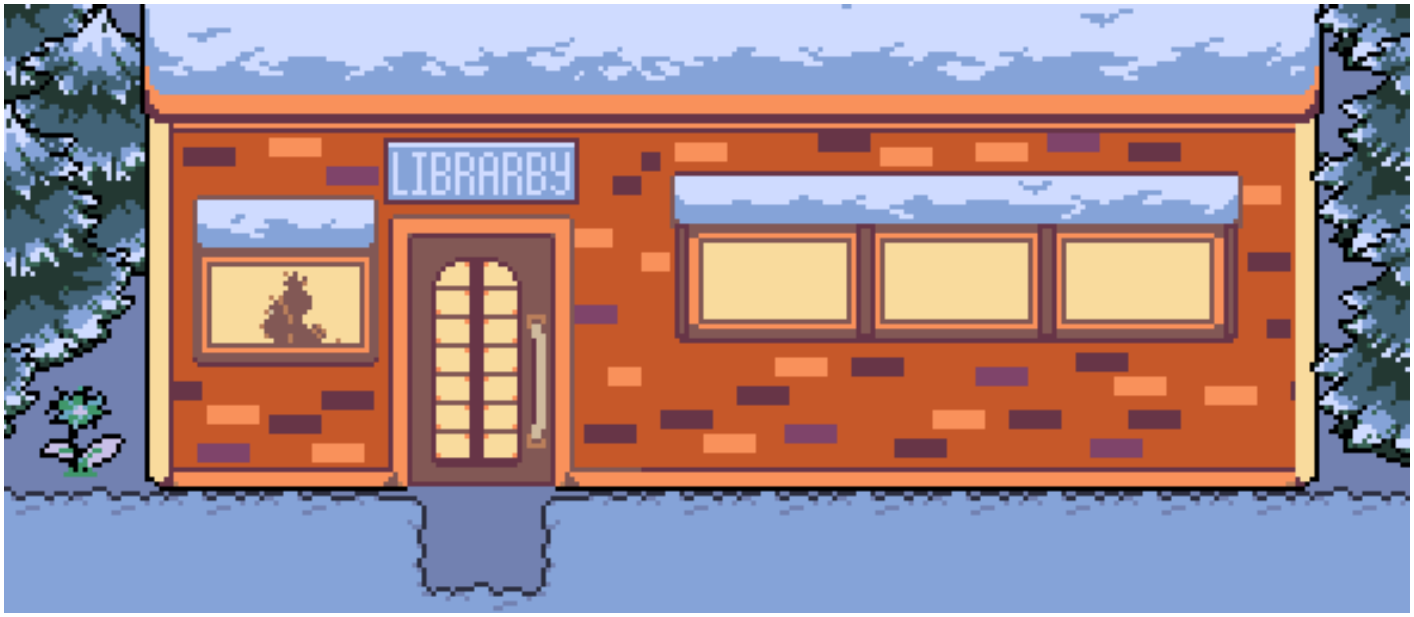
Object-Oriented Technologies: Librarby Project

Library Management Web Application Project Documentation



** Welcome to the library.*
** Yes, we know.*
** The sign is misspelled.*

Table of Contents

1. Introduction

Librarby is a RESTful backend application designed to support the core operations of a library management system. The project provides an API for managing users, books, rentals, and book reservations, offering a centralized and structured backend solution for library-related workflows.

The project was developed as part of the Object-Oriented Technologies university course, with the primary goal of applying object-oriented design principles in a real-world software engineering context. It focuses on building a well-structured backend system using modern Java technologies, emphasizing clean architecture, maintainability, and clear separation of responsibilities.

Librarby addresses the need for a unified backend that enables libraries to manage their resources and users in a consistent and secure manner. By exposing its functionality through a RESTful API, the system allows for flexible integration with potential frontend applications or external services, while remaining independent of any specific client implementation.

The system is designed around three main user roles: readers, librarians (library employees), and administrators. Each role has a distinct set of responsibilities and access rights within the system, reflecting typical organizational structures found in real-world library environments.

At its current stage, Librarby is implemented exclusively as a backend application and does not include a graphical user interface. Features such as online payments or cloud deployment are considered out of scope for this project. If time permits, a simple frontend application may be developed to demonstrate interaction with the API.

2. Project Description

Librarby is a backend system designed to support the core operational processes of a library through a RESTful API. The system is organized around several main functional areas: user management, book management, rentals, reservations, and authentication and authorization. Together, these components form a cohesive backend solution that can serve as the foundation for a complete library information system.

The application models the library domain using a set of core entities, including users, books, rentals, and reservations. These entities represent the fundamental concepts required to manage library resources and user interactions. The system is designed to be extensible, allowing additional entities and features to be introduced without significant architectural changes.

Typical system workflows reflect real-world library operations. Readers can register accounts and make online book reservations, or alternatively borrow books in person through a librarian. Librarians are responsible for managing rentals, handling reservations, and maintaining the book inventory, including adding, updating, and removing books from the system. Administrators primarily oversee employee accounts and manage system-level user roles. Future functionality is planned to allow readers to submit ratings and reviews for books they have read.

The API is designed to be publicly accessible in the sense that it can be consumed by multiple client applications, while still being intended for internal use within individual libraries. Access to resources is restricted based on user roles, ensuring that readers, librarians, and administrators can only perform actions appropriate to their responsibilities. The primary focus of the system is on supporting library staff workflows, with reader-facing functionality provided where relevant.

Throughout the project, particular emphasis is placed on clean separation of application layers, clearly defined API contracts using dedicated request and response objects, and secure handling of user data and authentication. The design prioritizes versatility and clarity, aiming to make the system both informative for educational purposes and practical for real-world usage.

At its current stage, Librarby supports user management, authentication, and the creation and management of books, rentals, and reservations. Planned extensions include the addition of book reviews and rating functionality, which will further enhance user interaction with the system.

3. Functional Requirements

This section defines the functional requirements of the Librarby system. Each requirement describes a capability that the system must provide and can be verified through API-level testing.

3.1. FR-1: User Registration

The system shall allow new users (readers) to create an account using the public API.

3.2. FR-2: User Authentication

The system shall allow registered users to authenticate using their credentials and receive a JWT token for subsequent requests.

3.3. FR-3: User Logout

The system shall provide a logout mechanism that invalidates the user session from the client perspective (e.g., by removing/invalidating the JWT token).

3.4. FR-4: Role-Based Authorization

The system shall enforce role-based access control and restrict API operations based on the authenticated user's role (reader, librarian, administrator).

3.5. FR-5: Reader Account Access

The system shall allow a reader to:

- retrieve their own account data (**GET**)
- update their own account data (**PATCH**),
  and shall prevent readers from modifying other users.

3.6. FR-6: Reader Management by Librarians

The system shall allow librarians to list, view, update, and delete reader accounts.

3.7. FR-7: Employee Management by Administrators

The system shall allow administrators to perform CRUD operations on librarian accounts (employee management).

3.8. FR-8: Book Catalog Browsing

The system shall allow all users (including unauthenticated users) to view information about books in the catalog.

3.9. FR-9: Book Inventory Management

The system shall allow librarians and administrators to manage book inventory, including:

- creating books
- creating book editions
- creating book copies
- updating book and inventory details
- removing books and/or inventory records

3.10. FR-10: Rental Creation

The system shall allow librarians and administrators to create rentals for readers based on selected book copies.

3.11. FR-11: Rental Tracking
The system shall enable rental tracking, including:

- librarians and administrators being able to view rental status and details,
- readers being able to view the status and details of their own rentals.

3.12. FR-12: Returning Books
The system shall allow librarians and administrators to process book returns and update the corresponding rental status accordingly.

3.13. FR-13: Creating Reservations
The system shall allow readers to create reservations for a selected book (or book edition), without requiring selection of a specific physical copy.

3.14. FR-14: Cancelling Reservations
The system shall allow readers to cancel their own reservations.

3.15. FR-15: Reservation Assignment to Copies
The system shall periodically check inventory availability and assign a specific book copy to an existing reservation when possible, and notify the reader about the assignment.

3.16. FR-16: Reservation-to-Rental Conversion
The system shall allow librarians to convert assigned reservations into rentals when the reader arrives to collect the reserved book.

3.17. FR-17: Error Reporting
The system shall return structured, informative API responses for failed operations, including cases such as invalid input, missing resources, and insufficient permissions.

3.18. FR-18: Requirements Extension (TODO)
Additional functional requirements (e.g., searching, filtering, pagination, ratings/reviews) are subject to extension and will be defined as the project evolves.

4. Non-Functional Requirements
This section defines the non-functional requirements of the Librarby system, describing quality attributes and constraints that influence the design and implementation of the application.

4.1. NFR-1: Security
The system shall enforce authentication for most API endpoints using a stateless JWT-based authentication mechanism. Access to protected resources shall be restricted through role-based authorization, ensuring that users can only perform actions permitted by their assigned role. User passwords shall be stored securely using appropriate hashing mechanisms to prevent exposure of sensitive credentials.

4.2. NFR-2: Performance
The system is designed to support a moderate number of concurrent users, appropriate for the scope of an academic project and a typical library environment. No strict performance guarantees are imposed; however, the application should provide responsive behavior under normal usage conditions.

4.3. NFR-3: Scalability
As a stateless RESTful backend application, the system is conceptually suitable for horizontal scaling if required in the future. While scalability is not a primary focus of the project, the architectural choices do not impose limitations that would prevent scaling.

4.4. NFR-4: Maintainability

The system shall emphasize maintainable and readable code through the use of a layered architecture and clear separation of concerns. Responsibilities are divided between controllers, services, and persistence layers, and communication between layers is performed using clearly defined Data Transfer Objects (DTOs). Code clarity and consistency are treated as important quality goals.

4.5. NFR-5: Reliability and Error Handling

The system shall handle invalid input and exceptional situations gracefully. Input data shall be validated before processing, and the API shall return consistent and structured error responses to clients, enabling clear identification of failure causes.

4.6. NFR-6: Portability

The application is intended to run in a local server environment and is platform-independent due to its reliance on the Java Virtual Machine. No cloud-specific infrastructure or deployment requirements are imposed.

4.7. NFR-7: Documentation and Developer Usability

The system shall provide comprehensive API documentation using Swagger and the OpenAPI specification. API contracts shall be clearly defined, including request and response schemas, example payloads, and response codes, to facilitate ease of use for developers and testers.

4.8. NFR-8: Project Constraints

The development of the system is subject to time constraints imposed by the university course structure, with implementation and documentation performed in iterative two-week milestones. Additionally, the project must satisfy course requirements, including the use of Spring Boot, authentication mechanisms, and core backend functionality. Development of a frontend application is considered a bonus feature.

5. System Architecture

Librarby is implemented as a single RESTful backend service, designed using a layered architecture and organized into domain-focused modules. The overall structure follows principles inspired by domain-driven design (DDD): the codebase is separated into packages representing key problem domains (e.g., authentication, users, books), while each domain contains the layers required to expose API endpoints and implement business logic. Although the current system is a monolithic service, the architecture is intended to remain modular to allow possible future migration toward microservices (for example, separating reviews/ratings or statistics into independent services).

5.1. Architectural Layers

The system uses a clear separation of responsibilities across application layers:

- **Controller layer**

  Exposes REST endpoints and handles request/response exchange with clients. Controllers validate input (where applicable) and delegate work to the service layer. They return structured response DTOs and appropriate HTTP status codes.

- **Service layer**

  Implements business logic and coordinates operations across repositories and other services. This layer is responsible for enforcing domain rules (e.g., whether an action is allowed, how a rental or reservation should be processed) and for mapping between persistence entities and DTOs.

- **Persistence layer (repositories)**

Provides database access using Hibernate and Spring Data JPA repositories. This layer handles data persistence and retrieval for the domain entities, while higher layers remain independent of low-level database concerns.

- **DTOs and mapping**

  The API contract is expressed using dedicated Data Transfer Objects (DTOs). DTOs are intentionally separated from persistence entities to keep the external API stable, prevent accidental exposure of internal fields, and allow controlled evolution of the data model. Mapping between entities and DTOs is performed in a dedicated manner (e.g., through mapper classes and/or service-layer mapping).

- **Exception handling**

  Error handling is centralized through global exception processing to ensure consistent error responses across the entire API. This improves reliability for clients and reduces duplication of error-handling logic within controllers.

- **Security and configuration**

  Authentication and authorization are implemented as a separate concern, primarily handled through the authentication module and configuration components (e.g., JWT-based security filters and related configuration). Protected endpoints rely on role-based access control enforced consistently across the application.

5.2. Domain Modules

The codebase is organized into domain-oriented packages. Not all domains are implemented fully yet; however, the structure reflects planned extensions.

- **auth**: configuration, controller, DTOs, domain model, service logic related to authentication and JWT-based security.
- **user**: controller, DTOs, mapping, domain model, repository, and service logic for managing user accounts and roles.
- **book**: controller, DTOs, domain model, repository, service logic, and utility components for book and inventory management.
- **review**: currently contains the domain model (planned extension for ratings and reviews).
- **rental**: currently contains the domain model (planned extension for full rental workflows).
- **reservation**: currently contains the domain model (planned extension for reservation workflows).
- **exception**: custom exceptions and a global exception handler (planned reorganization into a clearer structure if needed).

5.3. Request Processing Flow

A typical request is processed according to the following flow:

1. A client sends an HTTP request to a REST endpoint.
2. Security components validate authentication (JWT) and authorization (role-based access), if required for the endpoint.
3. The controller receives the request, validates input data, and delegates the operation to a service.
4. The service executes business logic and interacts with repositories to read/write data in the database.
5. Results are mapped into response DTOs and returned to the client; in case of errors, global exception handling generates a consistent error response.

5.4. Key Architectural Decisions

Two design decisions strongly influence the structure of the system:

- **Global exception handling**

  Centralized error processing ensures that failures are represented in a consistent format (e.g., via a standard error response object). This improves client usability and simplifies debugging.

- **DTO separation from persistence entities**

  Separating DTOs from database entities improves API stability and security by preventing unintended exposure of internal fields and enabling controlled evolution of the API contract independently from persistence concerns.

6. API Design

The Librarby system exposes its functionality through a RESTful API designed with clarity, consistency, and extensibility in mind. The API follows established REST conventions and uses standard HTTP methods and status codes to clearly communicate the outcome of client requests.

6.1. API Versioning and Base Path

All API endpoints are exposed under a common base path using explicit versioning:

```
/api/v1/...
```

While future API versioning is not currently required, the version identifier is included in the base path to allow backward-compatible evolution of the API if changes become necessary in later stages of development.

6.2. Resource Naming and URL Structure

The API follows consistent resource naming conventions:

- Resource names are pluralized (e.g., `/users`, `/books`).
- Individual resources are addressed using path parameters (e.g., `/users/{userId}`).
- URLs are designed to represent resources rather than actions.

At the current stage of development, nested resources are not used. This decision keeps the API structure simple and readable, with the possibility of introducing nested paths in the future if more complex relationships need to be exposed.

6.3. HTTP Methods

The API uses standard HTTP methods with clearly defined semantics:

- **GET** is used to retrieve resource representations.
- **POST** is used to create new resources, as well as for authentication-related operations such as login and registration.
- **PATCH** is used for partial updates of existing resources.
- **DELETE** is used to remove resources.

The **PUT** method is not currently used, as partial updates are sufficient for the system's requirements.

6.4. Status Code Conventions

The API consistently applies HTTP status codes to reflect request outcomes:

- **200 OK** for successful read operations or updates that return a response body.
- **201 Created** for successful resource creation.

- **204 No Content** for successful update or delete operations that do not return a response body.
- **400 Bad Request** for validation errors or malformed requests.
- **401 Unauthorized** when authentication is missing or invalid.
- **403 Forbidden** when the authenticated user lacks sufficient permissions.
- **404 Not Found** when the requested resource does not exist.

This approach ensures predictable and easily interpretable responses for API clients.

6.5. Request and Response Format

All API requests and responses use JSON as the data exchange format. Authentication-related endpoints (such as login and registration) are publicly accessible to allow users to obtain credentials, while all other endpoints require authentication unless explicitly stated otherwise (e.g., browsing available books).

Responses are expressed using dedicated response DTOs. For example, collections of resources are returned using wrapper objects (such as a response containing a list of user representations), rather than raw lists. This approach improves extensibility and allows additional metadata to be added in the future without breaking existing clients.

6.6. Error Handling Model

All error responses follow a unified structure represented by a dedicated error response object. Each error response includes information such as:

- timestamp of the error
- HTTP status code
- error type
- human-readable message
- request path

This error model is used consistently across the entire API. Maintaining a stable error response structure ensures that client applications can reliably parse and handle error conditions even as the system evolves.

6.7. Pagination, Filtering, and Search

At the current stage of development, the API does not implement pagination, filtering, or search functionality. These features are planned for future extensions, particularly for book-related endpoints, where browsing and discovery are essential.

6.8. OpenAPI and Swagger Integration

The API is fully documented using the OpenAPI specification and Swagger. Endpoints are annotated to describe request parameters, request and response schemas, example payloads, and possible response codes. Endpoints that require authentication are explicitly marked as secured using bearer token authorization, enabling accurate and interactive API documentation.

7. Security Model

Security in the Librarby system is designed to protect user data, restrict access based on user roles, and ensure that only authorized operations can be performed. The system follows a stateless authentication approach using JSON Web Tokens (JWT) combined with role-based access control enforced by Spring Security.

7.1. Authentication

The system uses JWT bearer tokens as its primary authentication mechanism. Users authenticate by submitting their credentials to a dedicated authentication endpoint. Upon successful authentication, the server issues a signed JWT that represents the user's identity and role.

7.2. Authentication flow

1. The client sends login credentials to the `/api/v1/auth/login` endpoint.
2. The server verifies the provided credentials.
3. If authentication succeeds, the server generates and returns a JWT.
4. The client includes the JWT in subsequent requests to access protected endpoints.

JWTs are configured with a fixed expiration time of 24 hours. Refresh tokens are not currently implemented but are planned for future development.

7.3. Authorization and Roles

Access to protected resources is controlled using role-based authorization. The system defines several user roles with a hierarchical permission model:

- **Unauthenticated users**

  Can register new accounts and log in. They may also access selected public endpoints, such as browsing available books.

- **Readers**

  Can view and update their own account data, browse books, create and cancel reservations, and (in future extensions) review and rate books they have rented. Readers have access only to their own data and statistics.

- **Librarians**

  Can manage reader accounts, create and manage rentals, moderate reviews, manage book inventory, and view reader statistics.

- **Administrators**

  Have full access to user management, including managing librarian accounts. Administrators can also access system-wide statistics, including those related to librarians.

Although the permission structure is primarily hierarchical, some exceptions may exist depending on specific use cases and future requirements.

7.4. Authorization Enforcement

Authorization rules are enforced using Spring Security configuration and annotations. Endpoint-level restrictions ensure that only users with the appropriate roles can access sensitive operations. This approach centralizes security logic and avoids duplicating authorization checks throughout the service layer.

7.5. Password Security

User passwords are never stored in plain text. All passwords are securely hashed using BCrypt before being persisted in the database. This ensures that credential data remains protected even in the event of unauthorized database access.

7.6. Public Endpoints

Certain endpoints are intentionally left unauthenticated to allow basic system access:

- `/api/v1/auth/login`
- `/api/v1/auth/register`
- Book browsing endpoints (including search and filtering, when implemented)

All other endpoints require authentication and appropriate authorization.

7.7. Logout and Token Revocation

Explicit logout functionality is not yet implemented. Since JWT authentication is stateless, logging out currently relies on client-side token removal. Server-side token revocation or blacklisting is planned for future versions.

7.8. Planned Security Improvements

- Several security enhancements are planned but not yet implemented:
- Refresh tokens
- Improved token lifecycle management
- Additional security mechanisms such as rate limiting or account lockout policies (under consideration)

8. Database Design

Librarby uses a relational database model implemented in MySQL, with persistence handled through Hibernate/JPA. The schema is designed to represent core library concepts such as users, books, inventory (editions and physical copies), rentals, reservations, and reviews. The database structure prioritizes normalized relations, clear ownership of data, and explicit constraints (primary keys, foreign keys, and uniqueness rules) to maintain consistency.

8.1. Core Domain Entities

The database schema can be grouped into the following logical areas:

8.1.1. Users and Roles

User-related data is modeled using a layered structure:

- `user_accounts` stores authentication-related information:
  - `email` (unique), `username` (unique), `password` (hashed), and `role` (`ADMIN`, `LIBRARIAN`, `READER`).
- `user_profiles` stores personal information:
  - `first_name`, `last_name`.
  - Uses the same primary key (`id`) as `user_accounts`, enforced via a foreign key to `user_accounts(id)`.

Role-specific data is stored in separate tables using a shared primary key approach (one-to-one inheritance-like modeling):

- `admins`, `librarians`, `readers` all reference `user_profiles(id)` as their primary key.
- The `readers` table extends the profile with additional attributes such as `date_of_birth` and `rental_limit`.

This design resembles a "joined" inheritance strategy: a base account/profile record is extended by a role-specific record when applicable.

8.1.2. Books, Authors, Genres, Publishers

Books are modeled with separation between the abstract book record and concrete editions/copies:

- `books` represents a logical book (title-level), including optional `age_rating`.
- `authors` stores author information (`first_name`, `last_name`).

- `book_authors` is a many-to-many join table between books and authors, using a composite primary key `(book_id, author_id)`.

Genres are stored as an enum association:

- `book_genres` associates a book with one (or potentially multiple) genres via `book_id` and an enum field `genre`.
    - (Note: the table does not define a primary key in the DDL, which may allow duplicates unless constrained at the ORM level.)

Publishers and editions:

- `publishers` stores publisher names.
- `book_editions` represents a particular edition of a book and references:
    - `book_id → books(id)` (required)
    - `publisher_id → publishers(id)` (optional)
    - `isbn` is unique (`UNIQUE (isbn)`), which is a key integrity constraint.

8.1.3. Physical Inventory (Copies)

Physical copies are tracked explicitly:

- `exact_book_copies` represents a physical copy of a given edition:
    - `book_edition_id → book_editions(id)`
    - `status` enum: `AVAILABLE`, `BORROWED`, `LOST`, `RESERVED`, `UNAVAILABLE`

This allows the system to manage real inventory independently of "title-level" or "edition-level" concepts.

8.1.4. Rentals and Reservations

Rentals (borrowing):

- `rentals` represents an active or historical borrowing record.
- Each rental references:
    - `reader_id → readers(id)`
    - `copy_id → exact_book_copies(id)`
- Important fields include:
    - `rented_at`, `due_date`, optional `returned_at`
    - `status` enum: `ACTIVE`, `LATE`, `ON_TIME`

This clearly ties each rental to one reader and one physical copy.

Reservations (holding):

- `reservations` represent a reader's request to reserve a book.
- Each reservation references:
    - `reader_id → readers(id)`
    - `book_id → books(id)` (reservation is currently made at the book/title level)
    - optional `assigned_exact_book_copy_id → exact_book_copies(id)`

A key constraint is:

- `UNIQUE (assigned_exact_book_copy_id)`

This ensures a physical copy cannot be assigned to multiple reservations simultaneously.

The reservation includes lifecycle tracking fields:

- `created_at`
- optional `hold_expiration_date`
- `status` enum: `PENDING`, `ASSIGNED`, `CANCELLED`, `EXPIRED`, `COMPLETED`

Design implication: Reservations are currently attached to `books(id)` rather than `book_editions(id)`. This means a reservation targets a title in general, not a specific ISBN/edition. If the intended business rule is "reserve a specific edition," the schema may later evolve to reference `book_editions` instead of (or in addition to) `books`.

8.1.5. Reviews

Book reviews are modeled as:

- reviews:
  - `reader_id → readers(id)`
  - `book_id → books(id)`
  - includes `rating`, optional `text`, and `created_at`

This allows readers to rate books independently of editions/copies (which is typically appropriate for reviews).

8.2. Integrity Constraints and Consistency

The schema includes several constraints that support data integrity:

- **Uniqueness**
  - `user_accounts.email` unique
  - `user_accounts.username` unique
  - `book_editions.isbn` unique
  - `reservations.assigned_exact_book_copy_id` unique
- **Referential integrity**
  - Foreign keys ensure rentals, reservations, and reviews cannot reference non-existent users/books/copies.
  - Shared primary key relationships enforce one-to-one consistency between accounts, profiles, and role tables.

9. Error Handling and Validation

Librarby applies a consistent approach to input validation and error handling to improve reliability and provide clear feedback to API clients. The goal is to ensure that invalid requests are detected early, failures are reported in a structured format, and client applications can handle errors predictably.

9.1. Input Validation

The system uses the `jakarta.validation` framework to validate incoming request data. Validation constraints are defined directly on request DTOs (e.g., required fields, format restrictions), and controllers apply validation by annotating request bodies with `@Valid`. This ensures that malformed or incomplete input is rejected before business logic is executed, reducing the likelihood of inconsistent or unsafe state changes.

Validation failures result in a standardized client-facing error response, typically returned with an HTTP **400 Bad Request** status code.

9.2. Global Exception Handling

Error handling is centralized using a global exception handler located in the dedicated `exception` package. By handling exceptions in one place, the system avoids duplicated error-handling logic across controllers and ensures consistent behavior for all endpoints.

Whenever applicable, exceptions are translated into a structured error payload represented by `ApiErrorResponse`. This mechanism supports common REST error scenarios such as invalid input data, missing resources, and insufficient permissions.

9.3. Standard Error Response Format

All error responses follow a unified schema and include the following fields:

- **timestamp** — time of error occurrence,
- **status** — HTTP status code,
- **error** — error type identifier (e.g., `NOT_FOUND`),
- **message** — human-readable description of the failure,
- **path** — request path associated with the error.

Using a single error format across the entire API improves consistency and makes it easier for clients to implement error parsing, logging, and user-facing messaging.

9.4. Supported Error Scenarios

The system is designed to return meaningful responses for common failure cases, including:

- **400 Bad Request** — invalid input or validation failure,
- **401 Unauthorized** — authentication missing or invalid,
- **403 Forbidden** — insufficient permissions for the requested operation,
- **404 Not Found** — referenced resource does not exist.

Additional error handling may be extended as the project evolves.

9.5. Success Responses and DTO Contracts

In addition to consistent error reporting, the system uses dedicated DTOs for successful responses in various endpoints (e.g., user data responses, authentication responses). This ensures that the API contract remains explicit, stable, and easy to document through OpenAPI/Swagger.

10. Technologies Used

The Librarby project is implemented as a modern backend application using a well-established Java ecosystem. The chosen technologies support clean architecture, security, maintainability, and clear API documentation.

10.1. Backend Framework

- **Java**

  The project is implemented in Java, providing strong object-oriented capabilities, a rich ecosystem, and mature tooling suitable for backend development.

- **Spring Boot**

  Spring Boot is used as the main application framework. It simplifies configuration, dependency management, and application startup, allowing the project to focus on business logic rather than infrastructure.

- **Spring Web** (**Spring MVC**)

Used to expose RESTful API endpoints and handle HTTP requests and responses.

10.2. Security

- **Spring Security**

  Responsible for authentication and authorization across the application. It enforces role-based access control and secures protected endpoints.

- **JWT (JSON Web Tokens)**

  Stateless authentication mechanism used to authenticate users and authorize API requests.

- **BCrypt**

  Password hashing algorithm used to securely store user credentials.

10.3. Persistence and Database

- **MySQL**

  Relational database used to persist application data, including users, books, rentals, reservations, and reviews.

- **Hibernate (JPA)**

  Object-relational mapping (ORM) framework used to map Java entities to database tables and manage database interactions.

- **Spring Data JPA**

  Simplifies data access by providing repository abstractions and reducing boilerplate code.

10.4. Containerization and Local Development

- **Docker & Docker Compose**

  Docker is used during development and testing to provide a consistent local environment. A Docker Compose configuration is included to start a MySQL database instance locally, allowing the application to be run and tested without requiring a manually installed database server.

  This approach simplifies onboarding, ensures environment consistency across development machines, and reduces setup time.

10.5. API Documentation

- **OpenAPI (Swagger)**

  The API is documented using the OpenAPI specification. Endpoints, request/response schemas, authentication requirements, and example payloads are described using annotations.

- **Swagger UI**

  Provides an interactive web interface for exploring and testing API endpoints.

10.6. Validation and Error Handling

- **Jakarta Validation (Bean Validation)**

  Used to validate incoming request data using declarative annotations.

- **Global Exception Handling** (`@ControllerAdvice`)

  Centralized error handling mechanism ensuring consistent error responses across the API.

10.7. Build and Tooling

- **Gradle**

  Build automation tool used for dependency management, building, and running the application, as well as auxiliary tasks such as generating API documentation.

10.8. Development and Documentation Tools

- **IntelliJ IDEA**

  Primary development environment.

- **Typst**

  Used to prepare formal project documentation for university submission.

- **Git**

  Version control system used for collaboration and source code management.

11. How to Run the Project

This section describes how to set up and run the Librarby backend application in a local development environment.

11.1. Prerequisites

Before running the project, ensure the following tools are installed:

- **Java 25**
- **Docker** and **Docker Compose** or own instance of **MySQL server**
- **Git** (to clone the repository)

The project includes the **Gradle Wrapper** (`gradlew`), so a local Gradle installation is not required.

11.2. Database Setup (MySQL)

For local development and testing, the project uses a MySQL database started via Docker Compose.

1. Navigate to the Docker configuration directory:

   ```
   cd docker
   ```

2. Start the database container:

   ```
   docker compose up -d
   ```

This command starts a MySQL instance exposed on port **3306**. The database schema is managed automatically by Hibernate based on the application configuration. If you do not wish to use Docker, you can connect to your own database - the config is located in `application.properties` file.

11.3. Application Configuration

The application is configured using `application.properties`, which contains database connection settings and other required configuration values for the test environment.

Additionally, a `.env` file is used to provide environment variables for initial setup, such as creating an administrator account.

(A `.env-example` template is planned for future revisions.)

No additional configuration is required to start the application in its default setup.

11.4. Running the Backend Application

The backend can be started in one of the following ways:

Option 1: Using Gradle (recommended)

From the project root directory, run:

```
./gradlew bootRun
```

Option 2: Using an IDE

The project can also be run directly from an IDE such as **IntelliJ IDEA** by executing the main Spring Boot application class.

By default, the backend application starts on port **8080**.

11.5. Accessing API Documentation (Swagger)

Once the application is running, the interactive API documentation is available at:

http://localhost:8080/swagger-ui/index.html

Most endpoints require authentication. To test secured endpoints via Swagger UI, a JWT token must be obtained using the login endpoint and provided through the Swagger authorization interface.

(Offline API documentation generation is currently a work in progress.)

12. API Documentation (Swagger)

Librarby provides interactive API documentation using **OpenAPI** and **Swagger UI**, generated automatically from the Spring Boot application at runtime. The documentation is intended to support development and testing by clearly describing available endpoints, request/response schemas, authentication requirements, and expected status codes.

12.1. Tooling and Integration

The project uses the **springdoc-openapi** integration for Spring WebMVC, specifically:

- `org.springdoc:springdoc-openapi-starter-webmvc-ui:2.8.15`
- `org.springdoc:springdoc-openapi-starter-webmvc-api:2.8.15`

OpenAPI metadata is produced based on controller mappings and Swagger/OpenAPI annotations placed throughout the codebase. The documentation is designed to remain consistent with the actual API implementation, reducing the risk of outdated external documentation.

12.2. Accessing Swagger UI

When the backend application is running (default port **8080**), Swagger UI is available at:

http://localhost:8080/swagger-ui/index.html

Swagger UI provides a browser-based interface for exploring endpoints, inspecting request and response schemas, and performing test requests directly from the documentation view.

12.3. OpenAPI Specification Endpoint

The generated OpenAPI specification is available in JSON format at:

http://localhost:8080/v3/api-docs

This endpoint can be used by external tools (e.g., API clients, code generators, or documentation generators) and also serves as the source for Swagger UI.

12.4. Authentication in Swagger UI

Most endpoints in Librarby are protected and require authentication. To test secured endpoints in Swagger UI:

1. Obtain a JWT token using the authentication endpoint (login).
2. Provide the token in Swagger UI using the authorization mechanism (bearer token).
3. Execute requests to protected endpoints as needed.

This workflow allows end-to-end testing of authorization rules and role-based permissions directly through the documentation interface.

12.5. Documentation Scope and Quality

Swagger/OpenAPI documentation is enhanced using annotations to provide:

- endpoint summaries and descriptions,
- request/response schemas,
- error response models (e.g., structured error payloads),
- example payloads where applicable,
- response codes for common success and failure scenarios.

13. Limitations and Future Work

While Librarby implements the core functionality required for a library management backend, the current version reflects both the academic scope of the project and time constraints imposed by milestone-based development. Several limitations have been identified, along with clear directions for future improvements and extensions.

13.1. Current Limitations

- **Limited feature completeness**

  Some modules, such as reviews, statistics, and advanced reservation workflows, are partially implemented or represented only at the data model level.

- **No frontend application**

  The system currently exposes a RESTful API only. Interaction with the system is performed via API clients or Swagger UI rather than a dedicated user interface.

- **No token refresh or logout mechanism**

  Authentication relies on stateless JWT tokens with a fixed expiration time. Token refresh and server-side logout are not yet implemented.

- **No pagination, filtering, or search**

  API endpoints return full result sets. Pagination, filtering, and search—particularly for book listings—are planned but not yet available.

- **Limited validation and business rule enforcement**

  While input validation and basic constraints are enforced, more advanced domain rules (e.g., rental limits, reservation prioritization) may require further refinement.

- **Single-service architecture**

  The application is implemented as a single backend service. Although modularized by domain, it is not yet decomposed into separate microservices.

13.2. Planned Improvements and Extensions

- **Book reviews and ratings**

  Readers will be able to rate and review books they have rented. Moderation tools for librarians and administrators are planned.

- **Statistics and reporting**

  The system will provide statistics for readers, librarians, and administrators, including rental history, usage metrics, and inventory insights.

- **Pagination, filtering, and search**

  Book browsing endpoints will be extended to support efficient searching, filtering by genre or author, and paginated responses.

- **Enhanced authentication and security**

  Planned improvements include refresh tokens, improved token lifecycle management, and explicit logout support.

- **Frontend application**

  A simple frontend interface may be developed to provide a user-friendly way to interact with the API, depending on available time and project priorities.

- **Architectural evolution**

  Selected features (e.g., reviews, ratings, or statistics) may be extracted into separate services in the future to explore a microservice-oriented architecture.

13.3. Project Scope Considerations

The scope and implementation choices of Librarby are influenced by:

- milestone-based development cycles,
- limited development time,
- course requirements emphasizing backend architecture, security, and API design.

As a result, design decisions prioritize clarity, correctness, and extensibility over exhaustive feature completeness.

14. Conclusion

Librarby is a RESTful backend system designed to support the core operations of a library, including user management, book inventory handling, rentals, reservations, and authentication. The project was developed as part of an academic course on object-oriented technologies, with a strong emphasis on clean architecture, modular design, and well-defined API contracts.

Throughout the implementation, attention was given to separation of concerns, consistent data modeling, and secure access control. The use of modern frameworks and tools such as Spring Boot, Spring Security, Hibernate, and OpenAPI enabled the development of a maintainable and extensible backend system that reflects real-world design practices.

While the current version of Librarby focuses on essential backend functionality, the system was intentionally designed with future extensions in mind. Planned enhancements such as reviews and ratings, statistics and reporting, pagination and search, and improved authentication mechanisms can be integrated without significant architectural changes.

Overall, Librarby serves both as a functional library management API and as a learning-oriented project demonstrating practical application of object-oriented design principles, RESTful API development, and backend security concepts.