SOFTWARE DESIGN DOCUMENT
for
PERSONAL LIBRARY MANAGEMENT SYSTEM

Issue No: 1.0

Issue Date: Dec 2025

Prepared by

Ahmet Mücahit Gündüz

Enes Korkmaz

Onurhan Talan

Rümeysa Kalay

Veysel Cemaloğlu

Table of Contents

1 Introduction

1.1 Purpose of the system

The Personal Library Management System (PLMS) is a centralized web platform developed using the **MERN Stack** (MongoDB, Express, React, Node.js). It adheres to a **Client-Server architectural style** utilizing a **RESTful API** for all communication. The system is designed to provide individuals with a comprehensive and intelligent solution to manage, track, and enrich their personal collections of mixed media (Books and Movies).

This intelligent platform provides detailed inventory management and enables users to:

- Catalog physical media items efficiently.
- Integrate with online APIs (Google Books, OMDB, Gemini AI) for detailed metadata and personalized content recommendations.
- Track reading and viewing progress with granular progress tracking.
- Build customized lists (e.g., “Favorites,” “Watch Later”).

The PLMS aims to reduce common inefficiencies (such as lost items or duplicate purchases), while enhancing the organization and enjoyment of personal media collections.

1.2 Design goals

The design of the PLMS is driven by the necessity to fulfill the most critical functional and non-functional requirements (FRs and NFRs). These goals dictate the choice of the MERN stack and its specific configuration:

Design Goal	Rationale / Specific Implementation
High Usability & Responsiveness	The architecture must deliver a seamless user experience using a **React SPA** with the component library **Material UI** (3.2) to support a responsive, efficient UI (NFR-PERF-1).

Maintainability & Scalability	The system is built on four decoupled subsystems (3.2), enhancing modularity and isolation (NFR-SUP-1). **Mongoose Discriminators** (3.4) are specifically used to ensure the data model can easily handle polymorphism for both **Book** and **Movie** schemas.
Security & Authorization	Authentication is delegated exclusively to **Google OAuth 2.0** (3.5). Authorization relies on **JWT** and strict **user-level security** (3.5), ensuring a user can only manipulate data where User.id == Document.ownerId. All network traffic is secured via **HTTPS (TLS 1.2+)**.
Reliability & Graceful Degradation	The system must be robust against external dependencies. The **External Integration Subsystem** (3.2) implements explicit **Graceful Degradation** logic (3.7), allowing the core application (CRUD and lists) to function even if external metadata APIs (Google Books, OMDB) fail.

1.3 Definitions, acronyms, and abbreviations

Acronyms and Abbreviations

- **API** – Application Programming Interface
- **CRUD** – Create, Read, Update, Delete
- **DVD** – Digital Versatile Disc
- **ISBN** – International Standard Book Number
- **OAuth 2.0** – Open standard for access delegation
- **UI** – User Interface

- **AI** – Artificial Intelligence
- **TLS** – Transport Layer Security
- **JSON** – JavaScript Object Notation
- **SRS** – Software Requirements Specification
- **PLMS** – Personal Library Management System
- **REST** – Representational State Transfer
- **SPA** – Single Page Application (the React Frontend).
- **JWT** – JSON Web Token. The secure, stateless mechanism used by the system for session management after OAuth authentication.
- **MongoDB** – The NoSQL database used for persistent storage.
- **VPS** – Virtual Private Server

Definitions

- **MERN Stack**: MongoDB, Express.js, React.js, Node.js. The chosen technology stack for the web application.
- **Mongoose** An Object Data Modeling (ODM) library used to manage data relationships and schema definition for MongoDB.
- **Graceful Degradation:** System behavior where core features remain operational even when external APIs fail.
- **Vite** – A modern frontend build tool
- **Node.js** – JavaScript runtime environment for server-side applications.
- **OMDB API** – The external third-party API used by the system to fetch rich metadata for Movies.
- **Gemini API** – The external Artificial Intelligence (AI) API used to generate personalized media recommendations.

1.4 References

[PLMS-Problem]: *Personal Library Management System - Problem Statement*. (Provided `Personal Library Management System.docx` document) .

[SRS-PLMS]: Requirements Analysis Document for Personal Library Management System.

[Larman-OOAD]: Larman, C. (2005). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design* (3rd ed.). Prentice-Hall.

[API-GoogleBooks]: Google Books API. (External API to be integrated by the system).

Google Books APIs | Google for Developers

[API-OMDb]: OMDb API. (External API to be integrated by the system).

OMDb API - The Open Movie Database

Akış - 1000Kitap
Letterboxd • Social film discovery.
Google Kitaplar
IMDb: Ratings, Reviews, and Where to Watch the Best Movies & TV Shows
Rotten Tomatoes: Movies | TV Shows | Movie Trailers | Reviews | Rotten Tomatoes
Movie Reviews, TV Reviews, Game Reviews, and Music Reviews - Metacritic

1.5 Overview

This document is the official **Software Design Document (SDD)**, detailing the **how** of the Personal Library Management System (PLMS) by specifying its conceptual structure, architecture, and technology choices.

- **Section 1 (Introduction):** Introduces the PLMS, provides a brief overview of the high-level **MERN Stack** software architecture, outlines the key **Design Goals** (such as Usability, Modularity, and Security), and lists necessary references.
- **Section 2 (Current Software Architecture):** Replaces a description of a previous system with a survey of the problems in fragmented, traditional media management methods, making explicit the functional and quality gaps that the new PLMS architecture is designed to address.
- **Section 3 (Proposed Software Architecture):** Documents the system design model of the new PLMS. This section is the primary source for all structural and technical design decisions, including:
    - **3.2 Subsystem Decomposition:** Defines the system's modular components (Client UI, API Gateway, Business Logic, Persistence, Connectors) based on the MERN stack and object-oriented principles..

- **3.3 Hardware/Software Mapping:** Specifies the deployment architecture, detailing how components (e.g., React, Node/Express, MongoDB) are mapped onto runtime environments using Docker containers.
- **3.4 Persistent Data Management:** Describes the use of **MongoDB** and the mechanisms for data encapsulation and ensuring relational integrity across the NoSQL schema.
- **3.5 Access Control and Security:** Details the security framework, including the OAuth 2.0 authentication mechanism, the access control matrix, and policies for encryption and secrets management.
- **3.6 Global Software Control:** Defines the event-driven control flow, explaining how asynchronous requests are initiated, processed, and managed for concurrency.
- **3.7 Boundary Conditions:** Addresses system behaviors for startup, graceful shutdown, and crucial **Error Handling** procedures like **Graceful Degradation** upon external API failure.

- **Section 4 (Subsystem Services):** Describes the high-level services and interfaces (e.g., REST API endpoints) exposed by each subsystem, serving as the blueprint for the subsequent Object Design Document.

This document will serve as the primary reference for the implementation, development, and maintenance teams.

2 Current software architecture

2.1 Brief Overview

The system being replaced is a manual or semi-digital personal media tracking approach used to record watched movies, TV series, and books. Its primary purpose is basic record keeping, allowing users to list media items without providing active assistance, personalization, or advanced interaction capabilities.

2.2 Architectural Assumptions

The replaced system is based on several implicit architectural assumptions:

- Media data is managed locally and manually without backend support.
- Data storage, presentation, and user interaction occur within a single, tightly coupled environment.

- User interaction history is not treated as analyzable or reusable data.
- The system is intended for individual, single-device use rather than synchronized multi-device access.

2.3 Functional and Non-Functional Characteristics

Functionally, the replaced system supports only basic media listing and manual annotation. Users can record media items but must manage organization, consistency, and updates themselves.

From a non-functional perspective, the system exhibits limited modularity and extensibility. It provides minimal support for usability at scale, lacks automation, and offers no built-in support for performance, portability, or integration with external services.

2.4 Limitations and Problems

The architectural characteristics of the replaced system lead to several significant limitations:

- Inability to support structured search, filtering, or multiple views of media collections.
- Absence of flexible list and tag management. Media items cannot be easily grouped into multiple lists, tagged, or reorganized dynamically, limiting user-defined categorization and exploration.
- Lack of progress tracking for reading or watching activities, such as tracking the current page in a book or the viewing status of a series.
- Absence of personalization, recommendation, or intelligent assistance based on user interaction history.
- No visibility into user behavior or consumption statistics, including frequently viewed categories, authors, or directors.
- Manual and error-prone data entry and metadata enrichment processes that rely on user effort.
- No support for data portability, structured import/export, or reuse across devices and platforms.

2.5 Implications for Replacement and Derived Requirements

The limitations identified in the replaced system lead to a set of derived requirements that guide the design of the Personal Library Management System. These requirements reflect both functional needs and architectural expectations for the replacement system.

- **Server-managed persistent storage**

The system should store user and media data in a centralized and durable backend with proper user isolation. This enables cross-device continuity, reliable data retention, and future extensibility of storage mechanisms.

- **Advanced search and filtering capabilities**

The system should support structured search and multi-criteria filtering to allow efficient discovery and exploration of large media collections.

- **Rich tagging and flexible organization model**

Media items should support tagging, user-defined lists, and typed metadata to enable personalized organization and flexible classification.

- **Interactive progress tracking and synchronization**

The system should persist per-user progress information, such as reading progress or status, and synchronize this data across user sessions and devices.

- **Personalized recommendation capability**

The system should provide personalized content recommendations based on user interaction history, including watched or read items, tags, and feedback.

- **External metadata enrichment**

Media items should be enriched through integration with external metadata providers to reduce manual data entry and improve data quality.

- **Data portability and reuse**

The architecture should support structured import and export of user data to enable migration, backup, and user ownership of stored information.

- **Security, privacy, and data protection**

The architecture should support secure handling of user data, access control, and data lifecycle management to protect privacy and prevent unauthorized use.

- **Scalability and performance considerations**

The system should be designed to accommodate growth in data volume and user activity without significant architectural changes.

- **Extensible integration points**

The architecture should define clear and extensible interfaces to support future integrations, such as recommendation services, metadata providers, and data exchange components.

2.6 Summary

In summary, the replaced system represents a passive, single-tier approach to personal media tracking focused on manual record keeping. Its architectural limitations in usability, extensibility, and intelligence motivate the design of a new system that actively supports users through structured data management, personalization, and intelligent recommendations.

3 Proposed software architecture

3.1 Overview

The Personal Library Management System (PLMS) follows a **Client-Server architectural style** utilizing a RESTful API for communication. The system is designed to provide a centralized platform for users to manage mixed media collections (Books and Movies) with automated metadata enrichment and progress tracking.

The architecture is decoupled into two main components:

- **The Client (Frontend):** A Single Page Application (SPA) implemented with **React**, responsible for the user interface, state management, and user interaction. **Vite** is used as the build tool for production.
- **The Server (Backend):** A **Node.js/Express** application that serves as the API gateway, handling business logic, authentication, and data persistence.

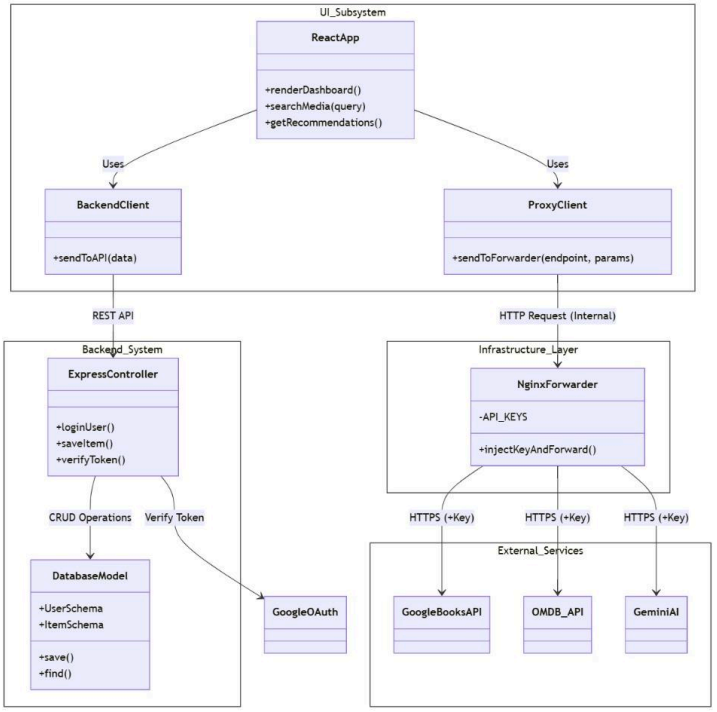
Data persistence is managed by **MongoDB**, a NoSQL database selected for its flexibility in handling diverse media schemas. The system integrates with external third-party services—**Google Books API** and **OMDB API**—to fetch rich metadata, **Gemini AI** for personalized content suggestions, and **Google OAuth** for secure authentication.

3.2 Subsystem decomposition

The system is decomposed into four logical subsystems, each with distinct responsibilities to ensure modularity and maintainability.

- **User Interface (UI) Subsystem:**
  - **Responsibility:** Renders the application views (Library Dashboard, Item Details, Search) and captures user events. It utilizes **Material UI** for consistent styling.
  - **Components:** React Router (navigation), Auth Context (session management), and API Service (HTTP client).
  - **Inputs:** User clicks, form data.
  - **Outputs:** Visualized library data, error notifications.
- **Application Logic (API) Subsystem:**

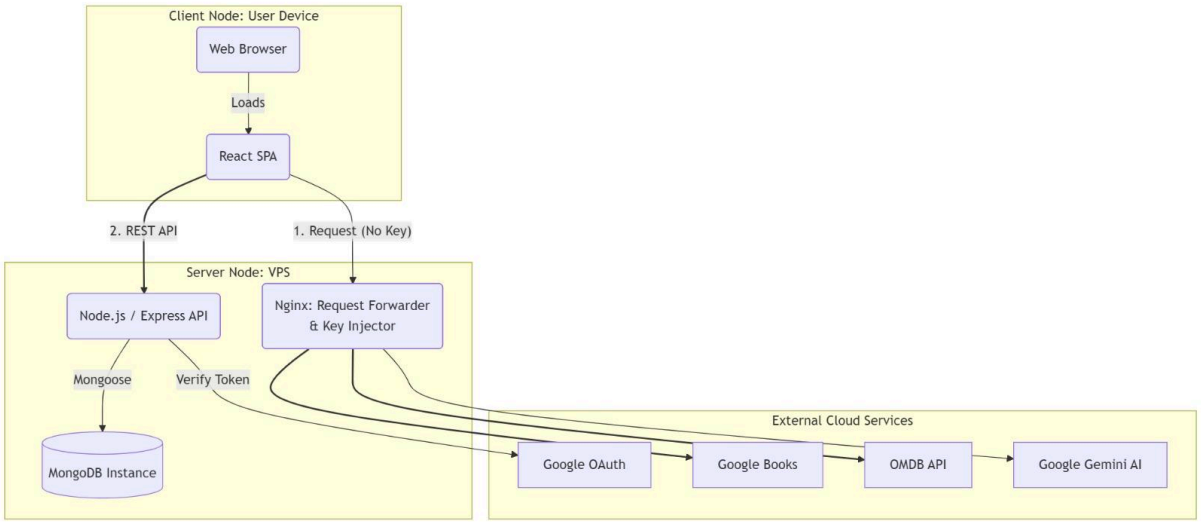
    - **Responsibility:** Hosted on a VPS, this subsystem processes RESTful requests, enforces business rules (request & access validation), and manages sessions via JWT.
    - **Components:** Express.js Controllers (request handling), Middleware (auth verification, validation), and Mongoose Models (data definition).
    - **Inputs:** HTTP Requests (GET, POST, PUT, PATCH, DELETE).
    - **Outputs:** JSON responses, HTTP Status Codes.
- **External Integration Subsystem:**
    - **Responsibility:** Acts as an adapter layer to communicate with third-party services. It normalizes data formats from different providers into the system's internal schema.
    - **Interfaces:**
        - **Metadata Connectors:** Google Books API (Books) and OMDB API (Movies).
        - **Identity Connector:** Google OAuth.
        - **AI Service Connector: Gemini API** (Recommendations). This connector handles data privacy checks before transmitting user history.
- **Data Access Subsystem:**
    - **Responsibility:** Manages direct interaction with the MongoDB database.
    - **Key Schemas:** **`User`**, **`MediaList`**, **`MediaItem`** (with **`Book`** and **`Movie`** discriminators), **`Tag`**.



3.3 Hardware/software mapping

The system is designed for cloud deployment on a Virtual Private Server (VPS).

- **Hardware Nodes:**
  - **Client Node:** The end-user's device (Laptop, Mobile) running a modern web browser.
  - **Server Node:** A VPS instance hosting the Backend API, the Nginx, and the MongoDB database instance.
- **Software Components & COTS:**
  - **Web Server:** Nginx.
  - **Runtime Environment:** Node.js.
  - **Database:** MongoDB.
  - **External APIs:** Google Books, OMDB, Gemini API.
  - **Libraries/Frameworks:** React (Frontend), Vite (Frontend Production Builds), MaterialUI (UI Components), NodeJS (Backend), Express (HTTP Server), Mongoose (ODM), JWT (Security), OpenAPI (API Documentation).



3.4 Persistent data management

The system utilizes **MongoDB** for persistent storage. This choice aligns with the need to store JSON-like documents with varying fields for different media types (Books vs. Movies). The expected data volume is low-to-moderate (Megabytes per user), allowing for efficient vertical scaling on the VPS.

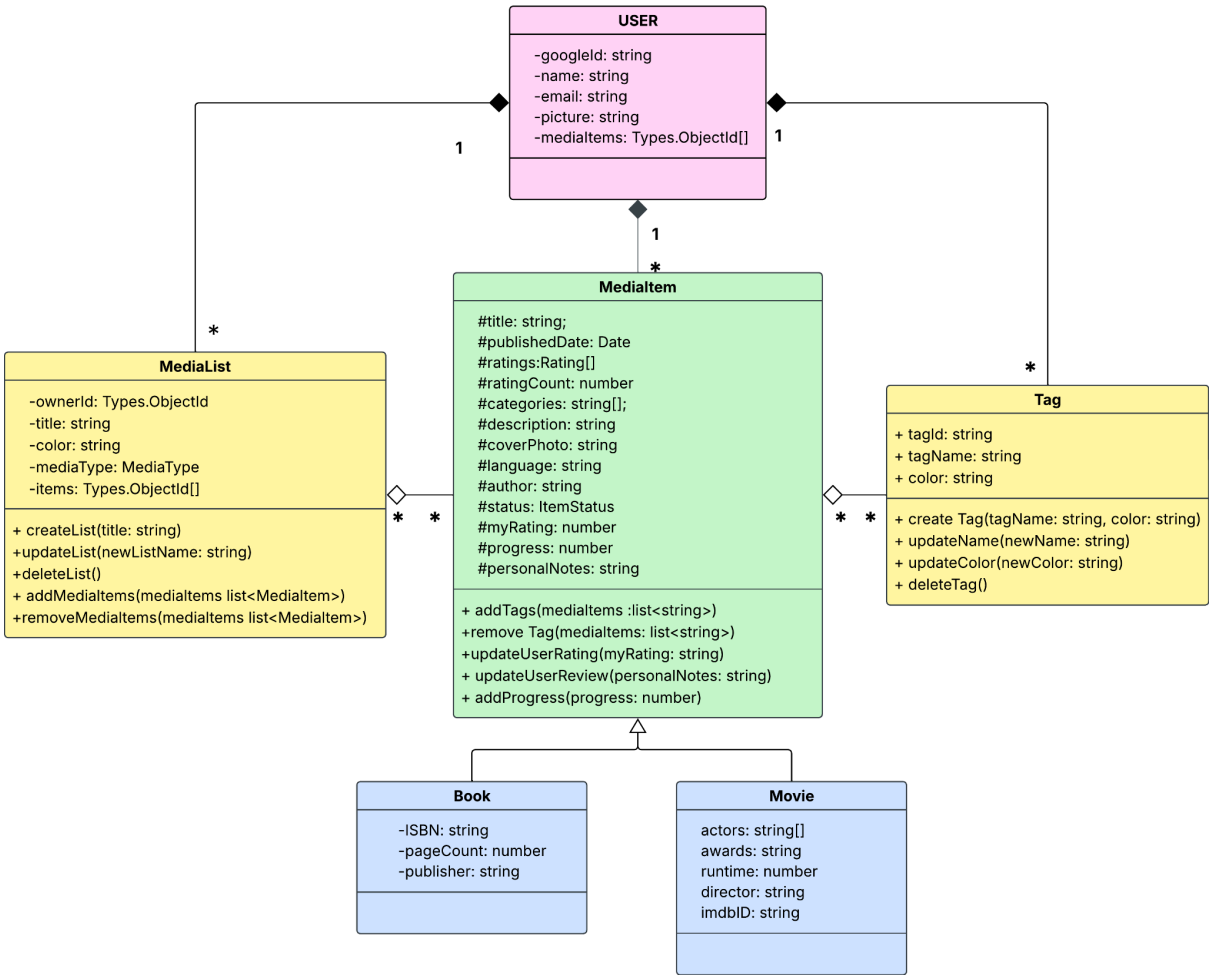
Note: AI-generated recommendations are stateless and are **not** persisted in the database; they are fetched on-demand to ensure relevance.

Data Schemas: The database schema utilizes Mongoose discriminators to handle polymorphism between different media types.

1. **User Schema:** Stores account details and references to the user's library.
    - Fields:
        i. **`googleId`** (unique)
        ii. **`email`**
        iii. **`mediaItems`** (array of references)
        iv. **`lists`** (array of references).
        v. tags (array of references)
2. **MediaItem Schema (Base):** Stores attributes common to all items.

- Fields:
    i. **`title`**
    ii. **`description`**
    iii. **`coverPhoto`**
    iv. **`myRating`**
    v. **`status`**
    vi. **`progress`**

3. **Book Schema (Discriminator):** Extends MediaItem.
    - Fields:
        i. **`ISBN`**
        ii. **`pageCount`**
        iii. **`publisher`**
4. **Movie Schema (Discriminator):** Extends MediaItem.
    - Fields:
        i. **`director`**
        ii. **`runtime`**
        iii. **`actors`**
        iv. **`imdbID`**
5. **MediaList Schema:** Manages custom user collections.
    - Fields:
        i. **`title`**
        ii. **`ownerId`** (ref User)
        iii. **`items`** (array of MediaItem refs)
        iv. **`color`**
6. **Tag Schema:** Stores user-defined labels to support flexible categorization across the library.
    - Fields:
        i. **`name`**
        ii. **`ownerId`** (ref User)
        iii. **`color`**



3.5 Access control and security

Access control is implemented using a **User-Level Security Model**.

- **Authentication:** The system delegates authentication to **Google OAuth 2.0**. No user passwords are stored within the PLMS database. Upon successful login, the server issues a **JSON Web Token (JWT)** to the client.
- **Authorization:** The system enforces strict row-level security. API endpoints utilize middleware to decode the JWT and ensure the **`userId`** in the token matches the **`ownerId`** of the requested resource.
    - Rule: A user may only Create, Read, Update, or Delete data where **`User.id == Document.ownerId`**.
- **Data Security:**

    - **Transmission:** All data transfer between the Client and Server is encrypted via **HTTPS** (TLS 1.2+).
    - **Storage:** Sensitive tokens are managed securely, and the database relies on MongoDB's internal security protocols.
- **Data Privacy (AI):** Explicit user consent is required before using the recommendation feature. Only when permission is granted does the system transmit reading/watching history, ratings, and comments to the external Gemini API.

3.6 Global software control

The system follows an **Event-Driven Execution Flow**.

- **Flow Control:** Operations are triggered by user actions (events) on the React frontend. These events dispatch asynchronous HTTP requests to the Node.js backend.
- **Concurrency:** The system leverages the **Node.js non-blocking I/O model**. The single-threaded event loop efficiently handles concurrent requests from multiple users without creating a new thread for each connection.
- **Synchronization:** Data synchronization is strictly server-authoritative. Since users cannot modify other users' data, complex locking mechanisms are not required. Atomicity is ensured at the database document level using MongoDB transactions where necessary (e.g., deleting a List also updates the references in the User document).

3.7 Boundary conditions

This section defines the system behavior during startup, shutdown, and failure scenarios.

- **Startup / Initialization:**
    - The backend server loads configuration from environment variables (DB URI, API Keys).
    - It attempts to establish a connection handshake with the MongoDB instance.
    - Upon success, the Express application starts listening on the configured port.
    - If the DB connection fails, the server logs a critical error and exits (Fail-Fast).
- **Error Handling (Unhappy Paths):**
    - **Database Failure:** If the database becomes unreachable during operation, the API returns a **`500 Internal Server Error`** or **`503 Service`**

`Unavailable`. The Frontend intercepts this and displays a "Service Unavailable" notification to the user.
- **External API Failure:** If Google Books or OMDB APIs are down, the system gracefully degrades. The item is created with manual user input, and the user is notified that automatic metadata fetching failed.
- **Input Validation:** Invalid inputs (e.g., malformed email, negative page counts) are caught by the middleware, returning **`400 Bad Request`** responses to prevent data corruption.

4 Subsystem services

This section describes the services provided by each subsystem. It defines the specific operations that other subsystems or actors can invoke, establishing clear boundaries and interfaces.

4.1 User Interface (UI) Subsystem Services

This subsystem is responsible for rendering the application views and capturing user interactions. Based on the React frontend architecture, it provides the following view-level services:

- **Authentication View Service:**
  - `renderAuthPage()`: Displays the login interface (`AuthPage`) allowing users to sign in via Google OAuth.
  - `handleLoginSuccess(credentialResponse)`: Captures the Google ID token upon successful authentication and initiates the backend session creation.
- **Main Dashboard Service:**
  - `renderMainPage()`: Displays the home dashboard (`MainPage`), showing a summary of user activity, recent items, or quick actions.
- **Library Management View Service:**
  - `renderLibraryPage()`: Renders the user's book collection, providing options to filter, sort, and manage book items.
  - `renderMoviesPage()`: Renders the user's DVD/Movie collection (`MoviesPage`) with specific metadata fields like director and runtime.
  - `displayItemDetails(itemId)`: Opens a detailed view for a specific media item to show enriched metadata and progress.

- **Search and Filter Service:**
  - `searchLibrary(query)`: Performs a search operation across titles, authors, and tags based on the user's query.
  - `filterItems(criteria)`: Returns a subset of items matching specific filters (e.g., "Genre: Sci-Fi", "Status: Unread").

4.2 Application Logic (API) Subsystem Services

This subsystem hosts the Express.js server and provides RESTful endpoints to process business logic and data requests from the UI.

- **Authentication Service:**
  - `authenticateUser(idToken)`: Verifies the Google OAuth token sent from the client and establishes a user session (issues a JWT).
  - `validateSession(token)`: Checks if the incoming request contains a valid JWT for protected route access.
- **Inventory Management Service:**
  - `addItem(itemData)`: Validates and processes a request to add a new book or movie to the user's library.
  - `updateItemProgress(itemId, progress)`: Updates the reading (pages) or viewing (minutes) progress of a specific item.
  - `deleteItem(itemId)`: Removes an item from the library and ensures it is unlinked from any custom lists.
- **Custom List Service:**
  - `createList(listData)`: Processes the creation of a new custom list associated with the authenticated user.
  - `addItemToList(listId, itemId)`: Adds a reference of an existing media item to a specific custom list.
- **Tag Management Service:**
  - `createTag(tagData)`: Creates a new user-defined tag (label and color) in the database.
  - `assignTag(itemId, tagId)`: Associates a specific tag with a media item to enable filtered views.
  - `detachTag(itemId, tagId)`: Removes the link between a tag and a media item.

4.3 External Integration Subsystem Services

This subsystem acts as an adapter to communicate with third-party APIs for metadata and AI features.

- **Metadata Connector Service:**
  - `fetchBookMetadata(isbn)`: Connects to the **Google Books API** to retrieve book details (title, author, cover image) using an ISBN.
  - `fetchMovieMetadata(title)`: Connects to the **OMDB API** to retrieve movie details (director, runtime, poster) using a title.
- **AI Recommendation Service:**
  - `generateRecommendations(userHistory)`: Formats user history data and sends it to the **Gemini AI API** to receive personalized book/movie suggestions.

4.4 Data Access Subsystem Services

This subsystem manages direct interactions with the MongoDB database, abstracting the storage logic.

- **Persistence Service:**
  - `createDocument(collection, data)`: Inserts a new document (User, MediaItem, or List) into the specified MongoDB collection.
  - `findDocuments(collection, query)`: Retrieves documents matching specific criteria (e.g., all books belonging to `userId`).
  - `updateDocument(collection, id, updates)`: Modifies existing fields of a document identified by its ID.
  - `deleteDocument(collection, id)`: Permanently removes a document from the database.

5 Glossary of Terms

Term	Definition
Media Item	An entry in the user's library, such as a book or DVD.
Book	A media item with attributes like title, author, and page count.
DVD	A media item with attributes like title, duration, director.

Category	A classification of a media item (e.g., Book, DVD).
Tag	A user-defined label applied to one or more media items to support flexible organization.
User List	A custom list created by the user to group selected media items (e.g., “Favorites”).
Progress Tracking	Recording reading or viewing progress on a media item.
Progress Log	Historical list of progress updates.
Recommendation Engine	External AI service used to generate personalized media suggestions.
External Metadata Service	APIs such as Google Books or IMDb used to enrich item information.
Graceful Degradation	System behavior in which core features (CRUD) remain operational even when external APIs fail.
Filtering	Showing items that match selected criteria.
Sorting	Ordering items by a chosen attribute (title, date, etc.).
Search Query	Keywords or parameters entered to locate items in the library.
User Consent	Explicit approval required to send personal data to the AI recommendation service.
Authentication	Verifying user identity using OAuth 2.0.
MERN Stack	The core technology stack: **M**ongoDB, **E**xpress.js, **R**eact.js, and **N**ode.js.

Discriminator	A Mongoose feature used to handle polymorphism in the MongoDB schema, allowing `Book` and `Movie` to inherit from the base `MediaItem` schema.
JWT	JSON Web Token. The secure, stateless token issued by the server to the client after successful OAuth authentication for session management.
OAuth 2.0	The standard protocol used by the system to delegate user authentication to a third-party identity provider (e.g., Google).

6 Traceability

This section establishes traceability by linking the problems identified in the **Current System** (as defined in the SRS) to the **Requirements** designed to solve them, and finally, to the specific **Design Subsystems** responsible for implementation in the SDD. This ensures every major design decision directly addresses a project requirement.

Problem Statement (from SRS Section 2)	Key Requirement(s) Addressed	Implementing Design Subsystem(s) (from SDD Section 3.2)
Lack of Personal Focus	FR-1 (Media CRUD), FR-3 (Tag Management), NFR-US (Usability)	**User Interface (UI) Subsystem** (via React/MaterialUI) **Application Logic (API) Subsystem** (via authorization middleware)

No Progress Tracking	FR-6.1 / FR-6.2 (Track Progress), FR-6.4 (Visual Indicators)	**Application Logic (API) Subsystem** (for calculation and logging) **Data Access Subsystem** (for progress persistence)
Limited Media Support	FR-2.1 (Categorization), NFR-SUP-1 (Modularity)	**Data Access Subsystem** (via Mongoose Discriminators in 3.4) **Application Logic (API) Subsystem** (to process different media logic)
No Custom Lists or Tagging	FR-3 (Tag Management), FR-5 (List Operations)	**Application Logic (API) Subsystem** (for CRUD logic on lists/tags) **Data Access Subsystem** (for persisting schemas)
Absence of Content Enrichment	FR-8.1 / FR-8.2 (Metadata Fetch)	**External Integration Subsystem** (Metadata Connectors) **Application Logic (API) Subsystem** (to normalize and update data)
No Recommendation Features	FR-9 (AI-Based Recommendations), NFR-DATA-1 (User Consent)	**External Integration Subsystem** (AI Service Connector) **Application Logic (API) Subsystem** (to enforce consent check 3.5)

Fragmented User Experience	NFR-PERF-1 (Performance), NFR-REL-1 (Availability)	**User Interface (UI) Subsystem** (React SPA) **Application Logic (API) Subsystem** (Node.js concurrency) **Hardware/Software Mapping** (Docker/VPS deployment 3.3)
No unified filtering/searching	FR-4 (Search, Filter, Sort), FR-5.8/5.9 (Search in Lists)	**Application Logic (API) Subsystem** (Advanced Filter logic) **Data Access Subsystem** (optimized MongoDB queries)

The work completed so far can be found at the GitHub link below.
https://github.com/Personal-Library-Management-System/Personal-Lib-Man-System