# TASK 13: Implement File Upload Feature for Book Cover Images

**Name**: Prathika

**College**: Sahyadri College of Engineering and Management

**Role**: React Developer

**Task Number**: 13

## 1.Introduction & Project Goals:

This document provides a comprehensive overview of the Book Management Application, a full-stack MERN (MongoDB, Express.js, React, Node.js) project. The primary goal of this application is to demonstrate proficiency in modern web development practices by building a robust, scalable, and user-friendly platform for managing a personal book collection.

The project emphasizes a decoupled architecture, seamless integration with third-party cloud services for file storage, centralized state management, and an automated deployment pipeline. It serves as a practical showcase of building and deploying a complete, production-ready web application from the ground up.

## 2.Live Application Links:

https://book-management-r8v4.onrender.com

## 3.Core Features:

- **Add Books:** Create new book entries with title, author, and a cover image.

- **View Collection:** Display all books in a responsive grid or list format.
- **Cloud Image Uploads:** Securely upload and serve cover images via the Cloudinary platform.
- **Rich UI Feedback:** Provide a seamless user experience with toast notifications, a real-time upload progress bar, and celebratory confetti animations.
- **Live Search:** Filter the book collection by title or author with a debounced search input for optimal performance.

# 4.System Architecture:

The application is architected with a clean separation of concerns between the frontend client and the backend server.

## 4.1 Frontend Architecture (React):

The frontend is a modern single-page application built with **React** and bootstrapped with **Vite** for a fast development experience.

- **Component-Based UI:** The user interface is composed of modular components, primarily BookForm (for user input) and BookList (for data display), orchestrated by the main App.jsx component.
- **Centralized State Management (Context API):** To avoid prop-drilling and maintain a clean data flow, the application leverages React's Context API.
- **BookContext**: A "smart" provider that encapsulates all business logic. It manages the state for the book collection, loading status, and upload progress. It also contains all async functions for interacting with the backend API.
- **ToastContext**: A dedicated provider for managing and displaying global UI notifications (toasts), keeping this cross-cutting concern separate from the main application logic.

### 4.2. Backend Architecture (Node.js/Express):

The backend is a RESTful API built with **Node.js** and the **Express.js** framework. It follows a modular, MVC-inspired pattern to ensure maintainability and separation of concerns.

- **Models:** The models/Book.js file defines the Mongoose schema, which dictates the structure of the book documents in the MongoDB database, including data types and validation rules.
- **Routes:** The routes/bookRoutes.js file defines the API endpoints (e.g., GET /api/books, POST /api/books) and maps them to the appropriate controller functions. It also attaches necessary middleware.
- **Controllers:** The controllers/bookController.js file contains the core business logic. Each function handles the request-response cycle, interacts with the Mongoose model to perform database operations (CRUD), and communicates with external services like Cloudinary.
- **Middleware:** The middleware/uploadMiddleware.js file configures multer to handle multipart/form-data, which is essential for processing file uploads. It includes validation for file types and size limits.

## 5. In-Depth Implementation Details:

### 5.1 The Image Upload Workflow:

The image upload process is a key feature and is designed to be secure and efficient by offloading storage to a dedicated cloud service.

1. Client-Side: The user selects a file via the <input type="file"> in the BookForm component. A client-side preview is immediately generated using URL.createObjectURL().
2. Request: On form submission, Axios sends a multipart/form-data request to the backend. The onUploadProgress event listener in the Axios config is used to track the upload progress.

3. Server-Side (Middleware): The multer middleware on the backend intercepts the request. It validates that the file is an accepted image type (jpeg, jpg, png) and within the size limit (2MB). It then temporarily stores the file for processing.
4. Cloud Upload: The createBook controller function takes the path of the temporary file and uses the Cloudinary Node.js SDK to upload the file to the Cloudinary cloud. The upload includes transformations to scale the image to a width of 400px.
5. Database Storage: Cloudinary returns a secure URL for the uploaded and transformed image. The controller saves this URL string (not the file itself) to the coverImage field in the new MongoDB document.
6. Response: The backend responds to the frontend with the newly created book object, including the Cloudinary image URL. The frontend then updates its state, and the new book appears in the collection.

### 5.2. Global State Management with Context API:

The React Context API was chosen as the state management solution for its simplicity and built-in nature, which is perfectly suited for the application's scale. It allows the BookForm and BookList components to share and manipulate application state without passing props down through multiple levels of the component tree. This makes the components more decoupled and easier to maintain.

## 6. Security Measures:

- **CORS Policy**: The backend server implements a strict Cross-Origin Resource Sharing (CORS) policy, ensuring that it only accepts requests from the deployed frontend URL and specified localhost ports.

- **Environment Variables**: All sensitive information—including the database connection string, Cloudinary credentials, and the frontend URL—is managed through environment variables (.env files) and is never hardcoded in the source code.
- **Server-Side File Validation**: In addition to client-side checks, the multer middleware on the server performs validation to reject files that are not of the allowed image types or exceed the size limit.

## 7. Testing Strategy:

The backend API is tested at the integration level using Jest as the test runner and Supertest for making HTTP requests to the Express app. This approach tests the entire request-response cycle, including middleware, routing, and controller logic.

The test suite (/backend/tests/book.test.js) covers:

- The "happy path" for creating a new book with a valid image file.
- The failure case where a user attempts to upload an invalid file type (e.g., a .txt file), ensuring the server rejects it as expected.

## 8. Deployment

The application is deployed on Render using a Continuous Deployment (CI/CD) pipeline linked to the main branch of the GitHub repository. Any push to the main branch automatically triggers a new build and deployment.

- Backend (Web Service): The Node.js application is deployed as a Render Web Service. All necessary environment variables are configured in the Render dashboard.
- Frontend (Static Site): The React application is deployed as a Render Static Site. It is configured to handle the mono repo structure by setting the Root Directory to frontend, ensuring that build commands run in the correct context.