**Full Stack Development with MERN**

**API Development and Integration Report**

|  |  |
| --- | --- |
| Date | 11-07-2024 |
| Team ID | SWTID1719997659 |
| Project Name | Book-Nest |
| Maximum Marks |  |

**Project Title: Book-Nest**  
**Date:** 11-07-2024  
**Prepared by:** J. Marthand Bhargav, G. Aakash , G. Harshith and V.Nagendra

**Objective**  
The objective of this report is to document the API development progress and key aspects of the backend services implementation for the Book-Nest project.

**Technologies Used**

* **Backend Framework:** Node.js with Express.js

Express.js provides a robust framework for building the backend server and handling HTTP requests.

Express.js simplifies routing for handling endpoints like /api/books for fetching books, /api/users for user authentication, etc.

* **Database:** MongoDB

MongoDB stores book data, user information, and other application data in a flexible JSON-like format.

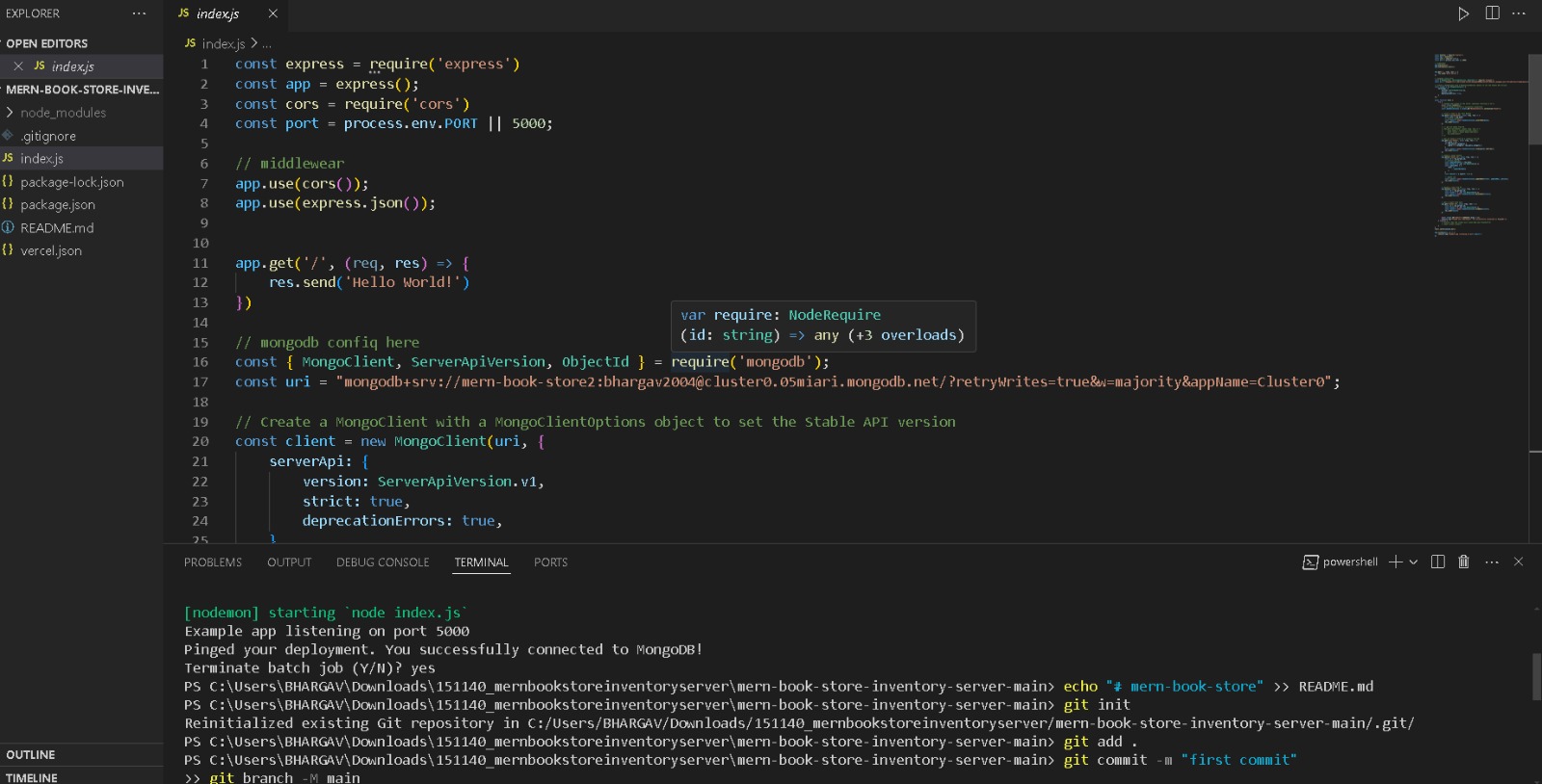
Storing book details such as title, author, price, and user information like usernames and hashed passwords.

* **Authentication:** JWT

JWT tokens are generated upon user login and used to authenticate subsequent API requests.

When a user logs in, a JWT token is issued containing the user's ID and other relevant information. This token is sent with each subsequent API request for authentication and authorization purposes.

**Project Structure**



**Key Directories and Files**

**Controllers** will handle business logic related to books, users, orders, etc. They orchestrate data retrieval from models and format responses for routes.

**Models** define schemas for MongoDB collections (e.g., Book.js), encapsulate data access methods (e.g., CRUD operations), and enforce data integrity.

**Routes** (e.g., bookRoutes.js) define API endpoints (/api/books) and map HTTP methods (GET, POST, PUT, DELETE) to corresponding controller functions.

**Middlewares** ensure application security and functionality by intercepting requests/responses; they handle tasks like authentication (authMiddleware.js) or logging.

**Config files** (database.js, etc.) centralize configuration settings such as database connections or API keys, keeping them separate from application logic.

**API Endpoints**  
 The main API endpoints and their purposes:

**Books**

1. **GET /api/books**
   * **Purpose**: Retrieve a list of all books available in the bookstore.
   * **Example**: Returns an array of book objects with details like title, author, price, etc.
2. **GET /api/books/**
   * **Purpose**: Retrieve details of a specific book by its ID.
   * **Example**: Returns details of a single book identified by :id.
3. **POST /api/books**
   * **Purpose**: Add a new book to the bookstore inventory.
   * **Example**: Accepts JSON payload with book details and creates a new book record in the database.
4. **PUT /api/books/**
   * **Purpose**: Update details of a specific book.
   * **Example**: Updates the book identified by :id with new information provided in the request payload.
5. **DELETE /api/books/**
   * **Purpose**: Delete a specific book from the bookstore inventory.
   * **Example**: Removes the book identified by :id from the database.

**Users**

1. **POST /api/users/register**
   * **Purpose**: Register a new user.
   * **Example**: Accepts user registration details (e.g., username, email, password), creates a new user account.
2. **POST /api/users/login**
   * **Purpose**: Authenticate user credentials and generate a JWT token.
   * **Example**: Validates user login credentials (e.g., username/email and password), returns a JWT token for subsequent authenticated requests.
3. **GET /api/users/profile**
   * **Purpose**: Retrieve current user's profile information.
   * **Example**: Requires a valid JWT token; returns user details such as username, email, etc.
4. **PUT /api/users/profile**
   * **Purpose**: Update current user's profile information.
   * **Example**: Updates user profile details like username, email, password (if allowed), etc.
5. **DELETE /api/users/**
   * **Purpose**: Delete a user account.
   * **Example**: Deletes the user account identified by :id from the database.

**Orders**

1. **GET /api/orders**
   * **Purpose**: Retrieve a list of all orders.
   * **Example**: Returns an array of order objects with details like order ID, items, total amount, etc.
2. **GET /api/orders/**
   * **Purpose**: Retrieve details of a specific order by its ID.
   * **Example**: Returns details of a single order identified by :id.
3. **POST /api/orders**
   * **Purpose**: Place a new order.
   * **Example**: Accepts JSON payload with order details (e.g., items, quantities), creates a new order record in the database.
4. **PUT /api/orders/**
   * **Purpose**: Update details of a specific order (e.g., update status).
   * **Example**: Updates the order identified by :id with new information provided in the request payload.
5. **DELETE /api/orders/**
   * **Purpose**: Cancel a specific order.
   * **Example**: Removes the order identified by :id from the database.

**Authentication**

1. **POST /api/auth/login**
   * **Purpose**: Authenticate user credentials and generate a JWT token.
   * **Example**: Similar to /api/users/login, validates credentials and returns a JWT token.
2. **POST /api/auth/logout**
   * **Purpose**: Invalidate the current JWT token, effectively logging out the user.
   * **Example**: Requires a valid JWT token; invalidates the token on the server side.

**User Integration with Frontend**  
In my online bookstore application, user authentication is handled via RESTful APIs where users register and log in securely. Data fetching involves API endpoints that retrieve books from MongoDB, ensuring efficient retrieval and display in the frontend. User profiles are managed through API endpoints that allow fetching and updating user details, ensuring seamless integration between backend Node.js and frontend React components for a responsive and secure user experience.

Top of Form

Bottom of Form

**Error Handling and Validation:**

**Error Handling:** In my online bookstore application, error handling should be robust to provide clear feedback and maintain application stability.

* **Centralized Error Handling**: Implement middleware in Node.js to catch errors globally using try-catch blocks or error-handling middleware (app.use((err, req, res, next) => { ... })).
* **HTTP Status Codes**: Use appropriate HTTP status codes (e.g., 400 for bad requests, 404 for not found, 500 for internal server errors) to indicate the nature of the error.
* **Error Response Format**: Standardize error responses with consistent JSON format containing error codes, messages, and optional details to help frontend developers and users understand and resolve issues.
* **Logging**: Log errors with sufficient details (timestamp, error message, stack trace) to aid in debugging and monitoring application health.

**Validation:** To ensure data integrity and security in our bookstore application, we implement validation mechanisms at various levels:

* **Input Validation**: Validate incoming data (e.g., user registration fields, book details) to ensure it meets expected formats and constraints (e.g., required fields, length limits, format checks).
* **Server-side Validation**: Use libraries like express-validator in Node.js to validate and sanitize input data before processing requests, preventing injection attacks and ensuring data consistency.
* **Database Constraints**: Utilize database constraints (e.g., unique indexes, foreign key constraints) to enforce data integrity rules at the database level, preventing invalid data from being stored.
* **Error Messaging**: Provide clear and user-friendly error messages to guide users in correcting input errors (e.g., "Invalid email format", "Password must be at least 8 characters long").
* **Client-side Validation**: Implement client-side validation in React using libraries like Formik and Yup to provide immediate feedback to users and reduce unnecessary server requests.

**Security Considerations**  
**Authentication:** In our online bookstore application, strong authentication mechanisms are crucial to protect user accounts and sensitive data:

* **JWT (JSON Web Tokens)**: Use JWT for authentication tokens, securely storing them in client-side storage (localStorage or sessionStorage) and transmitting them over HTTPS.
* **Secure Password Storage**: Hash user passwords using strong cryptographic algorithms (e.g., bcrypt) before storing them in the database to prevent plaintext exposure in case of a data breach.
* **Session Management**: Implement secure session management practices to handle user sessions effectively, including token expiration and refresh mechanisms to mitigate token hijacking.
* **Two-Factor Authentication (2FA)**: Optionally, offer 2FA for enhanced account security, requiring users to provide a second form of verification (e.g., SMS code, authenticator app) in addition to their password.

**Data Encryption:** To safeguard sensitive data stored in our bookstore application's database and during transmission:

* **HTTPS**: Enforce HTTPS (SSL/TLS) for all communication between clients and the server to encrypt data in transit and protect against eavesdropping and man-in-the-middle attacks.
* **Encryption at Rest**: Utilize database encryption features (e.g., MongoDB's encryption at rest) or file-level encryption to protect data stored in the database from unauthorized access at rest.
* **Sensitive Data Encryption**: Encrypt sensitive data fields (e.g., credit card details, personal information) before storing them in the database, using strong encryption algorithms and securely managing encryption keys.
* **Data Masking**: Implement data masking techniques to anonymize or obfuscate sensitive data fields in UI and reports where full data visibility isn't necessary.