activity Diagram

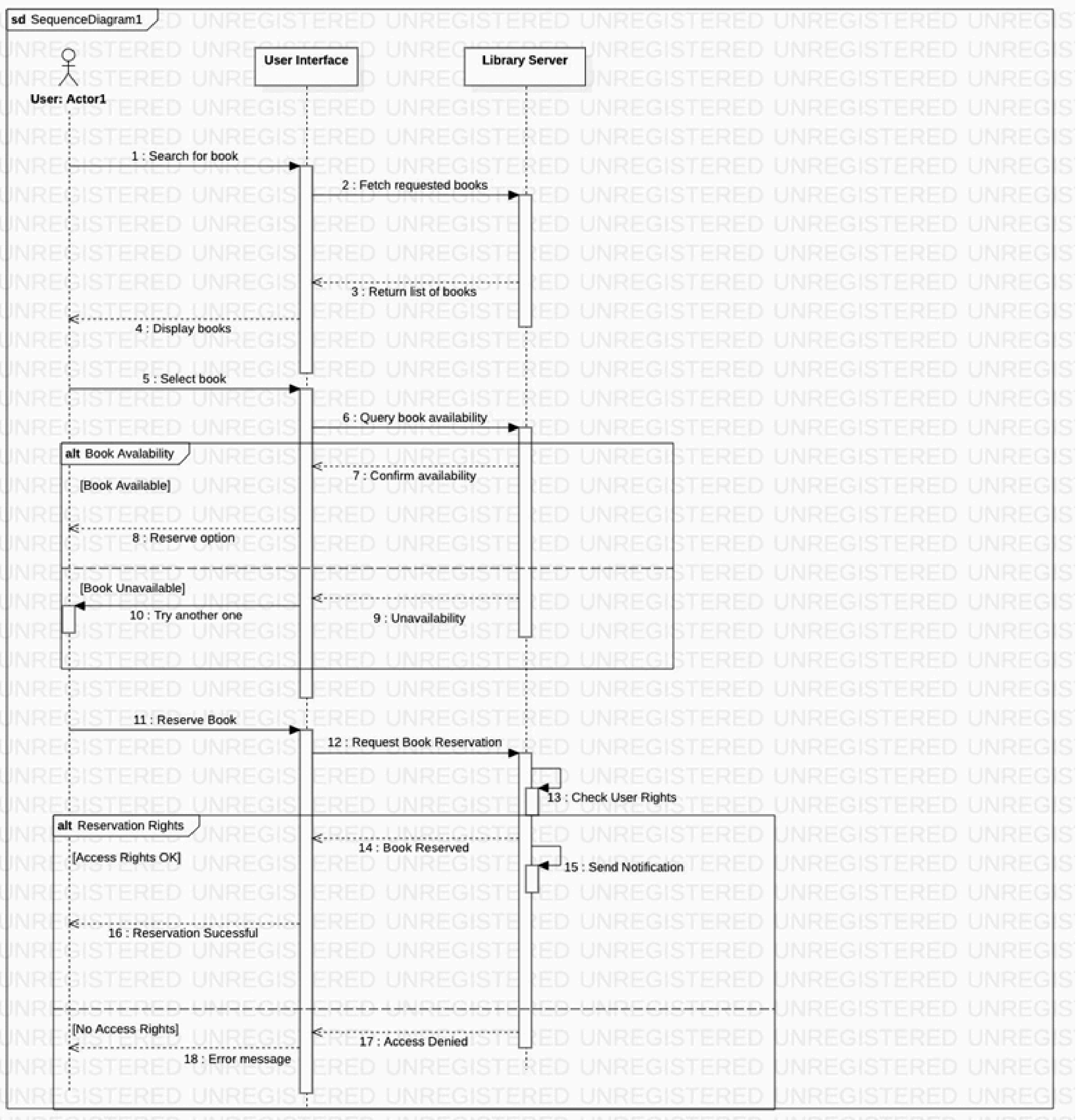

User
Application
Database
Search Book
Select Category or Author
Dsiplay Books
Select Book
Check Availabilty Status
{Availability}
Ask user to check another time
{Not Availability}
Check Authetication
Enter Credentials
Check User Role
{User Exists}
Report Fault
{Not Exists}
Try again
Check Book Count
Reservation denied
{No Reservations left}
{Enough Reservations left}
Update Reservation Status
Confirmation
Generate Notification

This activity diagram illustrates the end-to-end process of reserving a book:

1. The user searches and filters for books by category or author.
2. The system fetches and shows available books.
3. After a user selects a book, the system checks its availability.
4. If available, the user proceeds with authentication.
5. The system verifies credentials and checks if the user is within their reservation limit.
6. Upon successful validation, the reservation is recorded, a confirmation message is displayed, and a notification is generated.

This diagram emphasizes decision points like availability and authentication, ensuring a user-friendly and secure reservation flow.

2.2.4 Sequence Diagram



This sequence diagram models a typical book reservation interaction between the user and system:

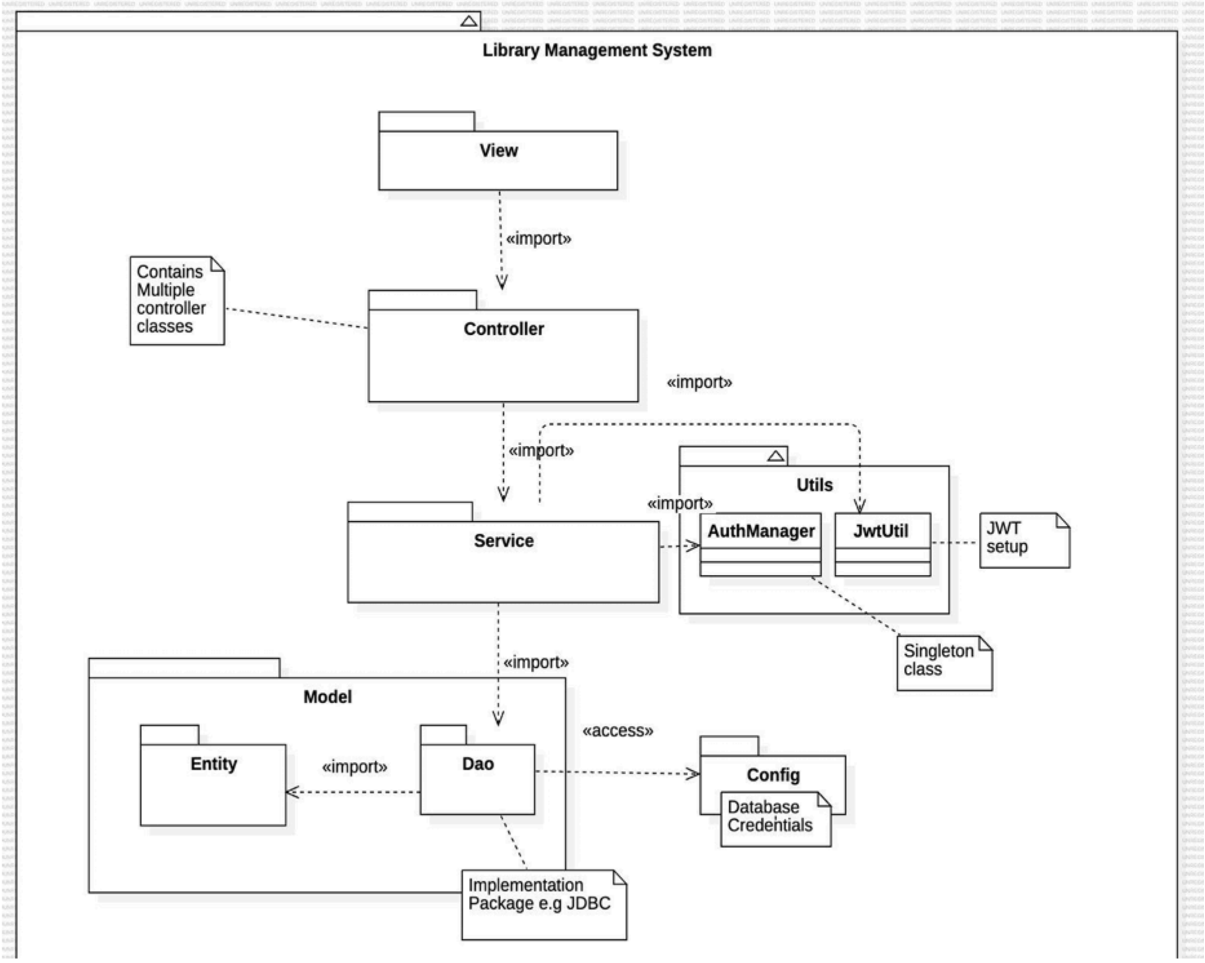
- **Actors**: A User interacts via a User Interface, which communicates with the Library Server.

- **Book Search Flow**: The user initiates a search. The UI requests matching books from the server, receives results, and displays them.

- **Book Selection and Availability Check**: The user selects a book; the system checks and confirms availability or suggests alternatives.

- **Reservation Flow**: The user attempts to reserve a book. The server checks access rights.
    - *If valid*: The reservation is confirmed, a due date notification is generated, and a success message is displayed.
    - *If denied*: The reservation is rejected with an appropriate error message.

2.2.5 Package Diagram



The package diagram outlines the modular structure of the system, promoting separation of concerns and clean architecture:

- **View Package**: Contains all UI elements, including JavaFX forms and components. It interacts directly with the Controller layer to handle user inputs.
- **Controller Package**: Manages user requests and navigation logic. Controllers invoke services and mediate between UI and backend logic.
- **Service Package**: Acts as a middle layer, processing data and business logic. It simplifies Controller responsibilities and communicates with the DAO.

- **Model Package**:
    - *Entity Sub-Package*: Represents the database schema using entity classes.
    - *DAO Sub-Package*: Manages database operations (CRUD) using JDBC or ORM methods.
- **Config Package**: Centralizes configuration files and manages database connection settings.
- **Utils Package**: Hosts reusable utility classes for authentication (e.g., `AuthManager`, `JwtUtil`) and other shared logic like token management.

3. UI Mockups

The platform's user interface is designed for clarity, ease of navigation, and responsiveness. Built using JavaFX and inspired by modern design principles, the UI ensures a seamless and intuitive experience for all users.

[Figma Link]

Key UI Features:

- **Consistent Header Across All Pages**:
  Each screen includes a fixed header with the platform logo, navigation buttons for filtering by *Category*, *Language*, and *Author*, as well as *Login* and *Search* functionalities.
- **Homepage**:
  A welcoming home screen featuring the header and a central image/banner that introduces the platform. Users can begin searching for books immediately via the prominent search bar.
- **Authentication Screens**:
  Dedicated *Login* and *Signup* pages with user-friendly forms, guiding new and returning users through quick access to the platform.
- **Category, Language, and Author Pages**:
  Separate screens allow users to browse books by:
    - Selected **Category** (e.g., Fiction, Science, etc.)
    - Chosen **Language** (English, Urdu, Russian)
    - **Author Listings**, with each author linked to their available books.
- **Logged-In Home Page**:
  Once authenticated, users see a personalized home page with their profile picture

and account options, such as bookings, profile details, and logout.

- **User Profile Page**:
  Displays personal information, including name, email, and preferred language settings.

- **User Bookings Page**:
  A list of current reservations, due dates, and status updates, all integrated with real-time availability and action buttons (e.g., return, extend).

- **Multilingual Support**:
  The interface includes a language toggle, allowing users to switch the UI and book listings between **English**, **Urdu**, and **Russian** seamlessly.

- **Notification Center**:
  Users are notified of upcoming due dates and system alerts through a dedicated notification area for better engagement and timely action.

3.1 Features Overview

The library platform is built with a set of essential and user-friendly features:

- **Book Search and Browsing**
  Quickly search by title, author, or keyword with dynamic filters.

- **Real-Time Availability & Reservation**
  Instantly view if a book is available and reserve it in one click.

- **Return Management**
  Return books with ease via a dedicated user dashboard.

- **Automated Notifications**
  Stay updated on due dates and overdue items through timely reminders.

- **Multilingual Support**
  Use the app in English, Urdu, or Russian with seamless switching.

- **User Roles & Access Control**
  Different permissions for students, teachers, and administrators.

- **Security Features**
  Secure JWT-based authentication and role-based access.

- **Accessibility-Focused UI**
  Designed to meet diverse needs, including screen reader compatibility and clear layouts.

4. Implementation Details

The **Luku Library Management System** is implemented using **Java** as the primary programming language and **Maven** as the build automation tool. The project follows a modular architecture, ensuring scalability, maintainability, and clarity in the codebase.

Below are the key implementation details:

1. *Frameworks and Libraries Used:*

- **JavaFX** for the graphical user interface (GUI).
- **MariaDB** as the relational database management system (RDBMS).
- **JUnit 5** and **Mockito** for unit and integration testing.
- **Maven Shade Plugin** for creating an executable JAR containing all dependencies.
- **Docker** and **Testcontainers** for containerization and testing.

2. *Architecture Overview*

The Luku project follows a **layered MVC architecture**, separating concerns into Model (business logic), View (UI with JavaFX), and Controller (handles user interaction).
It's a **monolithic application**, meaning all modules run in a single deployable unit, though it's structured in **modular packages** for scalability and maintainability — such as `controller`, `service`, `dao`, and `view`.

When a user interacts with view components, each component has its own controller. All the controllers have a main controller (Library controller) for code maintainability and scalability.

The controllers interact with the model service layer through the library controller, the request is then transferred from service layer to implementation layer (dao). The jakarta persistence API is used to map entities and tables into the database. The database then responds to the request accordingly.
This structure improves code readability, testing, and future extensibility. Architecture diagrams (like component or class diagrams) can further illustrate module interactions.

3. *Localization:*

- The system supports three languages: **English**, **Russian**, and **Urdu**.
- The user interface dynamically adjusts based on the selected language.

4. *Key Functionalities:*

- User Management:
    - Account registration and login
    - Profile editing
    - Role-based access control (admin, librarian, member)

- Book Reservation System:
    - Search functionality with filters (title, author, genre)
    - Book reservation and extension of due dates
    - Automatic restriction on overdues and limits

- Notifications:

    - Due date reminders and alerts via the GUI.

- Reporting & Analytics:
    - Real-time statistics on borrowed/reserved books
    - Exportable reports for administrative use

- Authentication and Security:
    - The system uses **JWT (JSON Web Token)** for secure, stateless authentication.
    - Upon successful login, the server generates a token signed with a secret key.
    - This token is included in the `Authorization` header of subsequent requests, allowing role-based access control without storing session data on the server.
    - Token expiration and refresh strategies are implemented to maintain security and usability.

5. *Database Design:*

- The system uses a MariaDB database named `library_db`.
- Tables include `users`, `books`, `reservations`, etc., with appropriate primary and foreign key relationships.
- Database initialization is automated through an SQL script mounted in the Docker container.

6. *Continuous Integration and Delivery (CI/CD):*

- Jenkins:
    - Pipelines are configured using the **Jenkinsfile** for automated builds, tests, and deployments.
    - Integrated unit, integration, and UI testing stages
- Static Code Analysis:
    - **SonarQube** for detecting code smells and security vulnerabilities
    - **Checkstyle** for enforcing code consistency and standards

7. *Containerization:*

- Dockerized architecture:

- Application and database run as isolated containers
- Ensures consistency across development, staging, and production environments

- Docker Compose:

  - Manages multi-container orchestration
  - Simplifies spin-up/down operations for the whole system

8. Code Documentation

The project uses Javadoc for inline API documentation. All public methods, classes, and interfaces are documented. You can generate the HTML documentation using:

The generated docs can be found in `/apidocs`.

It can be accessed at *http://localhost:63342/Luku/apidocs/index.html*

5. Testing Strategy and Results

The **Luku Library Management System** employs a multi-layered testing strategy to ensure the application is reliable, secure, and aligned with user expectations. Testing is carried out at multiple levels — from unit tests to user acceptance — to validate both functionality and quality attributes.

Testing Types and Approach
Functional Testing

1. Unit Testing

Definition: Verifies the behavior of individual units of code (typically methods or classes) in isolation.

JUnit

It is a popular unit testing framework in Java used to validate individual components (methods, classes) of your application. It ensures your logic works as expected by testing small, isolated parts of the codebase.

Mockito

Mockito is a mocking framework used alongside JUnit to simulate the behavior of external dependencies (e.g., databases, services) so you can test your code without needing the real implementation. It's crucial for testing services in isolation.

- Tools: **JUnit 5** and **Mockito**
- Scope: Core service logic, controller behavior, and validation logic
- Highlights:
  - High test coverage across critical components
  - Mocked dependencies for isolated and deterministic testing

2. Integration Testing

Definition: Ensures that different modules or services interact correctly, including external systems like databases.

Testcontainers

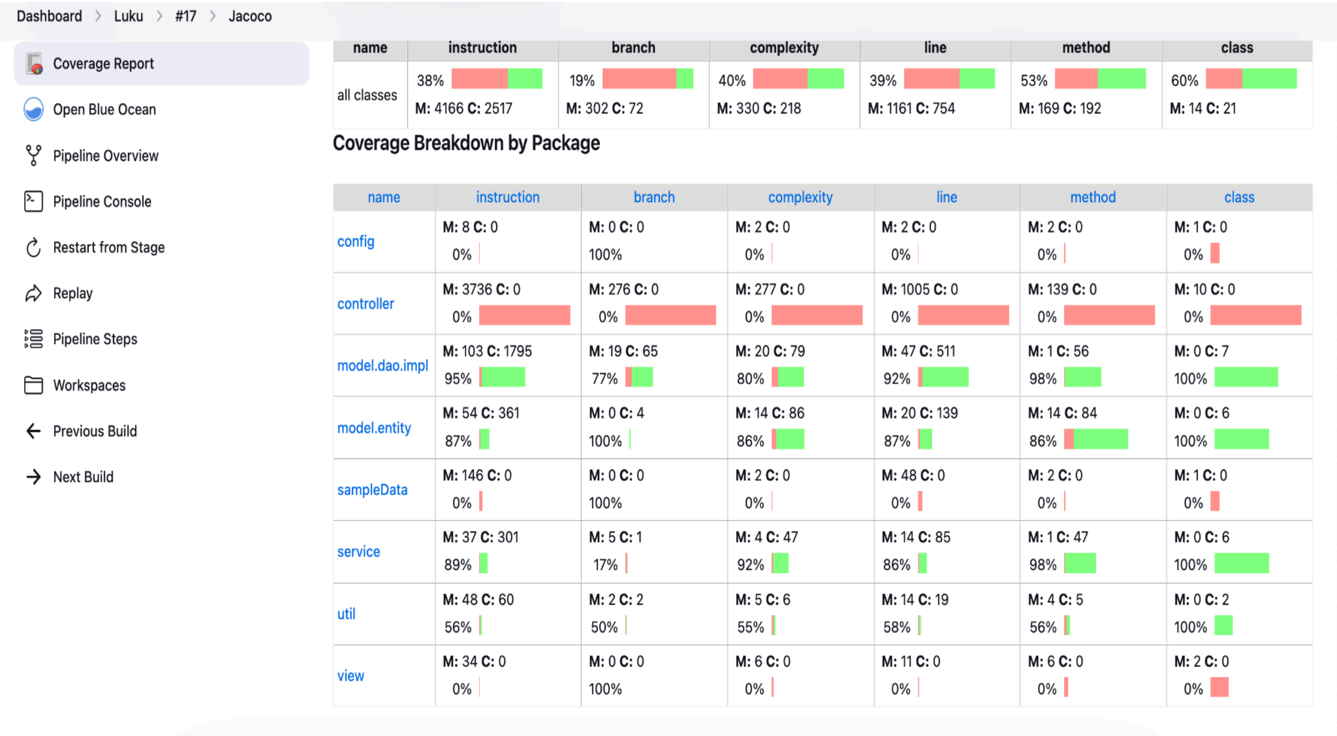
*Testcontainers is a Java library that uses Docker to spin up real instances of services like databases during tests. It's used for **integration testing**, especially when verifying how your application interacts with a real MariaDB instance—without needing to set up a database manually.*

- Tool: **Testcontainers** for real containerized MariaDB testing
- Scope: Repository methods, service-layer DB operations, and data consistency
- Verified:
  - SQL query execution and transaction integrity
  - Consistency of data between app and DB state

A total of **100** test cases using JUnit test, Mockitto and Testcontainers were conducted, giving a coverage report of 38%.

Jacoco report coverage:

JaCoCo (Java Code Coverage) in Jenkins is a plugin that generates test coverage reports for Java projects. It tracks which parts of your codebase are executed during tests, helping identify untested code. The report integrates with Jenkins pipelines, offering a visual summary of coverage metrics like line and branch coverage. It’s useful for maintaining high-quality, well-tested code.



name	instruction	branch	complexity	line	method	class
all classes	38% M: 4166 C: 2517	19% M: 302 C: 72	40% M: 330 C: 218	39% M: 1161 C: 754	53% M: 169 C: 192	60% M: 14 C: 21

Coverage Breakdown by Package

name	instruction	branch	complexity	line	method	class
config	M: 8 C: 0 0%	M: 0 C: 0 100%	M: 2 C: 0 0%	M: 2 C: 0 0%	M: 2 C: 0 0%	M: 1 C: 0 0%
controller	M: 3736 C: 0 0%	M: 276 C: 0 0%	M: 277 C: 0 0%	M: 1005 C: 0 0%	M: 139 C: 0 0%	M: 10 C: 0 0%
model.dao.impl	M: 103 C: 1795 95%	M: 19 C: 65 77%	M: 20 C: 79 80%	M: 47 C: 511 92%	M: 1 C: 56 98%	M: 0 C: 7 100%
model.entity	M: 54 C: 361 87%	M: 0 C: 4 100%	M: 14 C: 86 86%	M: 20 C: 139 87%	M: 14 C: 84 86%	M: 0 C: 6 100%
sampleData	M: 146 C: 0 0%	M: 0 C: 0 100%	M: 2 C: 0 0%	M: 48 C: 0 0%	M: 2 C: 0 0%	M: 1 C: 0 0%
service	M: 37 C: 301 89%	M: 5 C: 1 17%	M: 4 C: 47 92%	M: 14 C: 85 86%	M: 1 C: 47 98%	M: 0 C: 6 100%
util	M: 48 C: 60 56%	M: 2 C: 2 50%	M: 5 C: 6 55%	M: 14 C: 19 58%	M: 4 C: 5 56%	M: 0 C: 2 100%
view	M: 34 C: 0 0%	M: 0 C: 0 100%	M: 6 C: 0 0%	M: 11 C: 0 0%	M: 6 C: 0 0%	M: 2 C: 0 0%

Non Functional Testing

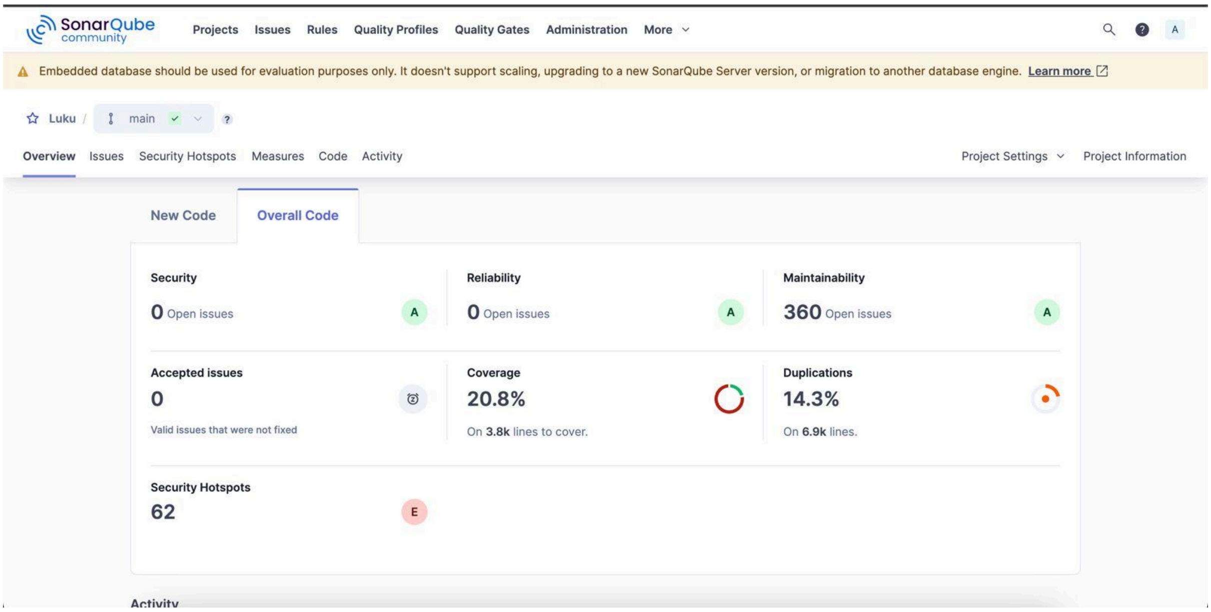
3. Static Code Analysis

Definition: Automated review of source code to detect bugs, security issues, and maintainability concerns without running the code.

Static Code Analysis (SonarQube & Checkstyle)

Analyzes code quality, style, potential bugs, and maintainability issues—without running the code. Helps enforce best practices and consistent standards.

- Tool: **SonarQube**
- Results:
    - **Security Rating**: A
    - **Reliability Rating**: A
    - **Maintainability Rating**: A



SonarQube community
Projects Issues Rules Quality Profiles Quality Gates Administration More

Embedded database should be used for evaluation purposes only. It doesn't support scaling, upgrading to a new SonarQube Server version, or migration to another database engine. Learn more

Luku / main
Overview Issues Security Hotspots Measures Code Activity
Project Settings Project Information
New Code
Overall Code

Security: 0 Open issues (A)

Reliability: 0 Open issues (A)

Maintainability: 360 Open issues (A)

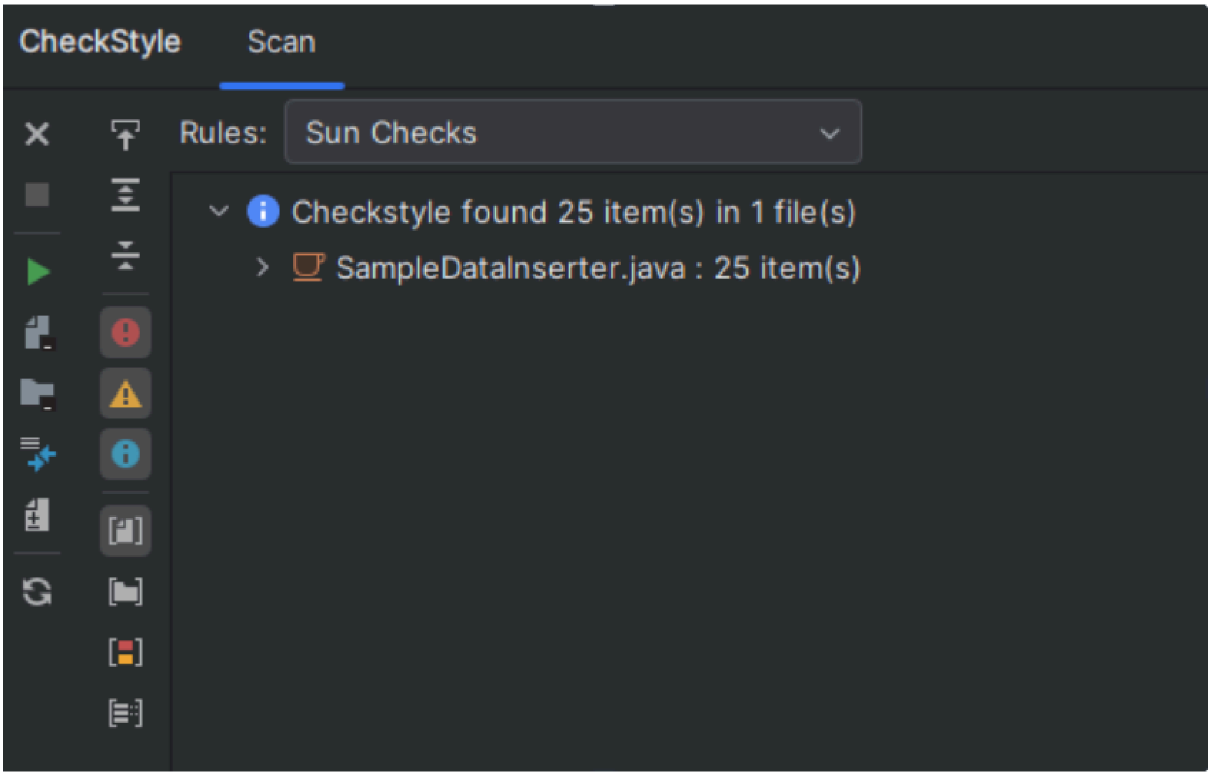
Accepted issues: 0 — Valid issues that were not fixed

Coverage: 20.8% — On 3.8k lines to cover.

Duplications: 14.3% — On 6.9k lines.

Security Hotspots: 62 (E)

- Code Quality: 1,597 of 1,622 issues resolved via **Checkstyle** compliance



Rules: Sun Checks

Checkstyle found 25 item(s) in 1 file(s)

SampleDataInserter.java : 25 item(s)

4. Non-Functional Testing

- **Heuristic Evaluation**

Definition: Expert-based usability review based on established UI/UX principles.

Heuristic Evaluation

A usability audit based on established design principles (e.g., Nielsen's heuristics). Identifies UI/UX problems like poor feedback or lack of error messages.

- Conducted in **Sprint 7** using Jakob Nielsen's 10 Usability Heuristics
- Identified UX flaws such as:
  - Vague error messaging
  - Missing cancel/exit options
- Fixes implemented and revalidated

Heuristic Evaluation Table is shown below:

No	Heuristic	Description of the Issue	Severity	Suggested Improvement
1	H1-1: Simple & natural dialog	The application uses clear and consistent language across the platform.	0	No suggestions needed.
2	H1-2: Speak the users' language	The application uses localization to make users feel at home.	0	No suggestions needed.
3	H1-3: Minimize users' memory load	The system is user-friendly. Clear book status (available/reserved), button labels, and reservation confirmation all help reduce guesswork.	0	No suggestions needed.
3	H1-3: Minimize users' memory load	After session timeout, users are left hanging — no visual indication, no message, no redirect. This forces them to figure out what's happening, increasing cognitive load.	3	Show auto-logout countdown or alert: "You've been logged out due to inactivity." Include options to "Login Again" or "Stay Logged In."
3	H1-3: Minimize users' memory load	No remember password options.	2	The app should have options like "remember me" or login with google, to help reduce stress on user memory
3	H1-3: Minimize users' memory load	The user's previous reservations that were returned, should also be visible in the "my booking" section.	2	The user should not remember all the previous bookings made.

4	4 H1-4: Consistency	The header of the application remains the same on all pages containing information about user and related reservations.	0	No suggestions needed.
4	5 H1-4: Consistency	The application employs the same primary, secondary and tertiary buttons and label styles to hold consistency.	0	No suggestions needed.
5	5 H1-5: Feedback	Users are informed about the reservations made and extended. User is also given reminder notifications about due dates	0	No suggestions needed.

Heuristic Evaluation Summary:
Strengths:

- Consistent UI with standard button and label styles.
- Localization implemented for user-friendliness.
- Visual feedback (notifications) provided for actions.

Identified Usability Issues (examples):

- Session timeout leads to stale screen without a redirect.
- “Invalid Token” message is vague — needs actionable feedback.
- Lack of "Cancel" buttons after accidental clicks.
- Navigation after using search filters is tedious.
- No keyboard shortcuts for power users.
- Missing help section or tooltips for user guidance.
- Users must re-login every time — no "Remember Me" option.

Suggested Improvements:

- Implement European date format.
- Show auto-logout alert with options to re-login.
- Add confirmation modals for important actions (reserve/return/extend).
- Improve error messaging for session expiration.
- Introduce help sections and tooltips.
- Lock book IDs to prevent double-booking.

- **User Acceptance Testing (UAT)**

  Definition: Final validation to confirm the system meets business and user needs.

  - Carried out with end-users
  - **Results**:
    - 10 test cases passed
    - 2 test cases failed (documented and addressed in issue tracker)
    - UAT success rate: **~83%**

UAT test example is below:

Test Case no.	TC-001	**Test Case Name:**	Browse Available Books
System:	Library Management System	**SubSystem:**	Book Catalogue
Designed By:		**Design Date:**	20.04.2025
Executed By:		**Execution Date:**	23.04.2025
Short Description:	Verify that users (student/teacher) can browse available books with basic search filters.		
Pre-Conditions:			
Step	**Action**	**Expected System Response**	**Pass/Comment/Fail**
1	Navigate to "Browse Books"	Book catalogue is displayed	pass
2	Enter search keyword ("History")	Filtered list of history books is shown	pass
3	Apply filter (e.g., by author)	Books by that author are displayed	pass

UAT Test Results:

Test Case	**Result**
Verify that users can browse books with filters	Passed
Ensure users can reserve books	Passed
Extend a book reservation	Passed

Notification generated when book is reserved	Passed
Reminder notification before due date	Passed
Language switching available	Passed
Guests cannot reserve books	Passed
Book is already reserved by someone else	Passed
Ensure both student and teacher roles can use features	Passed
Ensure that the student cannot reserve more than 5 books.	Passed
2 users reserving same book simultaneously	**Failed**
Session timeout does not redirect to login page	**Failed**

Summary of Results:

Test Category	Tool(s) Used	Status / Rating
Unit Testing	JUnit 5, Mockito	50 test cases, high coverage, all green
Integration Testing	Testcontainers	50 test cases, All DB interactions verified
Static Analysis	SonarQube, Checkstyle	A-grade across all categories
Heuristic Evaluation	N/A	Key UX flaws fixed
UAT	Manual	83% pass rate, remaining bugs reported in the documentation.

Additional Considerations

- **Continuous Testing** is integrated into the CI/CD pipeline using Jenkins.
- Failed test cases and known issues are tracked using an internal issue tracker (e.g. GitHub Issues, Jira).

6. Installation & Setup Guide

The **Luku Library Management System** can be set up and run both locally and via Docker. Follow the instructions below to configure the environment, build the system, and run the application.

Clone the Repository

~~~
 ```sh
          git clone https://github.com/S-Vilka/Luku.git
         cd Luku
```
~~~

I. Local Setup

1. Configure the Database

Ensure **MariaDB** is installed and running.

- Create a database named `library_db`.
- Alternatively, use Docker Compose:

~~~
```sh
         cd docker
         docker-compose -f docker-compose-db.yml up
=```
~~~

2. Set Environment Variables

Create a `.env` file in the root directory with the following content:

```
public static final String DB_URL =
"jdbc:mariadb://localhost:3306/library_db";
public static final String USER = "library_user";
public static final String PASSWORD = "library_password";
```

3. Insert Sample Data

Run the `SampleDataInserter.java` file to prepopulate the database with test entries.

4. Build the Project

```sh
        mvn clean install
```

5. Run the Application

- Using Maven:

```sh
        mvn exec:java
```

- Or run the compiled JAR:

```sh
        java -jar target/LukuLibrary.jar
```

- Or run the `Main` class directly from the `view` package in your IDE.

Running Tests

Run unit and integration tests:

```sh
        mvn test
```

Test coverage includes:

- Service and controller layers (JUnit 5, Mockito)
- Integration with MariaDB (Testcontainers)

II. Docker Deployment

1. Create Docker Env File

Create `.env.docker` with:

~~~~
public static final String DB_URL =
"jdbc:mariadb://mariadb:3306/library_db";
public static final String USER = "library_user";
public static final String PASSWORD = "library_password";
~~~~

2. Build and Run

~~~~
```sh
        docker build -t yourdockerhub/luku:v1 .
```
~~~~

To verify:

~~~~
```sh
        docker images
        docker  ps
```
~~~~

3. Push to Docker Hub

~~~~
```sh
        docker push yourdockerhub/luku:v1
```
~~~~

Running on macOS with Docker Compose

Install XQuartz for GUI:

~~~~
```sh
        brew install --cask xquartz
        Open -a XQuartz
```
~~~~

Enable X11:

- Security → Enable "Allow connections from network clients"

- Set environment:

~~~
```sh
        export DISPLAY=:0
        xhost +localhost
 ```
~~~

Run:

~~~
```sh
        docker-compose up
 ```
~~~

Running on Windows with Docker Compose

Install Chocolatey:

```
Set-ExecutionPolicy Bypass -Scope Process -Force
iex ((New-Object
System.Net.WebClient).DownloadString('https://community.chocol
atey.org/install.ps1'))
choco install xming
```

Configure Xming (via XLaunch). Then:

```
set DISPLAY=localhost:0.0
xhost +localhost
docker-compose up
```

Project Structure

```
src/main/java           # Main application code
src/test/java           # Unit and integration tests
src/main/resources      # Config files and assets
docker/                 # Dockerfiles and Compose configs
.env                    # Local environment variables
.env.docker             # Docker environment configuration
SampleDataInserter.java # DB seed utility
Jenkinsfile             # CI/CD pipeline config
```

```
Dockerfile                # Docker image configuration
sonar-project.properties# SonarQube analysis config

pom.xml                   # Maven configuration
```

Data Persistence

- MariaDB data is mounted using Docker volumes.
- Data persists even after container restart.

Volume used: `mariadb_data`

Database Initialization

An SQL script *(`db_init.sql`)* is auto-executed in Docker using:

~~~
```sh
    /docker-entrypoint-initdb.d/
```
~~~

This seeds initial data during container startup.

Backup & Restore

Backup:

~~~
```sh
    docker exec -t mariadb_container mysqldump -u root --password=root
    library_db > backup.sql
```
~~~

Restore:

~~~
```sh
    docker exec -i mariadb_container mysql -u root --password=root library_db
    < backup.sql
```
~~~

Code Quality & Static Analysis

Checkstyle (via IntelliJ IDEA)

- Plugin: **Checkstyle-IDEA**
- Configuration: **Sun Checks**
- Initial issues: 1,622
- Resolved: 1,597

Steps:

1. Install plugin via IntelliJ Marketplace.
2. Configure under *Settings* → *Tools* → *Checkstyle*.
3. Run scan via right-click menu.

SonarQube

- Tool for static analysis and technical debt
- Launch via local SonarQube server:

~~~
```sh
        # Start server
        cd sonarqube/bin/{your_os}
        ./sonar.sh start
```
~~~

Access at: http://localhost:9000

Add `sonar-project.properties:`

Properties:

```
sonar.projectKey=luku
sonar.sources=src
sonar.host.url=http://localhost:9000
sonar.login=your_token
```

Run:

```sh
        sonar-scanner
```

Current Quality Gate Results:

- Security: A
- Reliability: A
- Maintainability: A

7. Usage Instructions

Follow these steps to effectively use the **Luku Library Management System**:

Launching the Application

1. **Start the application** via `mvn exec:java` or run the compiled JAR.
2. **Log in** with your user credentials.
3. New here? Click **Register** to create an account.
4. **Choose your preferred language**: English, Russian, or Urdu.

Book Search and Reservation

- Use the **Search** feature to filter books by:
    - Title
    - Author
    - Category
- Click **Reserve** to place a hold on a book.
- Your reservation ability depends on your **user role** (e.g., student, staff, admin).

Manage Reservations

- View your list of reserved books.

- Click **Extend Due Date** if the book is eligible.
- Return books manually or wait for the return date to auto-expire.

Notifications

- Get timely alerts for:
    - Upcoming due dates
    - Overdue items

These will display on the user dashboard.

Non-Functional Features

- **View application analytics** via SonarQube.
- Use **Docker/Docker Compose** for:
    - Rapid deployment
    - Environment consistency
    - Container-based execution

8. Troubleshooting Guide

Here’s a quick hit list of common issues and how to resolve them:

Issue: App can't connect to the database

Solution:

- Ensure **MariaDB** is up and running.

Verify credentials in your `.env` or `.env.docker`:

`DB_URL, USER, PASSWORD`

- For Docker setups, confirm container status:

~~~~
```sh    docker ps
```
~~~~

Issue: Maven build fails

Solution:

- Ensure:
    - **Java 11+**
    - **Maven 3.6+**

Clean and build the project:

~~~
```sh
      mvn clean install
  ```
~~~

Issue: Localization doesn’t work

Solution:

Check if language `.properties` files exist in:

`src/main/resources/i18n/`

- Ensure the selected language matches a valid translation.

Issue: Docker containers won’t start

Solution:

- Ensure Docker is installed and daemon is running.

Force restart containers:

~~~
```sh

    docker-compose down
    docker-compose up --force-recreate
  ```
~~~

- Check logs for clues:

~~~
```sh
        docker-compose logs
 ```
~~~

Issue: Jenkins pipeline fails

Solution:

- Confirm:
  - Docker + Maven plugins are installed in Jenkins
  - `Jenkinsfile` syntax is correct
- Check Jenkins job logs under:

  `/var/lib/jenkins/jobs/YourJob/builds/lastBuild/log`

Issue: Tests not passing

Solution:

- Start MariaDB or run `docker-compose` before executing tests.

Run tests again:

~~~
```sh    mvn test
```
~~~

9. Frequently Asked Questions (FAQs)

Q1: Who is eligible to use this platform?
A: All enrolled students, faculty, and authorized university staff.

Q2: What happens if I forget to return a book?
A: You'll receive automated email or in-app reminders to return the book.

Q3: Can I switch the interface language?
A: Yes, the platform supports English, Urdu, and Russian, and you can switch at any time.

Q4: How secure is my account?
A: Your account is protected with JWT authentication and follows standard security protocols.

Q5: What do I do if a book I want is not available?
A: You can reserve it, and you'll be notified once it becomes available.

Q6: What devices can I use to access the platform?
A: The platform is accessible on desktop environments and optimized for cross-platform deployment via Docker.

10. Support and Contact Information

If you need help, want to report an issue, or have suggestions, reach out through the following channels:

- **Email**: support@luku.com
- **GitHub Repository**: [GitHub link]
- **Figma Mockups**: [Figma link]
- **Issue Reporting**: Use GitHub Issues or email us directly.
- **Support Hours**: Monday – Friday, 10:00 AM to 5:00 PM (University Time)