ADDIS ABABA SCIENCE AND TECHNOLOGY UNIVERSITY

SOFTWARE IMPLEMENTATION DOCUMENTATION

FOR

AASTU LIBRARY MANAGEMENT SYSTEM UPDATE

ADDIS ABABA SCIENCE AND TECHNOLOGY UNIVERSITY
UNIVERSITY FOR INDUSTRY

Table of Contents

1. Introduction

1.1. Purpose

This document provides a comprehensive outline of the implementation details necessary to upgrade a library management system utilizing the powerful capabilities of Next.js, Go (Golang), and Docker. The primary objective of this upgrade is to integrate modern web technologies, enhance overall system performance, and utilize containerization to achieve scalability and simplify deployment processes. By implementing these enhancements, the library management system will deliver an improved user experience, better maintainability, and increased operational efficiency.

1.2. Scope

The enhanced system will include the following functionalities:

- Empower librarians with tools to manage book inventories and user accounts more effectively.
- Provide users with the ability to search for books, borrow them, and return them seamlessly via an online platform.
- Ensure data persistence and integrity by integrating with a robust database.
- Leverage Docker to enable containerized deployment, ensuring consistent environments across development and production.

The upgrade also aims to streamline workflows for librarians, enhance accessibility for users, and provide a scalable foundation for future system improvements.

1.3. Target Audience

This documentation is tailored for software developers, system administrators, project managers, and other technical stakeholders involved in the development, maintenance, and extension of the upgraded library management system.

2. System Overview

2.1. Architecture

The upgraded library management system follows a modular and scalable architecture, comprising the following key components:

Front-End: Next.js

- React framework for building dynamic user interfaces.
- Server-side rendering for better SEO and faster load times.
- API integration for communication with the back-end.

Back-End: Go (Golang)

- High-performance, efficient server-side language for handling API requests.
- RESTful API endpoints for user management, book management, and transactions.

Database: MongoDB

- NoSQL database for flexible storage of books, users, and transactions.
- Collection-based structure to store records such as books, users, transactions, and reservations.
- Aggregation framework for reporting and statistics.

Containerization: Docker

- Docker for creating containers to run the application in isolated environments.
- Simplifies deployment and scaling on cloud platforms.
- Supports easy local development and testing.

2.2. Component Interaction

- The Frontend communicates with the Backend via RESTful APIs to fetch and display data dynamically.
- The Backend interacts with the Database to execute queries, manage data, and enforce business rules.
- Docker containers encapsulate each component, isolating dependencies and ensuring that the system functions cohesively across diverse environments.

3. Installation and Setup

3.1. Prerequisites

Before proceeding with the installation, ensure the following tools and technologies are installed on your system:

- Node.js (version 16 or higher) for building and running the Next.js frontend.
- Go (version 1.19 or higher) for developing and running the backend API.
- Docker (version 20.10 or higher) for containerization.
- MongoDB for persistent storage of library data.

3.2. Installation Instructions

1. Clone the Repository:

Start by cloning the project repository to your local machine:

bash
```
git clone https://github.com/Kalkidan-Amare/AASTU-Library-Upgrade.git
cd AASTU-Library-Upgrade
```

2. Setup Environment Variables:

Create a `.env` file in the project root directory and populate it with the required environment variables:

bash
```
cd Api
```
env
```
MONGO_URI =
"mongodb+srv://user:password@cluster0.d2j0zkv.mongodb.net/?retryWrites=true&w=majority
&appName=Cluster0"
JWT_SECRET = "the_secret_key"
```
bash
```
cd Client
```
env
```
BACKEND_URL= http://localhost:8080
```

3. Build and Start Containers:

Use Docker Compose to build and start the application containers:

bash

```
    docker-compose up –build
```

Build and start Next.js on production:

```
bash
    npm run build
    npm start
```

4. Access the Application:

Once the containers are running, access the application via the following URLs:

- Frontend: `http://localhost:3000`

- Backend API: `http://localhost:8080`

By following these steps, the upgraded system will be operational and ready for testing and deployment.

4. Code Structure

4.1. File Organization

Frontend (Next.js)

The frontend project directory is organized as follows:

```
/frontend
├── pages
│   ├── index.js        # Homepage for users
│   ├── books.js        # Page for browsing and managing books
│   ├── login.js        # User authentication page
├── components          # Reusable React components
├── styles              # Global and component-specific styles
└── utils               # Helper functions and utilities
```

Backend (Go)

The backend project directory is structured as follows:

```
/backend
├── main.go             # Application entry point
├── handlers            # HTTP route handlers
├── models              # Database schema definitions and ORM models
├── middleware          # Middleware functions for authentication and validation
├── routes              # API route definitions
└── utils               # Common utility functions
```

Docker Setup

The project includes the following Docker configuration files:

```
/
├── docker-compose.yml   # Defines services, networks, and volumes
├── Dockerfile.frontend  # Dockerfile for building the Next.js frontend
├── Dockerfile.backend   # Dockerfile for building the Go backend
└── .env                 # Environment variable configurations
```

This structure ensures modularity, maintainability, and ease of navigation for developers.

4.2. Coding Standards

To maintain consistency and readability in the Library Management System upgrade, the following coding standards are established for both front-end (Next.js) and back-end (Go) components.

General Guidelines

- Follow consistent naming conventions (camelCase, PascalCase, kebab-case).
- Use meaningful names and keep functions small.
- Avoid code duplication and use reusable components.
- Use comments to explain complex logic.
- Format code using linters.
- Handle errors gracefully and log them.

Frontend (Next.js) Standards

- Use functional components with React Hooks.
- Maintain component-based architecture.
- Follow ESLint rules and use TypeScript.
- Ensure accessibility and proper state management.

Backend (Go) Standards

- Follow idiomatic Go conventions.

- Maintain a clean project structure.
- Use RESTful API principles and Go modules.
- Optimize database queries and write tests.

Documentation and Collaboration

- Provide concise documentation and README files.
- Use Git branching strategies and meaningful commit messages.
- Conduct code reviews and use pull requests.

Security Best Practices

- Use environment variables for sensitive data.
- Validate user inputs and use secure authentication.
- Regularly update dependencies.

5. Functionality

5.1. Main Features

1. Book Management

- Librarians can add, edit, delete, and list books in the system.
- Users can search for books by title, author, or category, making it easy to find desired resources.

2. User Management

- Users can register for accounts, log in, and manage their profiles.
- Role-based access control ensures that administrative features are accessible only to authorized personnel.

3. Borrow/Return System

- Users can borrow available books with specified due dates.
- The system tracks borrow history, sends notifications for overdue items, and manages returns.

4 Analytics and Statistics

- Admins can see statistics on students, books.

- Admins can see analytics on active time, occupancy rate, borrowed books amount spread over a time period.

5.2. APIs

POST Login

```
https://aastulibraryupdate.onrender.com/users/login-admin
```

Body raw {json}

```
{
   "email": "kal@gmail.com",
   "password": "12345678"
}
```

Response:

```
{
  "message": "User logged in successfully",
  "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoiNjc3Njg0ZTVjZWI3MzcyN2Q3
ZTY1MDY3IiwidXNlcm5hbWUiOiJrYWx0aGVyZWFsYWRtaW4iLCJyb2xlIjoiYWRtaW4iL
CJleHAiOjE3MzU5MDcxMTR9.2MkO-fWgUIdXGj_H05gxiKpWJTuHoVdXKpcJp_0T4yw"
}
```

POST Register Admin

```
https://aastulibraryupdate.onrender.com/users/register
```

Body raw {json}

```
{
  "username": "jah",
  "email": "kalkidanamare11a@gmail.com",
  "password": "12345678",
  "student_id": "ETS1111/14"
}
```

Response:

```
{
  "id": "000000000000000000000000",
  "username": "kal",
  "email": "kalkidan@gmail.com",
  "student_id": "ETS0884/14",
  "password":
"$2a$10$fcj59S7zkZTFxmLbOvlaLukW4xDG5rZkma169NImXTMGtXnsZ2jIG",
  "role": "user",
  "borrowed_books": null
}
```

Authenticate Student

POST Authenticate Student

```
https://aastulibraryupdate.onrender.com/users/login-admin
```

Body raw {json}

```
{
    "email": "kal@gmail.com",
    "password": "12345678"
}
```

Response:

```
{
  "message": "User logged in successfully",
  "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoiNjc3Njg0ZTVjZWI3MzcyN2Q3ZTY1MDY3IiwidXNlcm5hbWUiOiJrYWx0aGVyZWFsYWRtaW4iLCJyb2xlIjoiYWRtaW4iLCJleHAiOjE3MzU5MDcxMTR9.2MkO-fWgUIdXGj_H05gxiKpWJTuHoVdXKpcJp_0T4yw"
}
```

Register a User

POST Register

```
https://aastulibraryupdate.onrender.com/users/register
```

Body raw {json}

```
{
  "username": "jah",
  "email": "kalkidanamare11a@gmail.com",
  "password": "12345678",
  "student_id": "ETS1111/14"
}
```

Response:

```
{
  "id": "000000000000000000000000",
  "username": "kal",
  "email": "kalkidan@gmail.com",
  "student_id": "ETS0884/14",
  "password":
"$2a$10$fcj59S7zkZTFxmLbOvlaLukW4xDG5rZkma169NImXTMGtXnsZ2jIG",
  "role": "user",
  "borrowed_books": null
}
```

Verify OTP

POST Login

```
https://aastulibraryupdate.onrender.com/users/verify-otp
```

Body raw {json}

```
{
    "email": "kal@gmail.com",
    "password": "12345678"
}
```

Response:

```
{
  "message": "User logged in successfully",
  "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoiNjc3Njg0ZTVjZWI3MzcyN2Q3
ZTY1MDY3IiwidXNlcm5hbWUiOiJrYWx0aGVyZWFsYWRtaW4iLCJyb2xlIjoiYWRtaW4iL
CJleHAiOjE3MzU5MDcxMTR9.2MkO-fWgUIdXGj_H05gxiKpWJTuHoVdXKpcJp_0T4yw"
}
```

Login User

POST Login

```
https://aastulibraryupdate.onrender.com/users/verify-otp
```

Body raw {json}

```
{
   "email": "kal@gmail.com",
   "password": "12345678"
}
```

Response:

```
{
  "message": "User logged in successfully",
  "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoiNjc3Njg0ZTVjZWI3MzcyN2Q3
ZTY1MDY3IiwidXNlcm5hbWUiOiJrYWx0aGVyZWFsYWRtaW4iLCJyb2xlIjoiYWRtaW4iL
CJleHAiOjE3MzU5MDcxMTR9.2MkO-fWgUIdXGj_H05gxiKpWJTuHoVdXKpcJp_0T4yw"
}
```

- For all APIs and development refer the testing document.

6. Database Design

1. Books Collection

Stores information about books available in the library.

Collection Name: books

Schema:

```
{
   "_id": "ObjectId",       // Unique identifier for the book (MongoDB ObjectID)
   "title": "string",       // Title of the book
   "description": "string", // Description of the book
   "bar_code": "string",    // Unique barcode of the book
   "status": "boolean",     // Availability status (true = available, false = borrowed)
   "shelf_no": "string",    // Shelf number where the book is stored
   "author": "string"       // Author of the book
}
```

Relationships:

Linked to the borrowed_books field in the users collection.

2. Borrowed Books Collection

Tracks borrowing and return details for library books.

Collection Name: borrowed_books

Schema:

```
{
   "_id": "ObjectId",           // Unique identifier for the borrow record
   "borrow_time": "string",     // Timestamp when the book was borrowed
   "return_time": "string",     // Timestamp when the book was returned
   "book_id": "ObjectId",       // Reference to the book being borrowed (from `books`)
   "user_id": "ObjectId"        // Reference to the user borrowing the book (from `users`)
}
```

Relationships:

book_id references _id in the books collection.

user_id references _id in the users collection.

3. Check-Ins Collection

Logs user check-in and check-out activities in the library.

Collection Name: checkins

Schema:

```
{
   "_id": "string",         // Unique identifier for the check-in record
   "user_id": "string",     // Reference to the user checking in (from `users`)
   "student_id": "string",  // Unique student ID
   "checkin_at": "string",  // Timestamp of the check-in
   "checkout_at": "string"  // Timestamp of the check-out
}
```

Relationships:

user_id references _id in the users collection.

4. Users Collection

Stores details about users (students, librarians, or administrators) in the system.

Collection Name: users

Schema:

```
{
   "_id": "ObjectId",              // Unique identifier for the user (MongoDB ObjectID)
   "username": "string",            // Username of the user
   "email": "string",              // Email address of the user
   "student_id": "string",          // Student ID associated with the user
```

```
    "password": "string",              // Hashed password
    "role": "string",                  // Role (e.g., "student", "librarian", "admin")
    "borrowed_books": ["ObjectId"],    // Array of borrowed book IDs (from `books`)
    "is_checked_in": "boolean"         // Status of the user's check-in (true/false)
}
```

Relationships:

borrowed_books references _id in the books collection.

7. External Services

The Library Management System uses several external services and libraries to enhance functionality, performance, and security across the front-end and back-end components.

Backend Dependencies (Go):

- gin – Web framework for building HTTP services.
- mongodriver – MongoDB driver for Go.
- crypto – Provides encryption and security utilities.
- jwt – Library for JSON Web Token authentication.
- mail package – Used for sending email notifications.

Frontend Dependencies (Next.js):

- radix-ui – Accessible, high-quality UI components.
- tanstack/react-query – Data fetching and state synchronization.
- tanstack/react-table – Advanced table handling and display.
- zustand – Lightweight state management solution.
- zod – Schema validation for form inputs and API responses.
- react-barcode – Barcode generation for book inventory tracking.

Third-Party Services:

- Postman – For API testing and development.
- MongoDB Atlas – Cloud-hosted database service.

8. Troubleshooting and Debugging

8.1. Common Issues

1. Database Connection Fails

- Verify that the `DATABASE_URL` in the `.env` file points to a running database instance with the correct credentials.

2. Frontend Not Loading

- Check the Next.js logs for errors and confirm that the backend API is operational and accessible.

3. Docker Build Errors

- Ensure Docker is installed and running, and verify the syntax of the Dockerfiles.

4. Database Missing

- Make sure the database is locally running or connecting using MongoDB atlas.

8.2. Debugging Tips

- Run the nextjs frontend in debugging mode:

bash

```
npm run dev
```

- Inspect logs using Docker:

bash

```
docker logs <container_name>
```

- Set `LOG_LEVEL=debug` in the `.env` file to enable detailed logging for the Go backend.

9. Versioning and Release Notes

9.1. Changelog

Commit 51c1995 - Updating the user input

Commit 65936c8 - Updating the borrow logic

Commit 3f33177 - Add location and proper logic to handle the intervals

Commit 1740f8d - Correcting the getUserByStudentID API

Commit 3eeccfa - Adding the getUserByStudentID function

Commit 91bb92f – Updating the check-in logic

Commit 6538587 - Add dockerfile for containerization

Commit 0e485a0 - Add LoginAdmin and Checkstudent structure

10. Future Work and Roadmap

10.1. Planned Improvements

Improved Search and Reporting:

- Integrating search engines like Elasticsearch and expanding reporting capabilities for more detailed, customizable, and faster reports.

Enhanced Offline Mode:

- Implementing offline functionalities like local caching or mobile app support for users to check out books without an internet connection.

Advanced User Interface and UX:

- Developing more intuitive and visually appealing interfaces with advanced components, responsive designs, and real-time updates.

Role-Based Access Control (RBAC):

- Adding more granular role management and permissions to control user access more specifically.

External System Integration:

- Adding integration with third-party APIs or external databases to enhance features, such as syncing books from external catalogs or processing payments.

Scalability and Performance Tuning:

- Incorporating horizontal scaling, load balancing, and database partitioning to handle high-volume users and transactions.

10.2. Known Limitations

Real-Time Book Availability:

- The system may have limitations in providing real-time availability updates for books if there are delays in synchronization between transactions (e.g., books being checked in/out simultaneously in different branches or by multiple users).

Limited Reporting Capabilities:

- While basic reporting features (book status, user activity, overdue reports) are implemented, advanced reporting (e.g., in-depth analytics, custom queries, or visual charts) may be limited in the initial version.
- Generating complex reports with large datasets may lead to performance issues or slow query times.

Scalability Constraints for Very Large Libraries:

- While MongoDB is scalable, very large volumes of books or users (millions of records) might affect performance, especially if data is not optimized or partitioned correctly.
- High-traffic usage (many concurrent users or transactions) might require additional performance tuning, load balancing, and possibly horizontal scaling of both the application and database.

Limited Offline Functionality:

- The system will require an internet connection to function optimally, as it relies on server-side processing, API communication, and database access.
- Offline features (e.g., local storage, limited access to book data) could be explored but may not be fully supported in the initial deployment.

Complexity of Docker Setup:

- Docker containers provide portability and environment consistency, but setting up and configuring Docker for local development or deployment might require a steep learning curve for new developers or administrators.
- Integration with specific cloud platforms or third-party services might introduce deployment-specific limitations.

Limited Search Features for Large Datasets:

- Advanced search features like full-text search or fuzzy matching may be limited with the basic MongoDB setup.

- For large collections (e.g., thousands of books), search performance may degrade unless additional indexing or search engines (e.g., Elasticsearch) are integrated.

Role-Based Access Control (RBAC) Limitations:

- The system might not fully support granular role-based access control across all modules. Certain users (e.g., librarians) may have broad access to system features, potentially creating security concerns.
- Fine-grained permissions (e.g., permissions for specific books or user data) may require additional development effort.

Basic User Interface and Experience (UI/UX):

- While the system will have a clean and functional UI/UX, advanced user interface features such as drag-and-drop, real-time notifications, or advanced dashboards might be limited in the initial version.
- The initial user interface may be simple and focus on core features, with future improvements planned based on user feedback.

Lack of Integration with External Systems:

- The system may have limited integration with external systems (e.g., third-party databases for book catalogs, library services, or payment gateways for fines).
- Importing/exporting data from other systems (e.g., old library management systems) may require additional custom work.

Data Import and Migration Challenges:

- Migrating data from legacy systems into MongoDB may be a time-consuming process, especially if the data format is not compatible or requires extensive transformation.
- Data import tools may be limited or require manual intervention for complex data.

User Feedback and Feature Requests:

- The system may not include all possible features initially requested by users (e.g., support for specific book formats, integration with social media, etc.).
- Feature prioritization and additional user requests will be addressed in future updates, based on demand and resources.

Backup and Disaster Recovery:

- While MongoDB supports backups, implementing a fully automated disaster recovery plan (e.g., in case of server failure or data corruption) may not be built-in by default.
- Additional configuration and monitoring are needed to ensure data integrity and recovery.

11. Conclusion

In conclusion, the proposed upgrade to the Library Management System (LMS) leverages modern technologies such as **Next.js**, **Go**, **MongoDB**, and **Docker** to enhance scalability, performance, and user experience. The use of **Next.js** allows for a fast and SEO-friendly front-end, while **Go** ensures efficient back-end processing, and **MongoDB** offers flexible, scalable data storage. **Docker** simplifies deployment and environment consistency, ensuring smooth scaling and management.

Despite the improvements, certain limitations remain, such as performance issues with large datasets, lack of real-time updates, and offline functionality. These will be addressed in future versions, with potential integrations like **Elasticsearch** and enhanced reporting features.

The upgrade aims to improve library operations, making them more efficient for users, librarians, and administrators. The system's modularity and scalability ensure that it can adapt to future needs, while a user-friendly interface improves the overall library experience. With ongoing updates and improvements, this LMS upgrade will continue to meet the demands of modern libraries, ensuring its long-term effectiveness.

Appendices

Glossary of Terms

API (Application Programming Interface): A set of protocols and tools that allow different software applications to communicate with each other.

Containerization: A technology that packages applications and their dependencies together in isolated units, called containers, for consistent and scalable deployments.

Docker: A containerization platform that packages applications and their dependencies into portable containers. It ensures consistent environments for development, testing, and deployment.

Go (Golang): A statically typed, compiled programming language known for its simplicity, high performance, and efficient handling of concurrency, commonly used for building scalable backend systems.

JSON (JavaScript Object Notation): A lightweight data format used to exchange information between systems. It is widely used in web development for APIs and data storage.

Library Management System (LMS): A software application used by libraries to manage their operations, such as cataloging, circulation, user management, and tracking borrowed and returned materials.

MongoDB: A NoSQL database that stores data in a flexible, JSON-like format. It is designed for scalability and is commonly used in applications with large datasets or dynamic data structures.

Next.js: A React-based framework for building modern web applications. It supports server-side rendering, static site generation, and routing to enhance application performance and SEO.

Scalability: The ability of a system to handle increased loads or grow in capacity without compromising performance or reliability.

Server-Side Rendering (SSR): A web development technique where HTML is generated on the server and sent to the browser, improving page load times and search engine optimization.

Role-Based Access Control (RBAC): A security mechanism that restricts system access based on a user's role, ensuring that users can only access features relevant to their responsibilities.

Elasticsearch: A distributed search and analytics engine often used to enhance search performance in applications, especially with large datasets.

NoSQL: A type of database that allows for the storage and retrieval of data in formats other than traditional tabular relations. Examples include document-based, key-value, and graph databases.

Concurrent Programming: A programming paradigm that allows multiple computations to run simultaneously, improving the performance of tasks that can be executed in parallel.

Load Balancing: A technique used to distribute workloads across multiple servers or systems to optimize resource use, ensure high availability, and prevent overloading.

Offline Functionality: Features of a system that allow users to perform certain tasks without an active internet connection, with data synchronization when reconnected.

Tech Stack: The combination of technologies, frameworks, and tools used to develop a software application.

References

[1] Next.js Documentation, "Next.js Docs," 2025. [Online]. Available: https://nextjs.org/docs. [Accessed: Jan. 26, 2025].

[2] Go Documentation, "Go Programming Language," 2025. [Online]. Available: https://golang.org/doc/. [Accessed: Jan. 26, 2025].

[3] MongoDB Documentation, "MongoDB Docs," 2025. [Online]. Available: https://www.mongodb.com/docs/. [Accessed: Jan. 26, 2025].

[4] Docker Documentation, "Docker Docs," 2025. [Online]. Available: https://docs.docker.com/. [Accessed: Jan. 26, 2025].

[5] M. A. Khan, M. A. Qadeer, and J. A. Ansari, "An Integrated Library Management System for Book Search and Placement Tasks," in Proceedings of the 2010 IEEE 2nd International Advance Computing Conference (IACC), Patiala, India, 2010, pp. 301-305. [Online]. Available: https://ieeexplore.ieee.org/document/5432715. [Accessed: Jan. 26, 2025].