

Scala project

Juillet 2025

Albane Coiffe

Maelwenn Labidurie

Constance Walusiak

Louise Lavergne

Amira Boudaoud



Table of content

Introduction	3
Link of the demo video of our application	3
Link to github repository of the project :	3
System Architecture	4
Overview of Project Structure	4
Main Source Code Structure (src/main/scala)	5
Backend – Functional Core in Scala 3	6
Data Flow and Interaction	7
Web Interface Overview (/public)	7
Structure and Components	8
index.html	8
app.js	8
How the Frontend Works	9
Key Features	10
Domain Model (Scala)	10
Book.scala	11
User.scala	11
Transaction.scala	12
Business Logic & Services	12
LibraryCatalog.scala	13
Structure	13
Functional Operations	13
Search	13
Book Loan	13
Book Return	13
Recommendations	14
Synchronization	14
LibraryCatalogCodec.scala – JSON Serialization	14
Time Handling	14
Custom Decoding for User	14
Custom Decoding for Transaction	14
Full Catalog Encoder/Decoder	14
Testing Strategy	15
Overview	15
Test Structure	15
Test Coverage Report Summary	18
Key Metrics	19

Running the Console Application	19
sbt "runMain LibraryApp"	19
Challenges and Lessins Learned	21
Critical Analysis and Perspectives	21
Conclusion	22

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 and search bar
│   └── app.js             # Frontend logic (fetch, DOM updates)
├── src/
│   ├── main/scala/
│   │   ├── api/          # DTOs for REST API (ApiModels.scala)
│   │   ├── models/       # Domain models (Book, User, Transaction)
│   │   ├── services/     # Business logic (LibraryCatalog, Codec)
│   │   ├── utils/        # JSON I/O + Scala 3 extensions
│   │   ├── LibraryApp.scala # Optional CLI-based app (legacy)
│   │   ├── LibraryServer.scala # Akka HTTP server for REST API
│   │   └── Test.scala     # Experimental / dev script
│   └── test/scala/
│       ├── api/          # API model tests
│       ├── models/       # Domain model tests
│       ├── services/     # Logic, codec, and integration tests
│       └── utils/        # File I/O, error handling, coverage tests
```

```

|   └─ FinalCoverageBoostTest.scala    # High-coverage test file
|─── Data/                             # Persistent JSON data (Library.json)
|─── build.sbt                         # Build configuration (SBT)
└─ README.md                          # Project documentation

```

Main Source Code Structure (**src/main/scala**)

The main Scala source code is organized into modular packages, each serving a specific responsibility:

Path	Purpose
api/ApiModels.scala	API layer DTOs: defines case classes like LoanRequest , ApiResponse .
models/Book.scala	Defines the Book case class and ISBN type alias.
models/User.scala	User models with sealed trait User and its subtypes (Student , etc.).
models/Transaction.scala	Transaction trait and subtypes: Loan , Return , Reservation .
services/LibraryCatalog.scala	Functional core of the app: search, loan, return, recommend, stats, etc.
services/LibraryCatalogCodec.scala	Circe JSON codecs for all domain models.

utils/JsonIO.scala	File I/O helpers for saving and loading the full catalog as JSON.
utils/UnionAndExtensionExamples.scala	Demonstrates advanced Scala 3 features like union types and extensions.
LibraryApp.scala	Legacy command-line interface (CLI), now superseded by the REST server.
LibraryServer.scala	Akka HTTP-based REST API server.
Test.scala	Exploratory or debug utility file (not used in production flow).

This modular structure follows the separation of concerns principle, ensuring clarity and scalability in code organization.

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.
- Uses **type aliases** like `ISBN`, `UserID` for improved clarity.
- `Transaction` types (`Loan`, `Return`, `Reservation`) carry timestamps and user associations.

Service Layer (`services/`)

- `LibraryCatalog` is the **pure functional core**, supporting:

- Searching books by title/author
 - Borrowing/returning/reserving
 - Recommending books based on user history
 - Statistics (top genres/authors)
- All methods are **pure**, deterministic, and testable.
- **synchronizeBookAvailability** ensures internal state matches 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

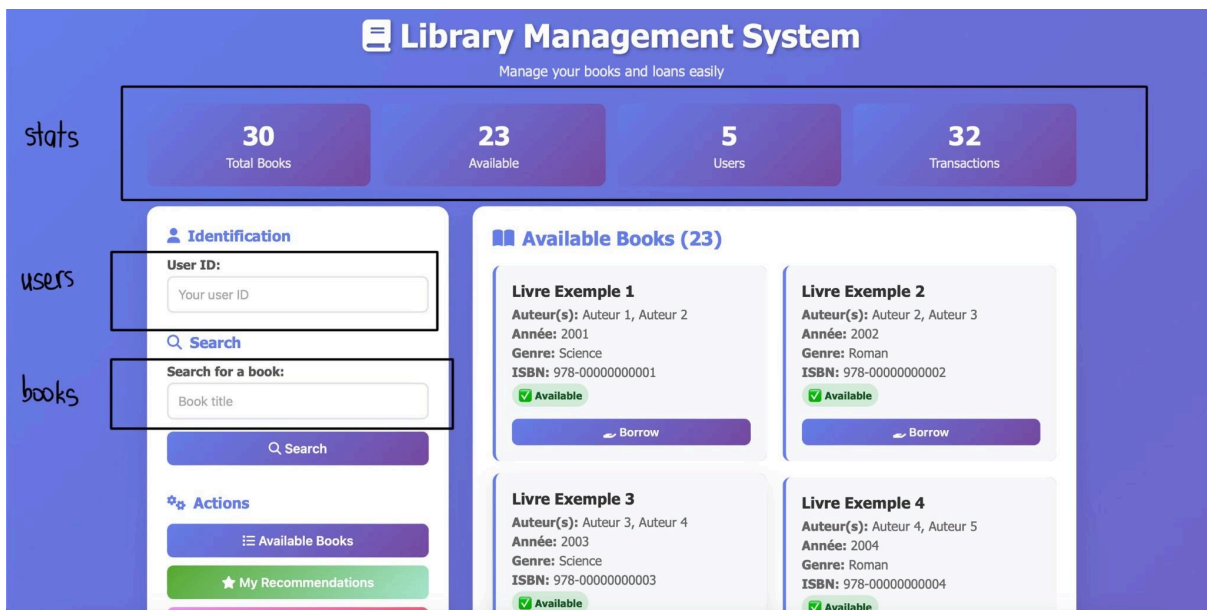
```
## Running the Web Server

sbt run

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

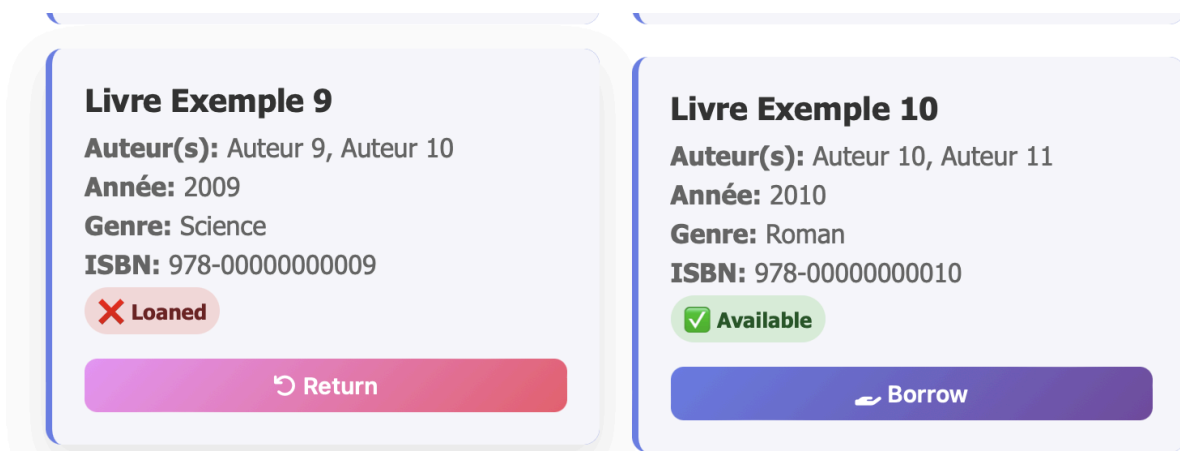
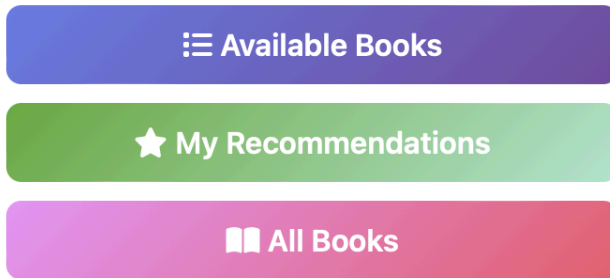
Here's a typical user interaction flow:

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`.

⚙️ Actions



- 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

Overview

Testing in this project is handled using the **ScalaTest** and **ScalaCheck** frameworks. The objective was to validate both the **correctness** and **resilience** of the system across all layers, from domain logic to API integration and persistence.

The test suite is composed of multiple strategies:

- **Unit tests** for fine-grained logic (using **AnyFunSuite**)
- **Property-based tests** with random data generators (**AnyPropSpec** + **ScalaCheck**)
- **Persistence tests** for JSON I/O and deserialization consistency
- **Integration tests** simulating full workflows (loan → return → recommend → save/load)
- **Error-handling tests** targeting invalid inputs, malformed files, and decoding failures

Test Structure

The test suite is organized under **src/test/scala/** and structured by functional domain:

Path	File(s)	Purpose
services/	LibraryCatalogTest.scala LibraryAppCoverageTest.scala LibraryCatalogCoverageTest.scala LibraryAppFullCoverageTest.scala	Unit tests for core logic: loan, return, recommend, statistics, and CLI behavior
	LibraryCatalogPropertyTest.scala	Property-based tests using random input generators

	<code>LibraryIntegrationTest.scala</code>	Full workflow integration test: loan → return → recommend
	<code>LibraryCatalogCodecTest.scala</code> <code>LibraryCatalogCodecFullCoverageTest.scala</code> <code>LibraryCatalogCodecAgressiveTest.scala</code> <code>LibraryCatalogCodecDirectTest.scala</code>	Serialization/deserialization of catalog and domain models
	<code>LibraryServer*Test.scala</code> (e.g. <code>LibraryServerLogicCoverageTest.scala</code> , <code>LibraryServerHttpErrorCoverageTest.scala</code> , etc.)	HTTP API tests: routing, logic, error handling, integration, edge cases
<code>models/</code>	<code>BookCoverageTest.scala</code> <code>UserCoverageTest.scala</code> <code>TransactionCoverageTest.scala</code>	Tests for domain models and their variants
<code>utils/</code>	<code>JsonIOTest.scala</code> <code>JsonIOCoverageTest.scala</code> <code>JsonIOErrorCoverageTest.scala</code> <code>JsonIOFullCoverageTest.scala</code>	JSON I/O correctness, error management, and file consistency
	<code>UnionAndExtensionExamplesTest.scala</code>	Scala 3 language features (union types,



	<code>UnionAndExtensionExamplesFullCoverageTest.scala</code>	extension methods, pattern matching, etc.)
<code>api/</code>	<code>ApiModelsFullCoverageTest.scala</code> <code>ApiResponseFullCoverageTest.scala</code>	API DTO serialization/deserialization and equality checks
<code>misc/</code>	<code>TestMainFullCoverageTest.scala</code> <code>TestScala3Syntax.scala</code> <code>QuickBoostCoverageTest.scala</code> <code>FinalCoverageBoostTest.scala</code>	Syntax validation and extra tests for 100% coverage
	<code>LibraryApp*CoverageTest.scala</code> (e.g. <code>LibraryAppMaxCoverageTest.scala</code> , <code>LibraryAppSessionCoverageTest.scala</code> , etc.)	Additional CLI-related tests (loop, sessions, complete coverage variants)
	<code>LibraryServer*CoverageTest.scala</code> (e.g. <code>LibraryServerFinalCoverageTest.scala</code> , <code>LibraryServerRoutingCoverageTestFixed.scala</code> , etc.)	Granular HTTP logic and routing coverage with error branches

Test Coverage Report Summary

All packages											
utils	88.00%	Lines of code:	869	Files:	9	Classes:	10	Methods:	62		
services	50.88%	Lines per file:	96.56	Packages:	5	Classes per package:	2.00	Methods per class:	6.20		
models	100.00%	Total statements:	1420	Invoked statements:	951	Total branches:	28	Invoked branches:	27		
api	100.00%	Ignored statements:	0								
<empty>	84.28%	Statement coverage:	66.97 %	<div><div></div></div>				Branch coverage:	96.43 %	<div><div></div></div>	
Class	Source file	Lines	Methods	Statements	Invoked	Coverage	Branches	Invoked	Coverage		
LibraryApp\$	LibraryApp.scala	325	10	336	280	<div><div></div></div>	83.33 %	22	21	<div><div></div></div>	
LibraryServer\$	LibraryServer.scala	216	7	302	256	<div><div></div></div>	84.77 %	0	0	<div><div></div></div>	
TestMain\$	Test.scala	7	1	11	11	<div><div></div></div>	100.00 %	0	0	<div><div></div></div>	
ApiResponse\$	ApiModels.scala	8	2	2	2	<div><div></div></div>	100.00 %	0	0	<div><div></div></div>	
Book	Book.scala	12	1	3	3	<div><div></div></div>	100.00 %	0	0	<div><div></div></div>	
\$anon	LibraryCatalogCodec.scala	169	5	381	110	<div><div></div></div>	28.87 %	0	0	<div><div></div></div>	
LibraryCatalog	LibraryCatalog.scala	97	15	102	80	<div><div></div></div>	78.43 %	4	4	<div><div></div></div>	
LibraryCatalogCodec\$package\$	LibraryCatalogCodec.scala	167	16	258	187	<div><div></div></div>	72.48 %	1	1	<div><div></div></div>	
JsonIO\$	JsonIO.scala	22	3	19	16	<div><div></div></div>	84.21 %	1	1	<div><div></div></div>	
UnionAndExtensionExamples\$package\$	UnionAndExtensionExamples.scala	13	2	6	6	<div><div></div></div>	100.00 %	0	0	<div><div></div></div>	

By the end of the testing phase, the test suite reached a total of **66.97% statement coverage**, up from the initial **8%**, thanks to extensive additions across all components.

Package	Coverage	Notes
models	✓ 100.00%	All case classes like Book , User , and Transaction fully covered
api	✓ 100.00%	DTOs (LoanRequest , ReturnRequest , ApiResponse , etc.) fully tested
utils	● 88.00%	Includes JsonIO.scala , various edge case and error handling tested

services	 50.88%	Business logic (e.g. <code>LibraryCatalog</code> , <code>Codec</code>) – many tests written, but coverage limited due to complexity
<empty>	 84.28%	Includes main applications like <code>LibraryApp.scala</code> , <code>LibraryServer.scala</code> , <code>TestMain.scala</code>

Key Metrics

- **Total coverage:** 66.97%
- **Statement coverage:** 951 / 1420 statements
- **Branch coverage:** 27 / 28 branches

Goal: While the original target was 80%, reaching nearly **67%** with broad branch coverage (96%) provides confidence in system reliability.

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 - 978-0000000001
■ Livre Exemple 2 - 978-0000000002
■ Livre Exemple 3 - 978-0000000003
■ Livre Exemple 4 - 978-0000000004
■ Livre Exemple 5 - 978-0000000005
■ Livre Exemple 10 - 978-0000000010
■ Livre Exemple 13 - 978-0000000013
■ Livre Exemple 14 - 978-0000000014
■ Livre Exemple 15 - 978-0000000015
■ Livre Exemple 17 - 978-0000000017
■ Livre Exemple 18 - 978-0000000018
■ Livre Exemple 19 - 978-0000000019
■ Livre Exemple 20 - 978-0000000020
■ Livre Exemple 21 - 978-0000000021
■ Livre Exemple 22 - 978-0000000022
■ Livre Exemple 23 - 978-0000000023
■ Livre Exemple 24 - 978-0000000024
■ Livre Exemple 25 - 978-0000000025
■ Livre Exemple 26 - 978-0000000026
■ Livre Exemple 27 - 978-0000000027
■ Livre Exemple 28 - 978-0000000028
■ Livre Exemple 29 - 978-0000000029
■ Livre Exemple 30 - 978-0000000030
```

```

-----
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: 978-0000000001
✔ 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: 978-0000000001
✔ Book returned 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: 4
Your user ID: 1
♥ Recommendations:
- Livre Exemple 2 (Roman)
- Livre Exemple 4 (Roman)
- Livre Exemple 10 (Roman)
- Livre Exemple 14 (Roman)
- Livre Exemple 18 (Roman)
- Livre Exemple 20 (Roman)
- Livre Exemple 22 (Roman)
- Livre Exemple 24 (Roman)
- Livre Exemple 26 (Roman)
- Livre Exemple 28 (Roman)
- Livre Exemple 30 (Roman)
- Livre Exemple 1 (Science)
- Livre Exemple 3 (Science)
- Livre Exemple 5 (Science)
- Livre Exemple 13 (Science)
- Livre Exemple 15 (Science)
- Livre Exemple 17 (Science)
- Livre Exemple 19 (Science)
- Livre Exemple 21 (Science)
- Livre Exemple 23 (Science)
- Livre Exemple 25 (Science)
- Livre Exemple 27 (Science)
- Livre Exemple 29 (Science)

-----
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: 6
👋 Thank you for using our system. See you soon!
[success] Total time: 152 s (0:02:32.0), completed 24 juil. 2025, 11:59:07
(base) constancewalusiak@bpbdeconstance project-root %

```

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

```
-----  
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: q  
✗ Error: Book not available or User not found
```

Challenges and Lessins Learned

Throughout the project, ensuring immutability and pure functions required careful design, especially for catalog updates and state management. Integrating the frontend and backend, as well as designing a clean API, presented challenges in handling asynchronous updates and maintaining a responsive user interface. Property-based testing was particularly valuable for uncovering edge cases and improving reliability. Providing clear feedback and usability in the interface was also a key focus.

Critical Analysis and Perspectives

While the library management system meets the main functional requirements and demonstrates functional programming principles, several limitations and areas for improvement remain. The current implementation relies on in-memory data structures, which limits scalability and persistence. Error handling is present but could be further refined, especially for invalid actions and edge cases. The user interface, while functional, could be enhanced for better usability.

From a technical perspective, the use of immutable data models and pure functions improves reliability and testability. However, operations such as file I/O could benefit from more explicit effect management. The modular design allows for easy addition of new features (e.g., reservations, reviews), and future versions could integrate authentication, authorization, and database support for scalability and security. Asynchronous operations and optimized data structures could further enhance performance.

For future work, integrating a persistent database, improving concurrency management, and expanding the user interface would make the system more production-ready. Exploring advanced Scala features could also provide educational value and further demonstrate the power of functional programming.

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.