Mini Project Report

1. Title

Library Management System Using MERN Stack with CI/CD Integration

---

2. Abstract

The Library Management System is a full-stack web application designed to automate and digitize core library operations such as book catalog management, member registration, book issuance and return, and user authentication. The project demonstrates the integration of multiple modern web development technologies, including React, Tailwind CSS, Node.js, Express, MongoDB, and Mongoose.

The system has been designed and developed as a collection of ten key experiments (practicals) that together form a cohesive full-stack development workflow — from frontend interface design to backend API creation, testing, and deployment using DevOps tools. Each practical focuses on a critical skill in modern software engineering such as responsive UI design, state management, API creation, JWT authentication, WebSockets, and CI/CD automation.

---

3. Aim

To develop a **secure, scalable, and responsive Library Management System** using the MERN (MongoDB, Express, React, Node.js) stack and modern DevOps tools, demonstrating end-to-end full-stack development principles through practical experiments.

---

4. Objectives

The primary objective of this project is to design and develop a full-stack web application that demonstrates the integration of modern web technologies and DevOps practices. The project emphasizes building highly responsive and interactive user interfaces using Tailwind CSS and dynamic component behavior through React Hooks such as `useEffect`, `useContext`, and custom hooks for better state and lifecycle management.

On the backend, the system focuses on developing RESTful APIs using Node.js and Express, integrated with MongoDB via Mongoose to ensure efficient data storage and retrieval. Security and reliability are achieved by creating production-ready APIs with middleware for validation and implementing authentication and authorization using JSON Web Tokens (JWT).

The project also highlights the importance of testing and validation of APIs using Postman to ensure proper functionality and robustness. Additionally, it incorporates real-time communication through WebSockets to enable instant data synchronization between users and the server. For deployment, the project adopts modern DevOps principles by setting up CI/CD pipelines using GitHub Actions for automated testing and deployment on Render and Vercel, along with Docker containerization to achieve platform independence, scalability, and consistency across environments.

---

5. Theory

5.1 MERN Stack Overview

The MERN stack combines **MongoDB**, **Express**, **React**, and **Node.js** to form a complete JavaScript-based development environment.

- **MongoDB** — NoSQL database that stores data as documents (BSON), allowing flexible schemas.
- **Express.js** — Lightweight backend framework for building REST APIs in Node.js.
- **React.js** — Frontend JavaScript library for building interactive user interfaces.
- **Node.js** — JavaScript runtime for building scalable, event-driven server applications.

This stack supports rapid development and easy code sharing between frontend and backend since both use JavaScript.

5.2 Tailwind CSS

Tailwind CSS is a utility-first CSS framework that enables developers to create responsive designs directly within HTML/JSX using prebuilt utility classes (e.g., `p-4`, `bg-gray-100`, `text-center`).
Unlike traditional frameworks like Bootstrap, Tailwind focuses on **custom design systems** rather than predefined components.
In this project, Tailwind CSS is used to build a **responsive, mobile-first layout** for library pages — including book catalogs, dashboards, and forms.

5.3 React Hooks and Context API

React Hooks simplify component logic and state management:

- `useState()` — manages component-level state.
- `useEffect()` — handles side effects like API calls or subscriptions.
- `useContext()` — allows sharing data across components without prop drilling.
- Custom Hooks — reusable logic blocks for data fetching, form validation, etc.

The **Context API** acts as a centralized state container for authentication, UI state, and session management — replacing Redux in this project.

5.4 RESTful APIs

Representational State Transfer (REST) defines architectural constraints for designing APIs that communicate using HTTP.
A REST API exposes resources through endpoints such as `/api/books`, `/api/users`, `/api/transactions`.

Using **Express**, each route is modularized, and controller functions handle CRUD operations.
Mongoose is used to define schemas and interact with MongoDB.

Example model:

```
const BookSchema = new mongoose.Schema({
  title: String,
  author: String,
  category: String,
  totalCopies: Number,
  availableCopies: Number,
});
```

5.5 Authentication and Security (JWT)

User authentication is implemented using **JSON Web Tokens (JWT)**.
Upon successful login, a signed token is issued to the client, which is used for subsequent API calls.
Admin-only routes are protected by authorization middleware that verifies token validity and user role.

Security measures include:

- Password hashing with **bcrypt**
- Token expiration and revocation
- Secure `.env` configuration
- Input validation with **express-validator**

5.6 WebSockets and Real-Time Communication

The project integrates **WebSockets** to allow real-time updates (e.g., notifying users when a book is returned or becomes available).
This is implemented using the **Socket.IO** library on both client and server sides.

5.7 CI/CD Pipeline (GitHub Actions + Render/Vercel)

Continuous Integration (CI) automatically tests and builds the code after each commit.
Continuous Deployment (CD) pushes the latest stable build to production.
GitHub Actions workflow:

- Lint and test code on every push
- Build frontend with `npm run build`
- Deploy backend to Render and frontend to Vercel

This ensures consistent and automated deployment, reducing manual errors.

5.8 Docker and DevOps

Docker enables packaging the entire application (frontend, backend, and database) into lightweight containers.
Each service runs in its own container:

- `frontend` container — React app served with Nginx
- `backend` container — Node.js + Express server
- `mongo` container — MongoDB database

A `docker-compose.yml` file orchestrates the containers and network. This setup simplifies deployment and guarantees consistency across environments.

---

6. Implementation

6.1 Project Structure

```
library-system/
├── backend/
│   ├── models/
│   ├── routes/
│   ├── middleware/
│   ├── server.js
│   ├── seed.js
│   ├── .env
│   └── package.json
├── src/
│   ├── components/
│   ├── contexts/
│   ├── pages/
│   ├── utils/
│   └── App.js
├── public/
│   ├── index.html
│   ├── manifest.json
│   ├── robots.txt
│   └── images/
├── tailwind.config.js
├── package.json
└── README.md
```

6.2 Backend Workflow

- **Server.js** initializes Express, connects MongoDB using Mongoose, and sets up routes.
- **Routes/** folder includes route files such as `books.js`, `users.js`, `auth.js`.
- **Middleware/** includes:
    - `auth.js` — verifies JWT token
    - `roleCheck.js` — ensures admin-only access
- **Models/** defines data schemas for User, Book, Transaction.

6.3 Frontend Workflow

- **Components** handle UI elements (Navbar, BookCard, Footer, Modal).
- **Pages** define screens (Login, Register, Dashboard, Book Catalog).
- **Contexts** manage authentication and global state.
- **Hooks** are used to fetch data and manage user sessions.
- **Tailwind** utilities ensure responsiveness across devices.

6.4 Database Integration

MongoDB stores structured documents for users, books, and transactions.
Relationships are managed using **object references** (`ref`).
`seed.js` populates sample data for demo and testing.

6.5 Authentication Flow

1. User logs in → credentials verified.
2. Server signs a JWT with user role.
3. Token stored in localStorage or cookies.
4. Protected routes validate token via middleware.
5. Role-based access: Admin vs User functionalities.

6.6 API Validation with Postman

- All API routes are tested using **Postman**.
- Collections include:
    - User Registration & Login
    - Book CRUD
    - Transactions (Issue/Return)

- Response codes verified:
    - `200 OK` for success
    - `400 Bad Request` for validation errors
    - `401 Unauthorized` for invalid tokens

6.7 WebSockets Implementation

The system uses **Socket.IO** to notify all clients when:

- A book is issued or returned.
- Inventory count changes.

6.8 CI/CD and Deployment

- **GitHub Actions** automates testing, building, and deployment.
- **Frontend** deployed to **Vercel** (auto-build from main branch).
- **Backend** deployed to **Render** with MongoDB Atlas as database.
- **Docker** containers can replicate the environment locally or in the cloud.

---

7. Results and Output

- The system provides a **responsive UI** accessible on desktops, tablets, and mobiles.
- Books can be added, edited, and issued with real-time updates.
- Role-based dashboard (Admin/User) works correctly.
- WebSocket notifications update the catalog dynamically.
- CI/CD ensures automatic deployment after every Git commit.

---

8. Challenges Faced

- Managing synchronization between frontend and backend during token-based authentication.
- Handling CORS policies between React (port 3000) and Express (port 5000).
- Implementing real-time updates while maintaining API performance.
- Configuring environment variables securely in deployment environments.

---

9. Future Enhancements

1. Integrate **Email/SMS notifications** for overdue books.
2. Add **analytics dashboard** for library usage statistics.
3. Implement **multi-language support** for accessibility.
4. Integrate **AI-based recommendations** for book suggestions.
5. Add **QR code scanning** for faster issue/return operations.

---

10. Conclusion

The Library Management System successfully demonstrates the complete development lifecycle of a full-stack application using the MERN stack and modern DevOps practices. It covers essential concepts from frontend design to backend security, API validation, real-time data handling, and cloud deployment.

By integrating all ten practicals — from Tailwind CSS UIs to Docker deployment — this project reflects an industry-ready application pipeline. It not only automates library operations but also serves as a comprehensive demonstration of full-stack and DevOps engineering.

---

11. References

1. React Official Documentation
2. Tailwind CSS Documentation
3. Express.js Guide
4. MongoDB & Mongoose Docs
5. Socket.IO
6. GitHub Actions Docs
7. Docker Documentation