Library Management System

Name: Movindu Perera

Submission Date: 16-12-2024

Table of Contents

If you encounter an error while running the library-management-frontend, please run the command npm install within the library-management-frontend directory. This will install the required dependencies and resolve the issue.

Thank you!

Introduction

The detailed report on Library Management System (LMS), developed as a RESTful API with a React-based frontend, is given. This project will basically manage books within a library; it supports basic CRUD operations for book data: Create, Read, Update, Delete. It's an ASP.NET Core application on the back end, and React.js on the front end, utilizing SQLite for the database.

Tools & Technologies Used

Frontend: React, TypeScript

Backend: ASP.NET Core Web API

Database: SQL Server

Development Tools: VS Code, Postman

Development Process

Backend Implementation

The backend is built using ASP.NET Core Web API.

Controllers:

BooksController: handles HTTP requests (GET, POST, PUT, DELETE) for book management.

```
BooksController.cs ×
LibraryManagementAPI > Controllers > BooksController.cs > BooksController > CreateBook
13      public class BooksController : ControllerBase
60          public async Task<IActionResult> UpdateBook(int id, Book book)
                    throw;
                }
            }

            return CreatedAtAction(nameof(GetBook), new { id = book.Id }, book);
        }

        // Delete a book by ID
        [HttpDelete("{id}")]
        0 references
        public async Task<IActionResult> DeleteBook(int id)
        {
            var book = await _context.Books.FindAsync(id);

            if (book == null)
            {
                return NotFound();  // Return 404 if the book is not found
            }

            _context.Books.Remove(book);
            await _context.SaveChangesAsync();

            return NoContent();  // Return 204 (No Content) on successful deletion
        }

        1 reference
        private bool BookExists(int id)
        {
            return _context.Books.Any(e => e.Id == id);
        }
    }
```

Database: SQLite was used to store book details such as Title, Author, and Description.

Endpoints:

GET /api/books - Fetch all books

POST /api/books - Add a new book

PUT /api/books/{id} - Update book details

DELETE /api/books/{id} - Delete a book

Frontend Implementation

The frontend is developed using React and TypeScript.

Key Components:

api/api.ts: Handles API calls to the backend.

```
TS api.ts  U ×
library-management-frontend > src > api > TS api.ts > API_BASE_URL
import axios from 'axios';

const API_BASE_URL = 'http://localhost:5100/api';

export const getBooks = async () => {
  const response = await axios.get(`${API_BASE_URL}/books`);
  return response.data;
};

export const createBook = async (book: { title: string; author: string; description: string }) => {
  try {
    const response = await axios.post(`${API_BASE_URL}/books`, book, {
      headers: {
        'Content-Type': 'application/json',
      },
    });
    return response.data;
  } catch (error) {
    console.error('Error in createBook API:', error);
    throw error;
  }
};

// API call to update a book
export const updateBook = async (id: string, book: { title: string; author: string; description: string }) => {
  try {
    const response = await axios.put(`${API_BASE_URL}/books/${id}`, book);
    return response.data;
  } catch (error) {
    console.error('Error in updateBook API:', error);
    throw error;
  }
```

components/: Includes reusable components for Create, View, Update, and Delete books.

API Integration: Fetches data using axios from the backend.

```
export const getBooks = async () => {
  const response = await axios.get(`${API_BASE_URL}/books`);
  return response.data;
};
```

Challenges Faced

Backend:

- Port conflicts while running the API locally.
- Database connection issues (SQL Server configuration).

Frontend:

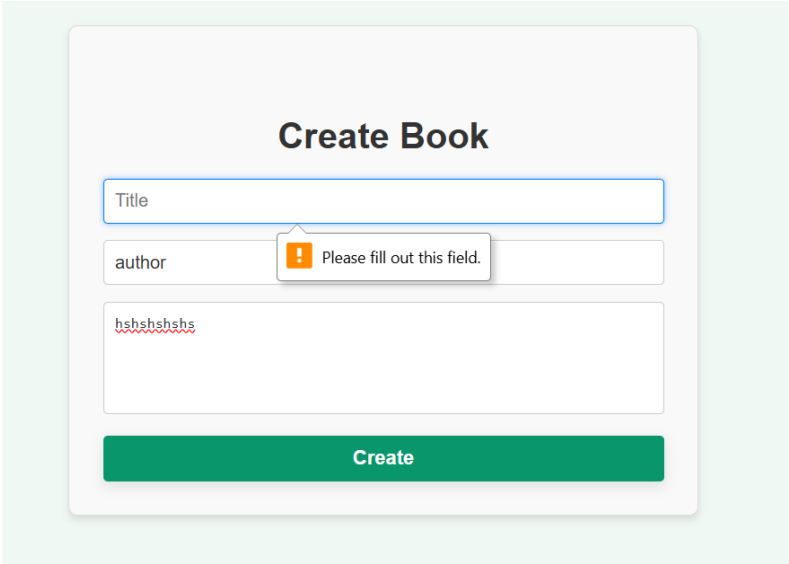
- Integration of API with React.
- Handling state management for dynamic content.

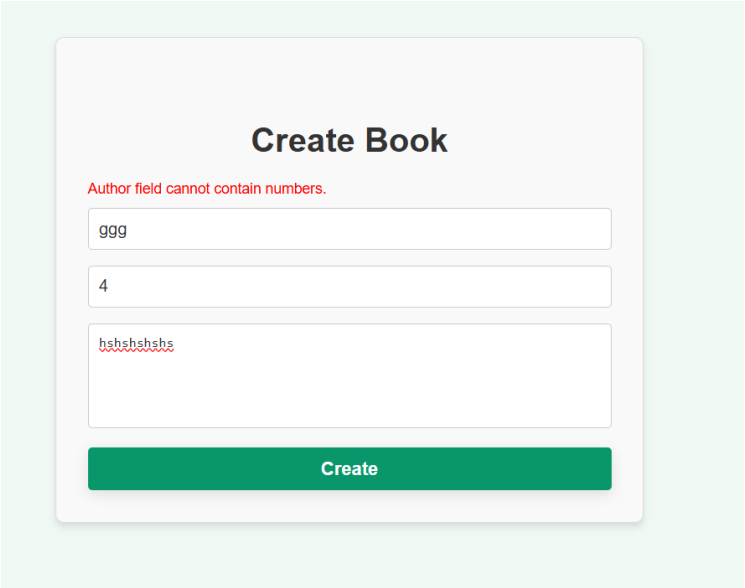
Testing

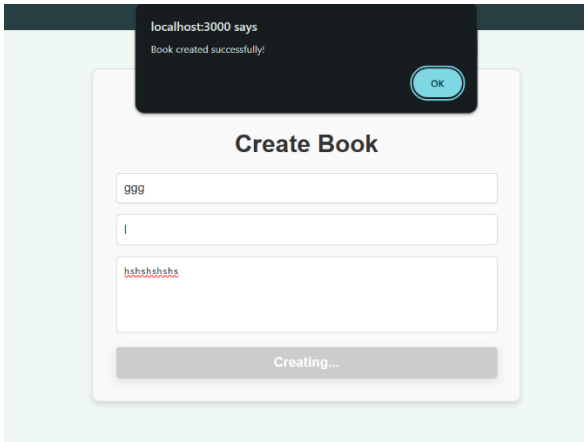
Test Plan

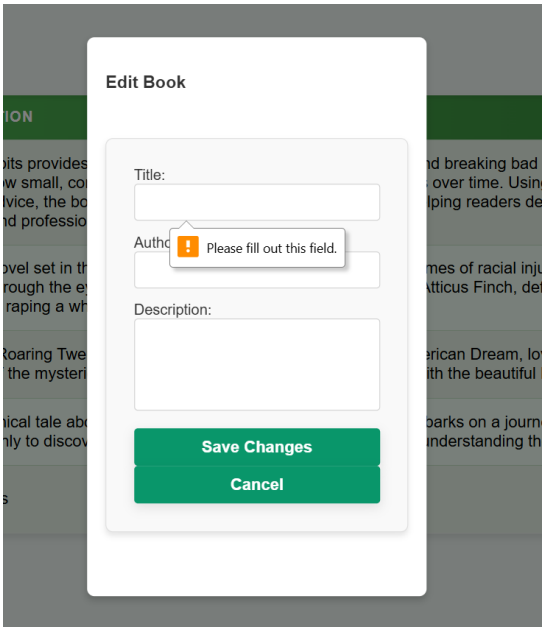
Test Case ID	Test Case Name	Scenario	Expected Results
TC1	Add a book with missing value.	Ensure that new book record cannot be created without filling out all the fields.	Please fill out this field.
TC2	Contain a number in author field.	Ensure that an author could not have a number.	Author field cannot contain numbers.
TC3	Update book details with valid data.	Ensure that book records can be edited.	Book details are updated successfully.
TC4	Update a book with an empty field.	Ensure that book recorded cannot be updated with empty fields.	Please fill out this field.
TC5	Delete an existing book.	Ensure that book record can be deleted.	Book is deleted successfully.

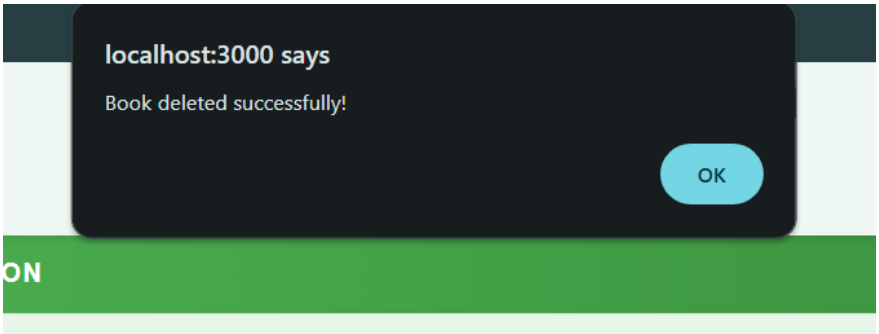
Test Cases

Test Case ID	TC1
Test Case Name	Add a book with missing value
Test Data	Author, Description
Expected Result	Please fill out this field.
Actual Result	
Status	Pass

Test Case ID	TC2
Test Case Name	Contain a number in author field.
Test Data	Author: “Author”
Expected Result	Author field cannot contain numbers.
Actual Result	
Status	Pass

Test Case ID	TC3
Test Case Name	Update book details with valid data.
Test Data	Title, Author, Description
Expected Result	Book details are updated successfully.
Actual Result	
Status	Pass

Test Case ID	TC4
Test Case Name	Update a book with an empty field.
Test Data	Title, Description
Expected Result	Please fill out this field.
Actual Result	
Status	Pass

Test Case ID	TC5
Test Case Name	Delete an existing book
Test Data	-
Expected Result	Book is deleted successfully.
Actual Result	
Status	Pass

Additional Features

- Error Handling: Relevant messages are displayed to the user if a book cannot be added or updated.
- Response Messages: Success and error messages are displayed dynamically on the frontend.
- Form validation: Ensures no field is left empty.

Key Insights

The development of a REST API and its integration with React improved my understanding of full-stack development.

The use of React's useState hook for handling input and API data was a big learning curve.

Postman has been extremely useful in debugging API issues.

Proper backend validation and error responses are critical to ensuring a seamless user experience.

Conclusion

The Library Management System was developed successfully as a functional solution to manage book data efficiently. Key features include seamless CRUD operations: Create, Read, Update, Delete, providing full control over the records of books.

The project will ensure smooth integration between the backend, built using ASP.NET Core Web API and the frontend, which is developed using React and TypeScript, into a cohesive and responsive system.