Scala project

Juillet 2025

Albane Coiffe

Maelwenn Labidurie

Constance Walusiak

Louise Lavergne

Amira Boudaoud

Summary

Introduction **3**
Link of the demo video of our application **3**
Link to github repository of the project : **4**
System Architecture **4**
Overview of Project Structure 4
Backend – Functional Core in Scala 3 5
Data Flow and Interaction 5
Web Interface Overview (/public) **6**
Structure and Components 6
index.html 6
app.js 7
How the Frontend Works **7**
Key Features 9
Domain Model (Scala) **9**
Book.scala 9
User.scala 10
Transaction.scala 11
Business Logic & Services **11**
LibraryCatalog.scala 12
Structure 12
Functional Operations 12
Search 12
Book Loan 12
Book Return 13
Recommendations 13
Synchronization 13
LibraryCatalogCodec.scala – JSON Serialization 13
Time Handling 13
Custom Decoding for User 13
Custom Decoding for Transaction 14
Full Catalog Encoder/Decoder 14
Testing Strategy **14**
Test Structure 14
Unit Tests (LibraryCatalogTest.scala) 14

Book Loan Success 14
Recommendation Logic 15
Property-Based Tests (LibraryCatalogPropertyTest.scala) 15
LoanBook Outcome 15
JSON I/O Tests (JsonIOTest.scala) 15
Save and Reload Consistency 15
Error Handling 16
Running the Console Application **16**
sbt "runMain LibraryApp" **16**
Additional Features & Possible Improvements **17**
Conclusion **18**

Introduction

This project was developed as part of the *Functional Programming with Scala* course and consists in building a fully functional **Library Management System**, combining the power of Scala 3's functional programming features with a clean and interactive web interface.

The goal of this system is to simulate the real-life operations of a library: managing a catalog of books, allowing users to search for and borrow books, tracking loans and returns, and offering personalized recommendations based on user history. Users are identified by a unique ID and can interact with the system through an intuitive interface that provides access to book search, availability status, and suggested titles.

From a technical perspective, the backend logic is written in **Scala 3**, following a pure functional programming approach. It makes use of:

- **Algebraic Data Types** (e.g., sealed traits for users),
- **Opaque and union types** for domain modeling (ISBNs, User IDs),
- **Functional error handling** with Either, Try, and Option,
- And **ScalaTest/ScalaCheck** for unit and property-based testing.

Library data is persisted in a structured JSON file (Library.json), and the system ensures data integrity through validation mechanisms. The web front-end, built separately, connects to the core logic to provide a seamless user experience, showing key stats such as total books, current users, and ongoing transactions.

This report details the architecture, key implementation decisions and development process behind the system.

Link of the demo video of our application

demo video link

Remark : examples of system usage are shown on the demo video as well on the link above

Link to github repository of the project :

Project github repository

System Architecture

The Library Management System is composed of two main layers: a functional backend written in Scala 3, and a web-based frontend implemented in HTML/CSS/JavaScript. The architecture follows a modular, type-safe, and layered design, enabling maintainability, extensibility, and testability.

Overview of Project Structure

```
project-root/
├── public/          # Web interface (HTML + JS)
│   ├── index.html   # UI layout
│   └── app.js       # Frontend logic
├── src/
│   ├── main/scala/
│   │   ├── models/   # Domain models (Book, User, Transaction)
│   │   ├── services/ # Business logic (LibraryCatalog, Codec)
│   │   └── utils/    # JSON I/O
│   └── test/scala/   # Unit + property-based tests
├── Data/            # Persistent JSON file (Library.json)
├── build.sbt        # Build configuration
└── README.md        # Project documentation
```

Backend – Functional Core in Scala 3

The backend uses modern functional programming paradigms and Scala 3 features such as:

- Algebraic Data Types (sealed trait, case class) for modeling,
- Immutable state (catalog updates produce new instances),
- Safe error handling using Either and Option,
- Functional I/O using Circe for JSON encoding/decoding.

Domain Layer (models/)

- Book, User, and Transaction are modeled as immutable case classes and sealed traits.
- Custom type aliases (ISBN, UserID) enhance domain expressiveness.
- Transactions include timestamps and user associations.

Service Layer (services/)

- LibraryCatalog: central class for operations such as book search, borrowing, returning, and recommendations.
- All methods are pure and return updated versions of the catalog or validation errors.
- A synchronizeBookAvailability function ensures consistency with transaction history.

Persistence Layer

- Implemented with LibraryCatalogCodec.scala using Circe's Encoder/Decoder typeclasses.
- Supports custom JSON handling for User, Transaction, and LocalDateTime.
- File I/O is encapsulated and tested to ensure data integrity with Library.json.

Data Flow and Interaction

- **Input/Output**: All library data is persisted in a single Library.json file, which stores:

    - the complete book catalog (books),
    - user data (users),
    - transaction history (transactions, including Loan and Return types).
- **Interaction Loop**:
    - On application start, the system loads Library.json into immutable Scala data structures.
    - Users interact with the frontend (search, borrow, get recommendations).
    - The Scala backend processes requests, updates in-memory structures, and writes changes back to Library.json.

Web Interface Overview (/public)

The public folder of the project contains the **entire web interface** of the Library Management System. It includes two key files:

- index.html – the static HTML layout and styling of the user interface,
- app.js – the dynamic behavior and frontend logic written in JavaScript.

Together, they create a responsive, single-page application (SPA) that allows users to interact with the library system in real time.

Structure and Components

index.html

This file defines the **static skeleton** of the interface, composed of the following sections:

- **Header**: Displays the title and tagline of the application.

- **Statistics Area** (#stats): Populated dynamically to show real-time counts of books, users, and transactions.

- **Sidebar**:
    - **User Identification**: Input for user ID.
    - **Search Interface**: Input field to search books by title.
    - **Action Buttons**:
        - Show available books,

        - Show personal recommendations,
        - Show all books.
- **Main Content**:
    - Book list display area,
    - Section title and dynamic updates via JavaScript,
    - Support for alerts and loading spinners.

It includes embedded CSS styles for:

- Aesthetic design (gradients, shadows, responsive layout),
- Highlighting book availability (green/red tags),
- Clean mobile adaptability (using media queries).

app.js

This script handles **all user interactions** and **data communication** with the backend API (/api/...), including:

- **Fetching data**:
    - /api/books, /api/users, /api/transactions for stats.
    - /api/books/available to show borrowable books.
    - /api/books/search for keyword searches.
    - /api/users/:id/recommendations for book suggestions.
- **User actions**:
    - Borrowing a book: /api/books/loan
    - Returning a book: /api/books/return
- **Real-time DOM updates**:
    - Displaying books as cards (createBookCard()),
    - Updating section titles and counts,
    - Showing visual alerts and loaders.
- **User feedback**:
    - Success and error alerts using showAlert(),
    - Inline loading animations during asynchronous calls.

How the Frontend Works

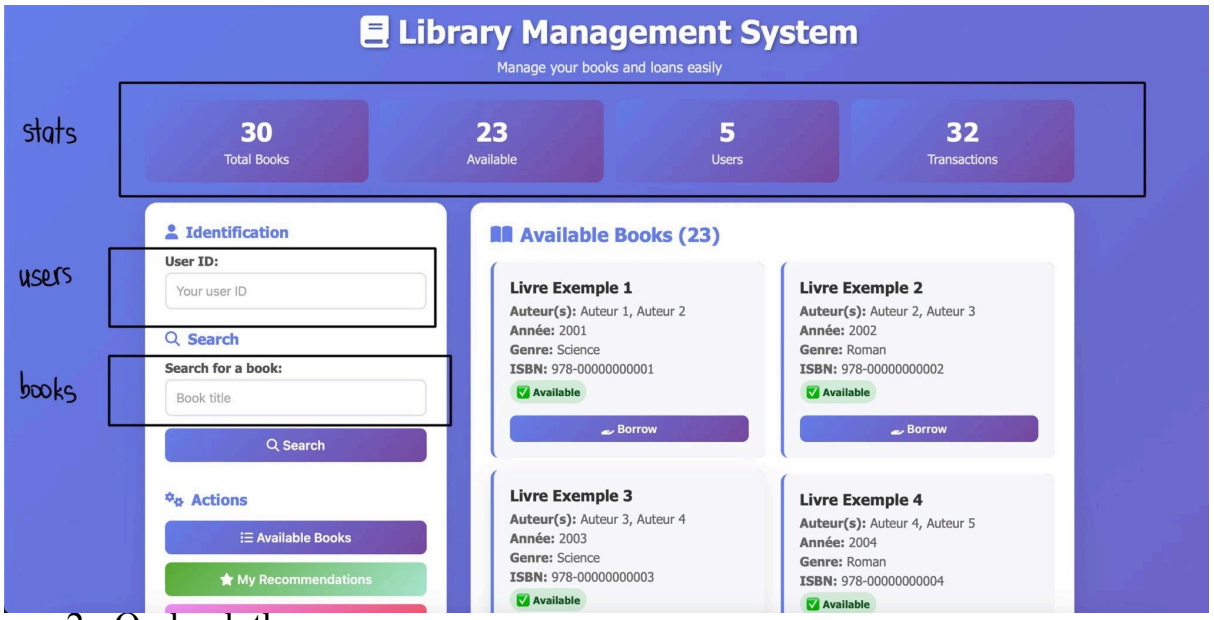
```
## Running the Web Server

sbt run
```

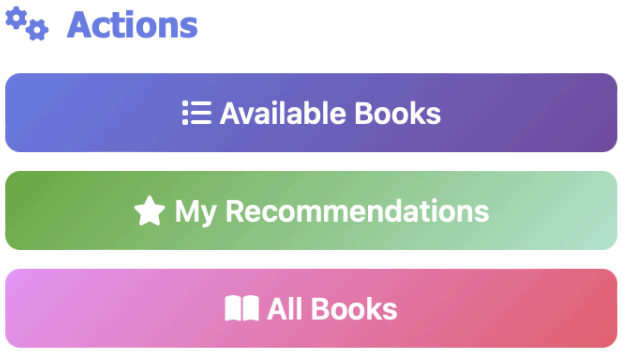
```
Access at http://localhost:8080
```

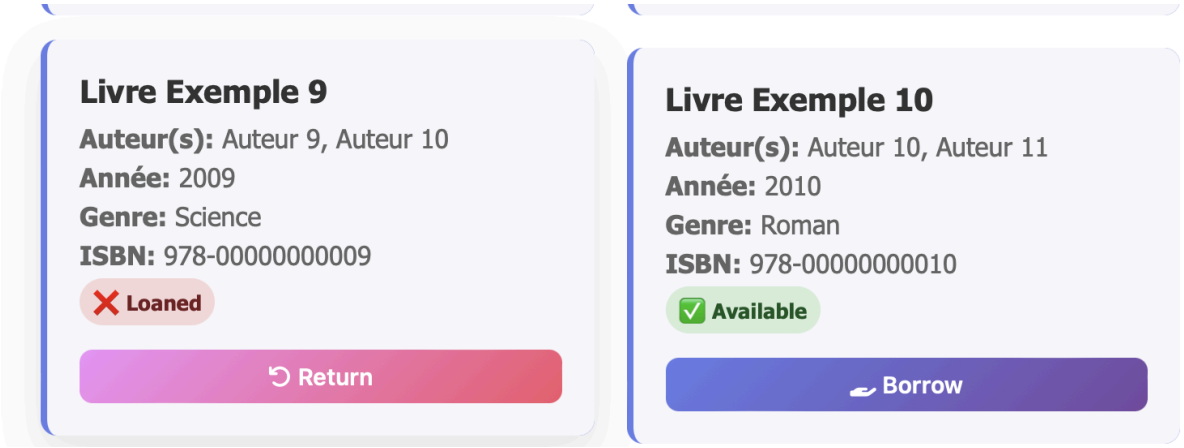
Here's a typical user interaction flow:

1. The user opens the page, which loads app.js.



2. On load, the app:
    - Fetches stats, books, and users from the backend.
    - Displays initial book list (e.g., available books).
3. The user enters their ID and:
    - Searches a book → real-time filtering from /books/search,
    - Views recommendations → query to /users/{id}/recommendations,
    - Borrows a book → POST to /books/loan,
    - Returns a book → POST to /books/return.





4. Any data change triggers automatic UI updates (book list and stats).

Key Features

- **Decoupled from backend logic**: All interactions go through RESTful APIs.
- **Fully dynamic**: No page reloads are needed thanks to fetch + DOM manipulation.
- **Mobile-friendly design**: Supports different screen sizes.
- **Clear user feedback**: Through alerts, spinners, and contextual messages.
- **Multi-user support**: Several users can interact with the system simultaneously.
- **Real-time statistics and recommendations**: The UI updates automatically the number of transactions after each action.

Domain Model (Scala)

The **domain model** defines the core data structures used in the application. It is implemented in the models/ package using **algebraic data types (ADTs)** and **case classes**, which are idiomatic in functional programming with Scala.

This modeling provides a **type-safe** and **immutable** foundation for all operations related to users, books, and transactions.

Book.scala

```
case class Book(
```

```
  isbn: ISBN,
  title: String,
  authors: List[String],
  publicationYear: Int,
  genre: String,
  available: Boolean
)
type ISBN = String
```

The Book class represents the metadata and availability of a library book. It includes:

- A unique identifier (isbn),
- Title and authors (as a list),
- Year of publication and genre,
- An available flag to track borrowing status.

A type alias is used for ISBN to improve domain clarity and support potential use of **opaque types** later in the project.

User.scala

The User model is defined as a **sealed trait hierarchy**, representing the different types of users in the system.

```
sealed trait User {
  def id: UserID
  def name: String
}
```

Subtypes include:

- Student – with an additional field level (e.g., Undergraduate, Graduate)

- Faculty – with a department
- Librarian – with a position

A **type alias** is used for user IDs:

```
type UserID = String
```

This structure allows for:

- **Pattern matching** on user types,
- **Strong typing** with shared interface methods (id, name),
- Easy extension in future (e.g., adding Guest or Alumnus).

Transaction.scala

The Transaction trait models any user-book interaction, timestamped with a LocalDateTime.

```
sealed trait Transaction {

  def book: Book

  def user: User

  def timestamp: LocalDateTime

}
```

Concrete implementations include:

- Loan: book borrowing,
- Return: book return,
- Reservation: reservation for future borrowing.

By using a **sealed trait**, all possible transaction types are known at compile time, enabling exhaustive pattern matching and functional error handling.

Business Logic & Services

The services/ package encapsulates the **core logic** of the Library Management System. It defines how books are borrowed or returned, how users are validated,

how recommendations are generated, and how the system synchronizes the availability of books.

LibraryCatalog.scala

This is the main service class that acts as an **immutable data container** and provides **pure functional operations** on the state of the library.

Structure

```
case class LibraryCatalog(
  books: List[Book],
  users: List[User],
  transactions: List[Transaction]
)
```

Functional Operations

All methods return either new instances of LibraryCatalog (immutability) or Either[String, LibraryCatalog] to handle errors functionally.

Search

- findByTitle(title: String)
- findByAuthor(author: String)
- availableBooks
  → All use a reusable findBooks(predicate) helper.

Book Loan

```
def loanBook(isbn: ISBN, userId: UserID): Either[String, LibraryCatalog]
```

- Checks if the book is available and user exists.
- Marks the book as unavailable.
- Adds a Loan transaction.

Book Return

```
def returnBook(isbn: ISBN, userId: UserID): Either[String, LibraryCatalog]
```

- Verifies the book is borrowed.
- Marks the book as available.
- Adds a Return transaction.

Recommendations

```
def recommendBooks(userId: UserID): List[Book]
```

- Based on the user's borrowing history (Loan transactions).
- Suggests available books from preferred genres, sorted by frequency.

Synchronization

```
def synchronizeBookAvailability: LibraryCatalog
```

- Ensures availability flags in books match transaction history.
- Useful in case of manual or inconsistent data updates.

LibraryCatalogCodec.scala – JSON Serialization

This file defines all the **Circe encoders and decoders** needed to persist and reload the catalog as JSON (used with Library.json).

Time Handling

```
given Encoder[LocalDateTime]

given Decoder[LocalDateTime]
```

→ Converts timestamps to ISO string format.

Custom Decoding for User

```
given Decoder[User] = ...
```

→ Maps any simple user object to a default Student if specific fields are missing. This enables backward compatibility with minimal JSON data.

Custom Decoding for Transaction

given Decoder[Transaction] = ...

→ Simplifies JSON loading by treating all transactions as Loan by default (can be improved later to match by type if needed).

Full Catalog Encoder/Decoder

given Encoder[LibraryCatalog]

given Decoder[LibraryCatalog]

→ Enables full reading/writing of the library state to/from Library.json.

Testing Strategy

Testing in this project is handled using the **ScalaTest** and **ScalaCheck** frameworks. All critical functionalities of the library system are tested through a combination of:

- **Unit tests** (with AnyFunSuite) for precise behavior checks,
- **Property-based tests** (with AnyPropSpec + ScalaCheck) for generative verification,
- **Persistence tests** for JSON serialization and deserialization.

Test Structure

Located under src/test/scala/, tests are organized by concern:

- services/LibraryCatalogTest.scala → unit tests for business logic
- services/LibraryCatalogPropertyTest.scala → property-based tests
- utils/JsonIOTest.scala → file I/O and JSON encoding/decoding

Unit Tests (LibraryCatalogTest.scala)

These tests verify the correctness of key use cases in the LibraryCatalog class.

Book Loan Success

test("Loan book should update availability and transactions")

→ Confirms that a borrowed book is marked as unavailable and a Loan is added.

Loan Failure (Unavailable Book)

test("Loan should fail if book not available")

→ Ensures proper handling of invalid loan attempts.

Recommendation Logic

test("Recommendation should return available books in preferred genre")

→ Tests that recommendations filter on genre and availability.

Property-Based Tests (LibraryCatalogPropertyTest.scala)

These tests use **random data generators** to ensure general behaviors hold across a wide range of inputs.

LoanBook Outcome

property("loanBook should either succeed or return error")

→ Verifies that loanBook always returns a valid Either, avoiding exceptions or nulls.

Generators are defined for:

- Valid Book instances,
- Multiple User roles (Student, Faculty, Librarian),
- Random values for title, author, availability, etc.

JSON I/O Tests (JsonIOTest.scala)

These tests ensure **data integrity** during persistence.

Save and Reload Consistency

test("Save and load JSON should be consistent")

→ Saves a catalog to a file, reloads it, and compares content.

Error Handling

test("Load should fail with non-existent file")

→ Verifies graceful failure for missing files.

Cleanup

test("Cleanup temporary file")

→ Ensures the test environment remains clean after execution.

Running the Console Application

```
sbt "runMain LibraryApp"
```

```
──────────────────────────────
1. Search for a book
2. Borrow a book
3. Return a book
4. Show my recommendations
5. List available books
6. Exit
──────────────────────────────

Your choice: 5
📗 Livre Exemple 1 – 978-00000000001
📗 Livre Exemple 2 – 978-00000000002
📗 Livre Exemple 3 – 978-00000000003
📗 Livre Exemple 4 – 978-00000000004
📗 Livre Exemple 5 – 978-00000000005
📗 Livre Exemple 10 – 978-00000000010
📗 Livre Exemple 13 – 978-00000000013
📗 Livre Exemple 14 – 978-00000000014
📗 Livre Exemple 15 – 978-00000000015
📗 Livre Exemple 17 – 978-00000000017
📗 Livre Exemple 18 – 978-00000000018
📗 Livre Exemple 19 – 978-00000000019
📗 Livre Exemple 20 – 978-00000000020
📗 Livre Exemple 21 – 978-00000000021
📗 Livre Exemple 22 – 978-00000000022
📗 Livre Exemple 23 – 978-00000000023
📗 Livre Exemple 24 – 978-00000000024
📗 Livre Exemple 25 – 978-00000000025
📗 Livre Exemple 26 – 978-00000000026
📗 Livre Exemple 27 – 978-00000000027
📗 Livre Exemple 28 – 978-00000000028
📗 Livre Exemple 29 – 978-00000000029
📗 Livre Exemple 30 – 978-00000000030
```

```
──────────────────────────────
1. Search for a book
2. Borrow a book
3. Return a book
4. Show my recommendations
5. List available books
6. Exit
──────────────────────────────

Your choice: 1
Title to search: Livre Exemple 10
📖 Livre Exemple 10 – Auteur 10, Auteur 11 (2010)

──────────────────────────────
1. Search for a book
2. Borrow a book
3. Return a book
4. Show my recommendations
5. List available books
6. Exit
──────────────────────────────

Your choice: 2
Your user ID: 1
ID of the book to borrow: 978-00000000001
✅ Book borrowed successfully.

──────────────────────────────
1. Search for a book
2. Borrow a book
3. Return a book
4. Show my recommendations
5. List available books
6. Exit
──────────────────────────────

Your choice: 3
Your user ID: 1
ID of the book to return: 978-00000000001
✅ Book returned successfully.
```

Here is an example of errors that occur when the user does not write in the correct format / the number does not exist:

Additional Features & Possible Improvements

- **Error Handling**: Improved user feedback for invalid actions (e.g., invalid user ID, unavailable book).
- **Extensibility**: The modular design allows for easy addition of new features (e.g., reservations, reviews).
- **Security**: Future versions could add authentication and authorization for users.
- **Database Integration**: For larger deployments, migrating from JSON to a database would improve scalability.
- **Performance**: Asynchronous operations and optimized data structures could further enhance responsiveness.
- **User-friendly interface**: the application that we made helps users to use our application.

Challenges & Lessons Learned

- **Functional State Management**: Ensuring immutability and pure functions required careful design, especially for catalog updates.
- **Frontend-Backend Integration**: Designing a clean API and handling asynchronous updates in the UI.
- **Testing**: Property-based testing helped uncover edge cases and improve reliability.
- **User Experience**: Providing clear feedback and a responsive interface was key for usability.

Conclusion

This project demonstrates the power and flexibility of functional programming in Scala for building real-world applications. By combining a robust backend with a modern web interface, we created a system that is both reliable and user-friendly. The modular architecture, type safety, and comprehensive testing ensure maintainability and extensibility for future development.