Library Management System – Scala 3 Project

Design Document

Academic Year: 2024/2025

1. Project Overview

This project implements a functional library management system in Scala 3 using a modern web-based interface. It allows users to search, borrow, and return books, view statistics, and receive recommendations based on borrowing history. The system is built using modern Scala programming paradigms, including algebraic data types (ADTs), higher-order functions, and functional error handling.

The backend logic is designed to be pure and modular, using immutable data structures and type-safe models. The frontend is responsive and interacts with the backend through a RESTful API built with Akka HTTP.

2. Architectural Structure

The project is divided into the following key components:

- **Models:** Define the core data types such as `Book`, `User`, and `Transaction`. These are implemented using `case class` and `sealed trait` to enable strong pattern matching and type safety.
- **Services (LibraryCatalog):** Encapsulates business logic including book search, loan/return operations, and recommendation generation. This class is purely functional and returns updated versions of the catalog instead of mutating state.
- **Utilities (JsonIO):** Manages reading from and writing to a JSON file using Circe. It abstracts away file I/O and handles decoding/encoding errors using `Either`.
- **Console Interface (LibraryApp):** Provides a text-based interactive menu for testing and demonstrating core functionality directly from the command line.
- **Web Interface (LibraryServer, app.js, index.html):** Exposes a RESTful API via Akka HTTP and a frontend UI built with HTML/CSS/JavaScript. It includes features like search, borrow/return actions, and a statistics dashboard.

3. Key Design Decisions

- **Functional Purity:** The `LibraryCatalog` is designed to be immutable. Each function that modifies data returns a new instance, supporting referential transparency and easier testing.
- **Algebraic Data Types:** The system uses sealed traits and case classes to define transactions (`Loan`, `Return`, `Reservation`) and user types (`Student`, `Faculty`, `Librarian`). This provides strong compile-time safety and exhaustiveness checking.
- **Error Handling with `Either`:** Instead of throwing exceptions, all business logic operations return `Either[String, ResultType]` to represent success or failure in a functional way.
- **JSON Persistence with Circe:** The system uses Circe's automatic and semi-automatic derivation for JSON encoding/decoding. Encoders and decoders are defined as `given` instances to leverage Scala 3's type class syntax.

- **RESTful API Design:** The API is designed using REST principles with distinct endpoints for books, users, and transactions. POST requests are used for state-changing operations like borrow and return.
- **Frontend Separation:** The frontend is fully decoupled from the backend, interacting only through REST endpoints. This ensures modularity and allows for future replacements or upgrades of either side independently.
- **Synchronization Logic:** A method `synchronizeBookAvailability` recalculates book availability based on transaction history, ensuring data consistency when reloading from file.

4. Use of Scala 3 Features

- **Given/Using Clauses:** Used in defining Circe encoders/decoders and for JSON serialization.
- **Opaque Types:** Applied to domain-specific types like `ISBN` and `UserID` for type safety without runtime cost.
- **Extension Methods:** Could be used to add utility functions to core types (optional enhancement).
- **Enums and ADTs:** Used in the model layer to represent different kinds of transactions and users.

5. Test Strategy

- Unit tests were written using ScalaTest to verify core business logic (e.g., borrowing a book updates its availability).
- Property-based tests using ScalaCheck validate consistent behavior under randomly generated inputs.
- IO tests ensure that the file-saving and loading mechanism behaves correctly and handles errors gracefully.

6. Scalability and Modularity

- The codebase is organized into well-defined modules: models, services, utils, API, and UI.
- Each component can evolve independently (e.g., replacing JSON with a database or using a frontend framework).
- The catalog is kept in memory and persisted to disk, which makes it simple but limits concurrent scalability (a trade-off accepted for this educational project).

7. Assumptions and Limitations

- User authentication is not implemented; the system assumes user IDs are known.
- The data is stored in a local JSON file; no concurrency or multi-user locking mechanisms are in place.
- Recommendations are based on past loan history by genre only.

8. Conclusion

This project demonstrates effective use of functional programming principles in a real-world application. It balances clarity, modularity, and correctness by leveraging Scala 3's type system, modern syntax, and ecosystem libraries. The code is maintainable, testable, and extendable, laying a strong foundation for future enhancements such as real-time database integration, role-based access control, or reactive streaming of events.