Datence Technologies

Technical Documentation

Book Explorer Application

A Full-Stack Web Application for Book Browsing and Management

Prepared by:
Aaryan Choudhary

September 2025

Contents

Chapter 1

Introduction

1.1 Project Overview

Book Explorer is a comprehensive full-stack web application that provides an online platform for browsing and exploring books. The application scrapes book data from an external source, stores it in a database, exposes RESTful APIs, and presents the books through a modern frontend interface.

1.2 Purpose and Scope

The primary purpose of this application is to allow users to:

- Browse a comprehensive collection of books
- Search for specific titles using advanced filters
- View detailed information about each book
- Save favorite books for future reference
- Add books to a shopping cart

1.3 Target Audience

- Book enthusiasts and readers
- Online bookstore customers
- Library patrons
- Educational institutions
- Book retailers

Chapter 2

System Architecture

2.1 High-Level Architecture

The Book Explorer application follows a three-tier architecture:

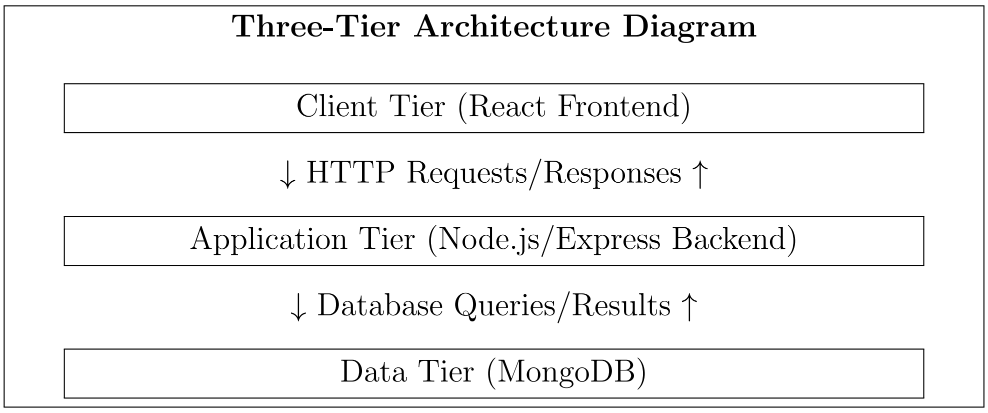
Three-Tier Architecture Diagram
Client Tier (React Frontend)
↓ HTTP Requests/Responses ↑
Application Tier (Node.js/Express Backend)
↓ Database Queries/Results ↑
Data Tier (MongoDB)

Figure 2.1: High-Level Architecture of the Book Explorer Application

2.2 Component Diagram

The application consists of three main components:

1. **Data Scraper**: A Node.js application that extracts book data from the external site "Books to Scrape" and populates the MongoDB database.
2. **Backend Service**: A RESTful API service built with Node.js and Express.js that provides endpoints for retrieving, filtering, and searching book data.
3. **Frontend Application**: A React application built with Vite that provides the user interface for browsing, searching, and interacting with the book data.

2.3 Deployment Architecture

For local development, the application uses:

- Frontend: `http://localhost:5173`

- Backend: `http://localhost:5000`
- Database: Local MongoDB instance at `mongodb://localhost:27017/book-explorer`

For production deployment, the application uses:

- Frontend: Netlify (Static Site Hosting)
- Backend: Render (Node.js Web Service)
- Database: MongoDB Atlas (Cloud Database)

Chapter 3

Technical Stack

3.1 Frontend Technologies

- **React 18.2**: JavaScript library for building user interfaces
- **Vite 5.0.0**: Frontend build tool for faster development
- **React Router 6.21**: Library for handling navigation
- **React Icons 4.12**: Icon library for UI components
- **Axios 1.6.2**: Promise-based HTTP client for API requests
- **CSS3**: Custom styling with responsive design
- **HTML5**: Structure and semantics

3.2 Backend Technologies

- **Node.js 20.x**: JavaScript runtime environment
- **Express 4.18.2**: Web application framework
- **MongoDB 6.0**: NoSQL database for storing book data
- **Mongoose 8.0.1**: MongoDB object modeling for Node.js
- **CORS 2.8.5**: Cross-Origin Resource Sharing middleware
- **dotenv 16.3.1**: Environment variable management
- **nodemon 3.0.1**: Development utility for auto-restarting server

3.3 Data Scraping Tools

- **Puppeteer 21.5.2**: Headless Chrome Node API for web scraping
- **cheerio 1.0.0-rc.12**: Fast, flexible HTML parser

3.4 Development Tools

- **ESLint**: JavaScript linting utility
- **Git**: Version control system
- **npm**: Package manager for JavaScript
- **Visual Studio Code**: Integrated development environment
- **Windows Batch Files**: Automation scripts for local development

Chapter 4

Component Breakdown

4.1 Data Scraper

The data scraper is responsible for extracting book data from the external website and populating the MongoDB database.

4.1.1 Key Files

- `scraper/scraper.js`: Main scraper script
- `scraper/package.json`: Dependencies and scripts

4.1.2 Functionality

- Navigates through all pages of the ”Books to Scrape” website
- Extracts book details: title, price, stock availability, rating, detail URL, and thumbnail
- Stores extracted data in the MongoDB database
- Handles pagination and error recovery
- Creates MongoDB text indices for efficient searching

4.1.3 Implementation Details

The scraper uses Puppeteer to launch a headless Chrome browser, navigate to the book website, and extract data using DOM traversal.

```
const puppeteer = require("puppeteer");
const mongoose = require("mongoose");

// MongoDB connection
mongoose.connect("mongodb://localhost:27017/book-explorer");

// Book model schema
const bookSchema = new mongoose.Schema({
    title: String,
    price: Number,
    inStock: Boolean,
    rating: Number,
    detailUrl: String,
    thumbnailUrl: String,
});

// Create text index for searching
bookSchema.index({ title: "text" });
const Book = mongoose.model("Book", bookSchema);

async function scrapeBooks() {
    const browser = await puppeteer.launch();
    const page = await browser.newPage();

    // Navigate to website and extract data
    await page.goto("https://books.toscrape.com/");

    // Extract book data
    const books = await page.evaluate(() => {
        // DOM extraction logic
    });

    // Save to database
    await Book.insertMany(books);

    await browser.close();
}

scrapeBooks();
```
Listing 4.1: Scraper Core Logic

4.2 Backend Service

The backend provides a RESTful API for the frontend to interact with the book data stored in MongoDB.

4.2.1 Key Files

- `backend/server.js`: Entry point for the Express server
- `backend/routes/books.js`: API route definitions for book operations
- `backend/models/Book.js`: Mongoose schema for book data
- `backend/package.json`: Dependencies and scripts

4.2.2 API Endpoints

- `GET /api/books`: Retrieves a paginated list of books with filtering options
- `GET /api/books/:id`: Retrieves detailed information for a specific book

4.2.3 Implementation Details

The backend uses Express.js to define routes and Mongoose to interact with MongoDB.

```
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");

const app = express();
const PORT = process.env.PORT || 5000;

// Middleware
app.use(cors());
app.use(express.json());

// MongoDB connection
mongoose.connect(process.env.MONGODB_URI || "mongodb://localhost:27017/
    book-explorer");

// Book model
const Book = require("./models/Book");

// Routes
app.get("/api/books", async (req, res) => {
    const { page = 1, limit = 12, rating, inStock, search } = req.query
    ;

    // Build query based on filters
    const query = {};

    if (rating) query.rating = { $gte: parseInt(rating) };
    if (inStock === "true") query.inStock = true;

    if (search) {
        query.$or = [
            { $text: { $search: search } },
            { title: { $regex: search, $options: "i" } }
        ];
    }

    // Execute query with pagination
    const books = await Book.find(query)
        .limit(limit)
        .skip((page - 1) * limit)
        .exec();

    // Get total count
    const count = await Book.countDocuments(query);

    res.json({
        books,
        totalPages: Math.ceil(count / limit),
        currentPage: page,
        totalBooks: count
    });
});

app.get("/api/books/:id", async (req, res) => {
    try {
        const book = await Book.findById(req.params.id);
        if (!book) return res.status(404).json({ message: "Book not
    found" });
        res.json(book);
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
});

app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

Listing 4.2: Backend API Implementation

4.3 Frontend Application

The frontend provides the user interface for browsing, searching, and interacting with books.

4.3.1 Key Files and Directories

- `frontend/src/App.jsx`: Main application component with routing
- `frontend/src/components/`: UI components
  - `Navbar.jsx`: Application navigation bar
  - `Home.jsx`: Main page with book listing and filters
  - `BookCard.jsx`: Individual book display component
  - `Pagination.jsx`: Page navigation component
  - `Cart.jsx`: Shopping cart management
  - `Favorites.jsx`: Saved favorites management
  - `Footer.jsx`: Application footer
- `frontend/src/context/AppContext.jsx`: Global state management
- `frontend/src/config/api.js`: API endpoint configuration

4.3.2 Key Features

- Responsive design for mobile and desktop
- Book grid display with pagination
- Advanced search and filtering options
- Shopping cart management
- Favorites collection
- Professional UI with Modern Library theme palette

4.3.3 Implementation Details

The frontend uses React with context API for state management and React Router for navigation.

```
const API_BASE_URL = import.meta.env.VITE_API_URL || 'http://localhost
    :5000';

export const API_ENDPOINTS = {
  books: `${API_BASE_URL}/api/books`,
  book: (id) => `${API_BASE_URL}/api/books/${id}`,
};

export default API_BASE_URL;
```

Listing 4.3: Frontend API Configuration

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';
import { API_ENDPOINTS } from '../config/api';
import BookCard from './BookCard';
import Pagination from './Pagination';
import './Home.css';

const Home = () => {
  const [books, setBooks] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
  const [filters, setFilters] = useState({
    page: 1,
    rating: '',
    inStock: false,
    search: '',
  });
  const [totalPages, setTotalPages] = useState(0);
  const [totalBooks, setTotalBooks] = useState(0);

  useEffect(() => {
    fetchBooks();
  }, [filters]);

  const fetchBooks = async () => {
    try {
      setLoading(true);
      const params = new URLSearchParams();

      for (const key in filters) {
        if (filters[key]) {
          params.append(key, filters[key]);
        }
      }

      const response = await axios.get(`${API_ENDPOINTS.books}?${params
    }`);
      setBooks(response.data.books);
      setTotalPages(response.data.totalPages);
      setTotalBooks(response.data.totalBooks);
      setError(null);
    } catch (err) {
      setError('Failed to load books. Please try again later.');
    } finally {
      setLoading(false);
    }
  };

  const handlePageChange = (page) => {
    setFilters(prev => ({ ...prev, page }));
  };

  return (
    <div className="home-container">
      {/* Search and filter components */}

      {loading ? (
        <div className="loading">Loading books...</div>
      ) : error ? (
        <div className="error">{error}</div>
      ) : (
        <>
          <div className="books-grid">
            {books.map(book => (
              <BookCard key={book._id} book={book} />
            ))}
          </div>

          <Pagination
            currentPage={filters.page}
            totalPages={totalPages}
            onPageChange={handlePageChange}
          />
        </>
      )}
    </div>
  );
};

export default Home;
```
Listing 4.4: React Component Example

Chapter 5

Database Schema

5.1 MongoDB Collections

5.1.1 Books Collection

```javascript
const bookSchema = new mongoose.Schema({
  title: {
    type: String,
    required: true,
    index: true
  },
  price: {
    type: Number,
    required: true
  },
  inStock: {
    type: Boolean,
    default: true
  },
  rating: {
    type: Number,
    min: 0,
    max: 5,
    default: 0
  },
  detailUrl: {
    type: String,
    required: true
  },
  thumbnailUrl: {
    type: String,
    required: true
  },
  createdAt: {
    type: Date,
    default: Date.now
  }
});

// Text index for search functionality
bookSchema.index({ title: 'text' });
```

Listing 5.1: Book Schema

5.2 Indexing Strategy

The application uses the following MongoDB indices to optimize query performance:

- **Text index** on book title for efficient text search queries
- **Single field index** on rating for filter performance
- **Single field index** on inStock for availability filtering

Chapter 6

API Documentation

6.1 Book API Endpoints

6.1.1 Get Books List

- **Endpoint**: `GET /api/books`
- **Description**: Retrieves a paginated list of books with filtering options
- **Query Parameters**:
  - `page` (optional): Page number (default: 1)
  - `limit` (optional): Number of books per page (default: 12)
  - `rating` (optional): Minimum rating filter (0-5)
  - `inStock` (optional): Filter by availability (true/false)
  - `search` (optional): Search term for book titles
- **Response Format**:

```
{
  "books": [
    {
      "_id": "6578a1b2c3d4e5f6a7b8c9d0",
      "title": "The Requiem Red",
      "price": 22.65,
      "inStock": true,
      "rating": 4,
      "detailUrl": "https://books.toscrape.com/catalogue/the-requiem-red_995/index.html",
      "thumbnailUrl": "https://books.toscrape.com/media/cache/0b/bc/0bbcd0a6f4bcd81ccb1049a52736406e.jpg"
    },
    // Additional books...
  ],
  "totalPages": 84,
  "currentPage": 1,
  "totalBooks": 999
}
```

Listing 6.1: Get Books Response

6.1.2 Get Book Details

- **Endpoint**: `GET /api/books/:id`
- **Description**: Retrieves detailed information for a specific book
- **URL Parameters**:
  - `id`: MongoDB ObjectID of the book
- **Response Format**:

```
{
  "_id": "6578a1b2c3d4e5f6a7b8c9d0",
  "title": "The Requiem Red",
  "price": 22.65,
  "inStock": true,
  "rating": 4,
  "detailUrl": "https://books.toscrape.com/catalogue/the-requiem-red_995/index.html",
  "thumbnailUrl": "https://books.toscrape.com/media/cache/0b/bc/0bbcd0a6f4bcd81ccb1049a52736406e.jpg",
  "createdAt": "2025-09-01T15:30:45.123Z"
}
```

Listing 6.2: Get Book Details Response

Chapter 7

User Interface Design

7.1 Design System

The application uses the Modern Library Palette for its color scheme:

Color Name	Hex Code	RGB	Usage
Navy Blue	#1A3D6D	(26, 61, 109)	Primary color, navigation, buttons
Slate Gray	#708090	(112, 128, 144)	Secondary elements, borders
White/Light Cream	#FAF9F6	(250, 249, 246)	Backgrounds, cards
Mustard Yellow	#E1AD01	(225, 173, 1)	Accents, highlights, ratings
Muted Teal	#4C9085	(76, 144, 133)	Secondary buttons, accents

Table 7.1: Color Palette

7.2 Responsive Design

The application is fully responsive with the following breakpoints:

Device	Width Range	Design Adaptations
Mobile	¡ 480px	Single column, stacked elements
Tablet	480px - 768px	Two columns, simplified filters
Small Desktop	768px - 1024px	Three columns, full features
Large Desktop	¿ 1024px	Four or more columns, expanded layout

Table 7.2: Responsive Breakpoints

7.3 Key Components

7.3.1 Navbar

- Logo and application title
- Navigation links: Home, Favorites, Cart
- GitHub repository link
- Badge counters for cart and favorites

7.3.2 Book Card

- Book thumbnail image
- Title with truncation for long titles
- Price display
- Rating display (1-5 stars)
- Stock availability indicator
- Add to Cart button (disabled for out-of-stock items)
- Add to Favorites button

7.3.3 Filters Panel

- Search input with debounced queries
- Rating filter (dropdown)
- In-stock filter (checkbox)
- Clear search button
- Search indicator text

7.3.4 Pagination

- Current page indicator
- Next/Previous page navigation
- Page number buttons
- Ellipsis for large page ranges
- Total books and current range display

7.3.5 Cart

- List of cart items
- Quantity adjustment controls
- Item removal button
- Price subtotal calculation
- Order summary with totals
- Clear cart button
- Checkout button

7.3.6 Favorites

- List of favorited books
- Remove from favorites option
- Add to cart from favorites
- Empty state with message

Chapter 8

Implementation Details

8.1 State Management

The application uses React Context API for global state management, particularly for cart and favorites.

```
import React, { createContext, useContext, useState, useEffect } from '
    react';

const AppContext = createContext();

export const useAppContext = () => useContext(AppContext);

export const AppProvider = ({ children }) => {
  // Initialize state from localStorage if available
  const [cart, setCart] = useState(() => {
    const savedCart = localStorage.getItem('cart');
    return savedCart ? JSON.parse(savedCart) : {};
  });

  const [favorites, setFavorites] = useState(() => {
    const savedFavorites = localStorage.getItem('favorites');
    return savedFavorites ? JSON.parse(savedFavorites) : {};
  });

  // Save to localStorage when state changes
  useEffect(() => {
    localStorage.setItem('cart', JSON.stringify(cart));
  }, [cart]);

  useEffect(() => {
    localStorage.setItem('favorites', JSON.stringify(favorites));
  }, [favorites]);

  // Cart management functions
  const addToCart = (book) => {
    setCart(prev => ({
      ...prev,
      [book._id]: {
        book,
        quantity: (prev[book._id]?.quantity || 0) + 1
      }
    }));
  };

  const removeFromCart = (bookId) => {
    setCart(prev => {
      const newCart = { ...prev };
      delete newCart[bookId];
      return newCart;
    });
  };

  const updateCartQuantity = (bookId, quantity) => {
    if (quantity < 1) return;

    setCart(prev => ({
      ...prev,
      [bookId]: {
        ...prev[bookId],
        quantity
      }
    }));
  };

  const clearCart = () => {
    setCart({});
  };

  // Favorites management functions
  const toggleFavorite = (book) => {
    setFavorites(prev => {
      if (prev[book._id]) {
        const newFavorites = { ...prev };
        delete newFavorites[book._id];
        return newFavorites;
      } else {
        return {
          ...prev,
          [book._id]: { book }
        };
      }
    });
  };

  // Context value to be provided
  const value = {
    cart,
    favorites,
    addToCart,
    removeFromCart,
    updateCartQuantity,
    clearCart,
    toggleFavorite,
    cartCount: Object.keys(cart).length,
    favoritesCount: Object.keys(favorites).length,
    isInCart: (bookId) => !!cart[bookId],
    isFavorite: (bookId) => !!favorites[bookId],
    cartTotal: Object.values(cart).reduce(
      (total, item) => total + item.book.price * item.quantity,
      0
    )
  };

  return (
    <AppContext.Provider value={value}>
      {children}
    </AppContext.Provider>
  );
};
```

Listing 8.1: AppContext Implementation

8.2 API Integration

The application uses Axios for API requests and implements debouncing for search queries.

```
import { useState, useEffect, useCallback } from 'react';
import axios from 'axios';
import { API_ENDPOINTS } from '../config/api';

const Home = () => {
  const [searchTerm, setSearchTerm] = useState('');
  const [filters, setFilters] = useState({
    page: 1,
    rating: '',
    inStock: false,
    search: ''
  });

  // Debounced search implementation
  useEffect(() => {
    const timerId = setTimeout(() => {
      if (searchTerm !== filters.search) {
        setFilters(prev => ({
          ...prev,
          search: searchTerm,
          page: 1 // Reset to first page on new search
        }));
      }
    }, 500); // 500ms debounce delay

    return () => clearTimeout(timerId);
  }, [searchTerm]);

  // API fetch implementation
  const fetchBooks = useCallback(async () => {
    try {
      setLoading(true);
      const params = new URLSearchParams();

      for (const key in filters) {
        if (filters[key]) {
          params.append(key, filters[key]);
        }
      }

      const response = await axios.get(`${API_ENDPOINTS.books}?${params
    }`);
      setBooks(response.data.books);
      setTotalPages(response.data.totalPages);
      setTotalBooks(response.data.totalBooks);
      setError(null);
    } catch (err) {
      setError('Failed to load books. Please try again later.');
      console.error(err);
    } finally {
      setLoading(false);
    }
  }, [filters]);

  useEffect(() => {
    fetchBooks();
  }, [fetchBooks]);

  // Component implementation
};
```

Listing 8.2: API Integration with Debouncing

8.3 Responsive Design Implementation

The application uses CSS media queries and flexible layouts for responsive design.

```
/* Base styles for all devices */
.books-grid {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(250px, 1fr));
  gap: 2rem;
  width: 100%;
}

.book-card {
  display: flex;
  flex-direction: column;
  border-radius: 8px;
  overflow: hidden;
  transition: transform 0.3s ease, box-shadow 0.3s ease;
}

/* Media queries for responsive design */
/* Mobile devices */
@media (max-width: 480px) {
  .books-grid {
    grid-template-columns: 1fr;
    gap: 1rem;
  }

  .filters-container {
    flex-direction: column;
    gap: 0.8rem;
  }
}

```

```css
/* Tablet devices */
@media (min-width: 481px) and (max-width: 768px) {
  .books-grid {
    grid-template-columns: repeat(2, 1fr);
    gap: 1.5rem;
  }
}

/* Desktop devices */
@media (min-width: 769px) and (max-width: 1024px) {
  .books-grid {
    grid-template-columns: repeat(3, 1fr);
  }
}

/* Large desktop devices */
@media (min-width: 1025px) {
  .books-grid {
    grid-template-columns: repeat(4, 1fr);
  }

  .container {
    width: 90%;
  }
}

/* Extra large desktop devices */
@media (min-width: 1440px) {
  .books-grid {
    grid-template-columns: repeat(5, 1fr);
  }

  .container {
    width: 85%;
    max-width: 1800px;
  }
}
```

Listing 8.3: Responsive CSS Implementation

Chapter 9

Testing

9.1 Manual Testing Scenarios

- **Book Browsing Test**: Verify that books load and display properly
- **Search Functionality Test**: Test text search with various queries
- **Filter Test**: Test rating and in-stock filters
- **Pagination Test**: Verify page navigation works correctly
- **Cart Functionality Test**: Add, update, and remove items from cart
- **Favorites Test**: Add and remove books from favorites
- **Responsive Design Test**: Test on various screen sizes
- **API Error Handling Test**: Test behavior when API is unavailable

9.2 Performance Considerations

- **Debounced Search**: Prevents excessive API calls during typing
- **MongoDB Text Index**: Optimizes search query performance
- **Pagination**: Limits data transfer and improves load times
- **Responsive Images**: Appropriate sizing for different devices
- **Minimal State Updates**: Optimized React component rendering

Chapter 10

Deployment

10.1 Local Development

- **Prerequisites**: Node.js, MongoDB, npm
- **Setup Steps**:
  1. Clone repository
  2. Install dependencies for each component
  3. Start MongoDB locally
  4. Run scraper to populate database
  5. Start backend server
  6. Start frontend development server
- **Automation Scripts**:
  - `setup.bat`: First-time setup script
  - `start-app.bat`: Starts all application components
  - `stop-app.bat`: Gracefully stops running components

10.2 Production Deployment

- **Frontend**: Netlify deployment
  - Build command: `npm run build`
  - Publish directory: `dist/`
  - Environment variables: `VITE_API_URL`
- **Backend**: Render web service
  - Build command: `npm install`
  - Start command: `npm start`
  - Environment variables: `NODE_ENV`, `PORT`, `MONGODB_URI`
- **Database**: MongoDB Atlas

  - Cluster configuration: M0 Free Tier
  - Database access: Username/password authentication
  - Network access: IP allowlist

10.3 Environment Variables

- **Backend**:
  - `NODE_ENV`: `development` or `production`
  - `PORT`: Server port (default: 5000)
  - `MONGODB_URI`: MongoDB connection string
- **Frontend**:
  - `VITE_API_URL`: Backend API base URL

Chapter 11

Future Enhancements

11.1 Potential Improvements

- **User Authentication**: Add user accounts and profile management
- **Advanced Search**: Implement full-text search with filtering
- **Book Details Page**: Expand book information display
- **Reviews System**: Allow users to leave and view book reviews
- **Order Processing**: Complete checkout flow with order history
- **Book Recommendations**: Implement recommendation algorithm
- **Admin Dashboard**: Create interface for managing book data
- **Dark Mode**: Implement theme toggling functionality
- **Performance Optimizations**: Implement server-side rendering

11.2 Scalability Considerations

- **Database Sharding**: For large book collections
- **Caching Layer**: Redis for frequently accessed data
- **API Rate Limiting**: Protect against abuse
- **CDN Integration**: For static assets and images
- **Containerization**: Docker for consistent deployment
- **Microservices**: Split functionality into separate services

Chapter 12

Conclusion

The Book Explorer application demonstrates a comprehensive full-stack web application architecture with data scraping, database storage, API development, and frontend presentation. It showcases modern web development practices including responsive design, state management, API integration, and deployment strategies.

The modular structure of the application makes it highly maintainable and extensible, allowing for future enhancements and feature additions. The clean separation between data acquisition (scraper), data serving (backend), and presentation (frontend) follows best practices in software architecture.

This documentation provides a thorough overview of the application's design, implementation, and deployment, serving as a comprehensive reference for understanding and extending the Book Explorer system.