**Full Stack Development with MERN**

1. **Introduction**

**PROJECT TITLE: Freelance Platform**

**TEAM MEMBERS:**

**Kalaivani - Developer**: Responsible for both frontend and backend development, ensuring integration between the components, and implementing key features.

**AshaRani- Developer**:  Backend developer responsible for creating and managing Express routes, server-side logic, and API integration.

**Lavanya M - Developer**:  Focused on frontend development using React, including UI design and functionality for user interactions.

**Bhavadharani - Developer**: Database management with MongoDB, including schema design, data storage, and optimization for queries.

**2.Project Overview**

**Purpose:** Our purpose is to connect skilled freelancers with businesses seeking quality talent, creating a dynamic marketplace that fosters collaboration, transparency, and innovation. We aim to empower independent professionals to thrive in their careers while providing companies with the resources they need to achieve their goals efficiently and effectively."

**Features:**

**1. User Profiles**

* **Freelancer Profiles**: Include skills, experience, portfolio samples, and client reviews.
* **Client Profiles**: Allow clients to showcase their company, project history, and preferences.

**2. Job Listings**

* **Posting Projects**: Clients can easily create and post project listings with detailed descriptions, budgets, and deadlines.
* **Categories and Tags**: Organize jobs by categories (e.g., writing, design, programming) for easy navigation.

**3. Search and Filters**

* **Advanced Search**: Enable users to search for freelancers or projects using keywords, skills, location, and experience.
* **Filter Options**: Allow filtering by project type, budget, ratings, and deadlines.

**4. Proposal Submission**

* **Custom Proposals**: Freelancers can submit tailored proposals for specific projects, outlining their approach and fees.
* **Template Options**: Provide templates for freelancers to streamline proposal writing.

**5. Communication Tools**

* **Messaging System**: Built-in chat for real-time communication between freelancers and clients.
* **Video Conferencing**: Integrate video call features for discussions and interviews.

**6. Project Management Tools**

* **Task Management**: Features to assign tasks, set milestones, and track progress.
* **File Sharing**: Securely share documents, designs, and other files related to projects.

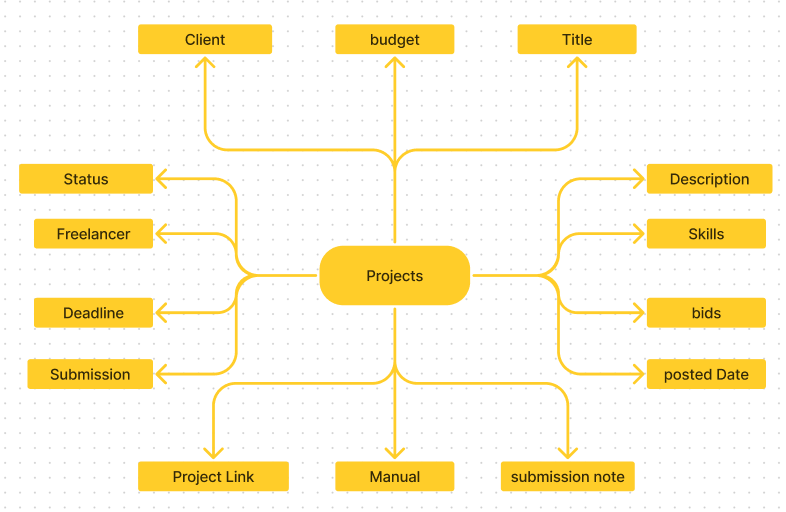
**7. Payment System**

* **Secure Payment Processing**: Offer various payment methods, including credit cards, PayPal, and direct bank transfers.
* **Escrow Services**: Hold payments in escrow until project milestones are met to ensure security for both parties.

**8. Reviews and Ratings**

* **Feedback System**: Clients can leave reviews and ratings for freelancers upon project completion, fostering trust and accountability.
* **Profile Display**: Show ratings prominently on freelancer profiles.

**3. Architecture**



*Freelance architecture*

**Frontend development**

**1. Setting the Stage:**

The SB Works frontend thrives on React.js. To get started, we'll:

* Create the initial React application structure.
* Install essential libraries for enhanced functionality.
* Organize project files for a smooth development experience.
* This solid foundation ensures an efficient workflow as we bring the SB Works interface to life.

**Backend Development**

**1. Project Setup:**

* Create a project directory and initialize it using npm init.
* Install required dependencies like Express.js, Mongoose, body-parser, and cors.

**2. Database Configuration:**

* Set up a MongoDB database (locally or using a cloud service like MongoDB Atlas).
* Create collections for:
* Users (storing user information, account type)
* Projects (project details, budget, skills required)
* Applications (freelancer proposals, rate, portfolio link)
* Chat (communication history for each project)
* Freelancer (extended user details with skills, experience, ratings)

**3. Express.js Server:**

* Create an Express.js server to handle HTTP requests and API endpoints.
* **Configure body-parser to parse request bodies and cors for cross-origin requests.**

**4. API Routes:**

* Define separate route files for user management, project listing, application handling, chat functionality, and freelancer profiles.
* Implement route handlers using Express.js to interact with the database:
* User routes: registration, login, profile management.
* Project routes: project creation, listing, details retrieval.
* Application routes: submit proposals, view applications.
* Chat routes: send and receive messages within projects.
* Freelancer routes: view and update profiles, showcase skills.

**5. Data Models:**

* Define Mongoose schemas for each data entity:
* User schema
* Project schema
* Application schema
* Chat schema
* Freelancer schema (extends User schema with skills, experience)
* Create Mongoose models to interact with the MongoDB database.
* Implement CRUD operations for each model to manage data.

**6. User Authentication:**

* Implement user authentication using JWT or session-based methods.
* Create routes and middleware for user registration, login, and logout.
* Use authentication middleware to protect routes requiring user authorization (e.g., applying for projects).

**7. Project Management:**

* Allow clients to post projects with details and budget.
* Enable freelancers to browse projects, search by skills, and submit proposals.
* Implement a system for clients to review applications and choose freelancers.

**8. Secure Communication & Collaboration:**

* Integrate a secure chat system within projects for communication between clients and freelancers.
* Allow file attachments and feedback exchange to facilitate collaboration.

**9. Admin Panel (Optional):**

* Implement an admin panel with functionalities like:
* Managing users
* Monitoring project updates and applications
* Accessing transaction history

**Database (MongoDB)**

* Set up a MongoDB database either locally or using a cloud-based MongoDB service like MongoDB Atlas.
* Create a database and define the necessary collections for users, freelancer, projects, chats, and applications.
* Connect the database to the server with the code provided below.

**4.Setup Instructions**

#### **Prerequisites**

Before starting with the setup, make sure your system has the following software installed:

1. **Node.js**  
   Essential for running the server and managing project dependencies.
2. **MongoDB**  
   The database system used to store user, seller, and project-related information.
3. **Vite**  
   A modern build tool for fast frontend development and bundling.
4. **Git**  
   Used to clone the project repository.

#### **Installation Process**

##### **1. Clone the Repository**

* Use Git to clone the project repository and navigate into the project folder:

git clone <repository-url>

cd Project-Name

##### **2. Backend Setup**

1. Move into the backend directory:

cd backend

1. Install all required backend dependencies:

npm install

1. Configure the environment variables:
   * Create a .env file in the backend folder and add the following:

env

PORT=4000

MONGO\_URI=<your-mongodb-uri>

JWT\_SECRET=<your-jwt-secret>

1. Start the backend server:

node server.js

##### **3. Frontend Setup**

1. Navigate to the frontend directory:

cd ../frontend

1. Install all frontend dependencies:

npm install

1. Start the frontend server:

npm run dev

##### **4. Access the Application**

* Open your browser and go to:

arduino

http://localhost:<port>

* Replace <port> with the frontend server’s port (typically **3000**).

**5. Folder Structure**

#### **Client (React Frontend)**

The React frontend is located in the frontend/src directory and is structured as follows:

1. **Admin, Seller, User**
   * Each folder represents a specific user type in the application.
   * **Content**: Components and pages tailored to the roles:
     + **Admin**: Manage books, users, and orders.
     + **Seller**: Manage their inventory and orders.
     + **User**: Browse books, add items to the cart, and place orders.
2. **Components**
   * Houses reusable components shared across the app.
   * Examples: Navigation bars, buttons, forms, and modals.
   * **Purpose**: Promotes modularity and avoids redundancy.
3. **App.jsx**
   * The central application file where routes are defined.
   * Links URLs to components and serves as the app's entry point.
   * Dynamically renders components based on user actions and navigation.
4. **main.jsx**
   * The root file responsible for rendering the React app into the DOM.
   * Connects the app to the HTML file and imports global settings or context providers.
5. **App.css & index.css**
   * **App.css**: Contains styles specific to the app's components and pages.
   * **index.css**: Holds global styles that are applied across the application.

#### **Server (Node.js Backend)**

The Node.js backend is organized within the Backend directory as follows:

1. **db**
   * **Subfolders**:
     + **Seller** and **Users** for managing database schemas and models.
   * **config.js**: Contains database configuration and establishes the MongoDB connection.
2. **uploads**
   * Stores user-uploaded files like book images and profile pictures.
3. **server.js**
   * The main server file where the Express server is initialized.
   * Handles routes, middleware, and other server-level configurations.
4. **package.json & package-lock.json**
   * **package.json**: Defines dependencies, scripts, and metadata for the backend.

**6. Running the Application**

#### **Frontend Setup**

1. **Navigate to the Frontend Directory**  
   Open your terminal and run:

cd book-store/frontend

1. **Start the Frontend Server**  
   Run the following command to launch the frontend in development mode:

npm start

* + The frontend will typically run at:

arduino

http://localhost:3000

#### **Backend Setup**

1. **Open a New Terminal Window**
   * Keep the frontend server running in one terminal, and open a new terminal window.
2. **Navigate to the Backend Directory**  
   Run the following command:

cd book-store/backend

1. **Start the Backend Server**  
   Use this command to start the backend server:

npm start

* + The backend server will run on the port defined in your .env file, e.g.:

arduino

http://localhost:4000

#### **Testing the Application**

* With both servers running, the **frontend** will communicate seamlessly with the **backend**, enabling all features of the Book-Store application.
* Open your browser and navigate to the frontend URL (http://localhost:3000) to start using the app.

1. **API Documentation**

#### **General Information**

* **Base URL:** http://localhost:4000

### ****Admin Endpoints****

1. **Admin Login**
   * **Endpoint:** /alogin
   * **Method:** POST
   * **Description:** Allows an admin to log in.
   * **Request Body:**

json

Copy code

{

"email": "admin@example.com",

"password": "password123"

}

* + **Response:**
    - **Success:**

{

"Status": "Success",

"user": {

"id": "user\_id",

"name": "Admin Name",

"email": "admin@example.com"

}

}

* + - **Failure:** "login fail" or "no user"

1. **Admin Registration**
   * **Endpoint:** /asignup
   * **Method:** POST
   * **Description:** Registers a new admin.
   * **Request Body:**

json

Copy code

{

"name": "Admin Name",

"email": "admin@example.com",

"password": "password123"

}

* + **Response:** "Account Created" or "Already have an account"

### ****User Endpoints****

1. **Get All Users**
   * **Endpoint:** /users
   * **Method:** GET
   * **Description:** Retrieves all users.
   * **Response:** Array of user objects.
2. **Delete User**
   * **Endpoint:** /userdelete/:id
   * **Method:** DELETE
   * **Description:** Deletes a user by ID.
   * **Response:** 200 OK on success, or an error message.
3. **User Login**
   * **Endpoint:** /login
   * **Method:** POST
   * **Description:** Logs in a user.
   * **Request Body:**

{

"email": "user@example.com",

"password": "password123"

}

* + **Response:** Success with user info or failure messages.

1. **User Registration**
   * **Endpoint:** /signup
   * **Method:** POST
   * **Description:** Registers a new user.
   * **Request Body:**

{

"name": "User Name",

"email": "user@example.com",

"password": "password123"

}

* + **Response:** "Account Created" or "Already have an account"

### ****Seller Endpoin****

1. **Seller Login**
   * **Endpoint:** /slogin
   * **Method:** POST
   * **Description:** Logs in a seller.
   * **Request Body:**

{

"email": "seller@example.com",

"password": "password123"

}

* + **Response:** Success with seller info or failure messages.

1. **Seller Registration**
   * **Endpoint:** /ssignup
   * **Method:** POST
   * **Description:** Registers a new seller.
   * **Request Body:**

{

"name": "Seller Name",

"email": "seller@example.com",

"password": "password123"

}

* + **Response:** "Account Created" or "Already have an account"

1. **Add Book**
   * **Endpoint:** /items
   * **Method:** POST
   * **Description:** Adds a new book item.
   * **Request Body:** Form-data with book details and itemImage file.
   * **Response:** Created item object or error message.
2. **Get Seller’s Items**
   * **Endpoint:** /getitem/:userId
   * **Method:** GET
   * **Description:** Retrieves items uploaded by a seller.
   * **Response:** Array of item objects.

### ****Order Endpoints****

1. **Add Order**
   * **Endpoint:** /userorder
   * **Method:** POST
   * **Description:** Creates a new order.
   * **Request Body:**

{

"flatno": "123",

"city": "City",

"state": "State",

"pincode": "123456",

"totalamount": 100,

"seller": "Seller Name",

"sellerId": "seller\_id",

"BookingDate": "2024-11-14",

"description": "Order description",

"Delivery": "Delivery details",

"userId": "user\_id",

"userName": "User Name",

"booktitle": "Book Title",

"bookauthor": "Author",

"bookgenre": "Genre",

"itemImage": "path/to/image.jpg"

}

* + **Response:** Created order object or error message.

1. **Get User Orders**
   * **Endpoint:** /getorders/:userId
   * **Method:** GET
   * **Description:** Retrieves orders placed by a user.
   * **Response:** Array of order objects.

### ****Wishlist Endpoints****

1. **Get All Wishlist Items**
   * **Endpoint:** /wishlist
   * **Method:** GET
   * **Description:** Retrieves all wishlist items.
   * **Response:** Array of wishlist item objects.
2. **Add Item to Wishlist**
   * **Endpoint:** /wishlist/add
   * **Method:** POST
   * **Description:** Adds an item to the wishlist.
   * **Request Body:**

{

"itemId": "item\_id",

"title": "Item Title",

"itemImage": "path/to/image.jpg",

"userId": "user\_id",

"userName": "User Name"

}

* + **Response:** Created wishlist item or error message.

1. **Remove Item from Wishlist**
   * **Endpoint:** /wishlist/remove
   * **Method:** POST
   * **Description:** Removes an item from the wishlist.
   * **Request Body:**

json

{

"itemId": "item\_id"

}

* + **Response:** Success message or error message.

1. **Authentication**

**1. Authentication:**

**User Login:**

* When a user logs in (for both regular users and sellers), their credentials (email and password) are validated against the stored data in the database.
* Upon successful login, a **JSON Web Token (JWT)** is issued. This token contains essential information, such as the **user ID**, **role**, and **expiration time**, which are used for further authorization.

**Token Storage:**

* The JWT is sent back to the client (frontend) and stored locally, either in **cookies** or **localStorage**, to be included in subsequent requests for secure access.

**2. Authorization:**

**Role-Based Access Control:**

* For secure endpoints, such as creating orders or managing products, the client must include the **JWT** in the Authorization header of the HTTP request.
* Middleware on the server validates the token by checking its signature and expiration. Upon successful validation, it extracts the user’s role (admin, seller, or user) from the token and grants access based on their permissions.

**Role Permissions:**

* **Admins**: Admins can access routes related to managing users, handling orders, and administrative tasks.
* **Sellers**: Sellers can manage their inventory, such as adding or editing products.
* **Users**: Regular users can browse the products, place orders, and manage their own profiles.

**3. Token Expiration and Renewal:**

**Token Expiration:**

* JWTs are issued with an expiration time, typically set to a specific period (e.g., 1 hour). This expiration enhances security by limiting how long a token can be used before requiring re-authentication.

**Token Renewal:**

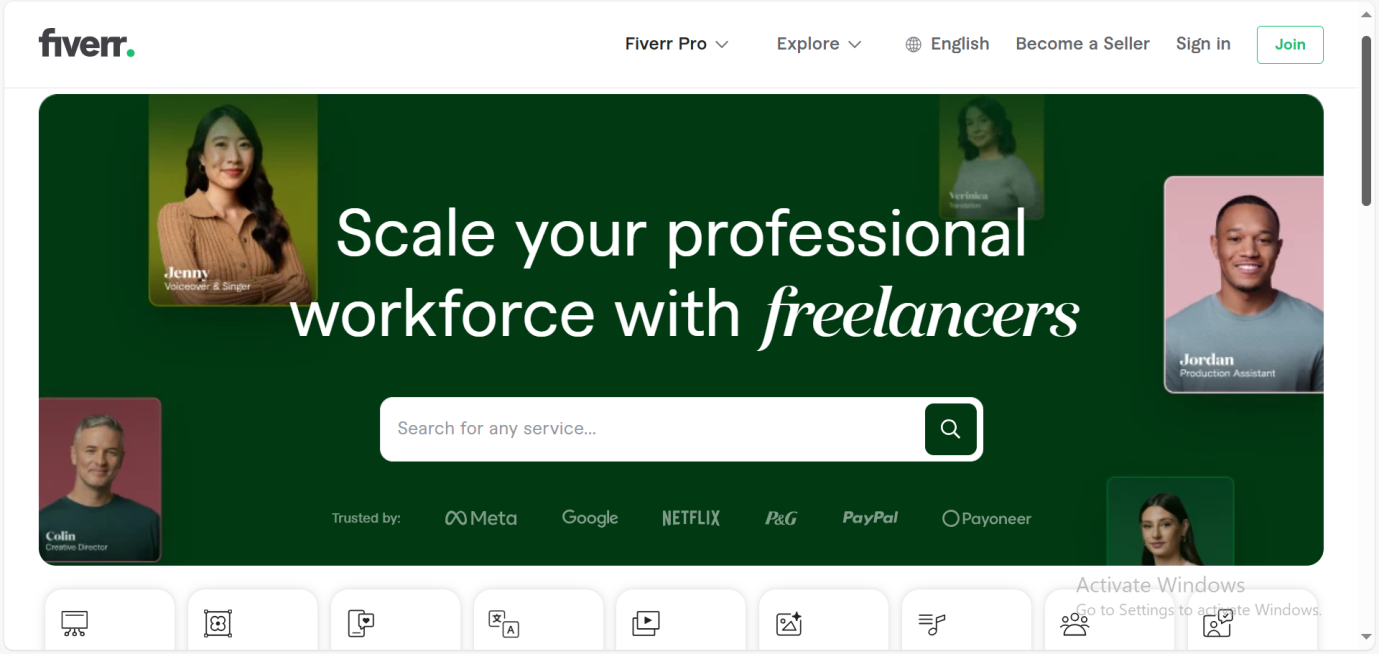
* If a JWT expires, the user will be required to log in again. Alternatively, a **refresh token** mechanism can be implemented.
* **Refresh Tokens**: Refresh tokens are long-lived tokens that allow the user to obtain a new JWT without needing to log in again. When the JWT expires, the client can request a new JWT using the refresh token.

**Security Considerations:**

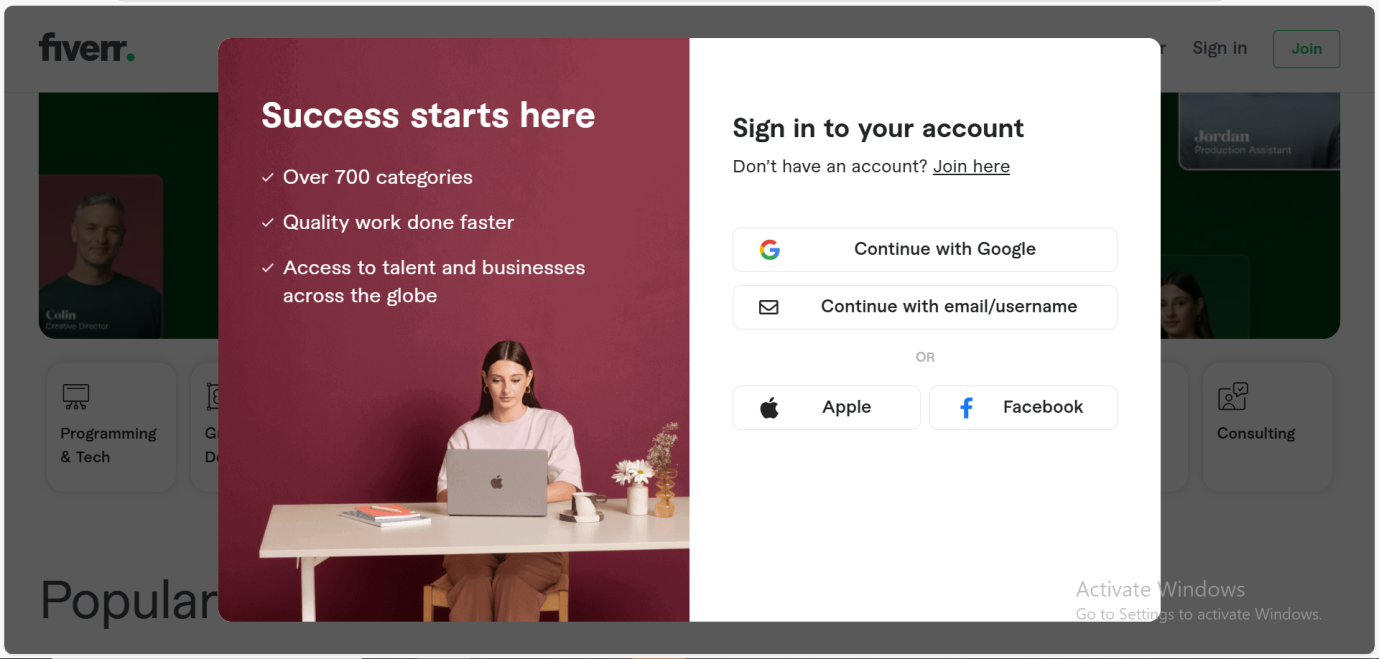
* Both JWTs and refresh tokens should be stored securely. Refresh tokens, in particular, should be stored in secure environments (such as **HttpOnly cookies**) to prevent unauthorized access.

1. **User Interface**

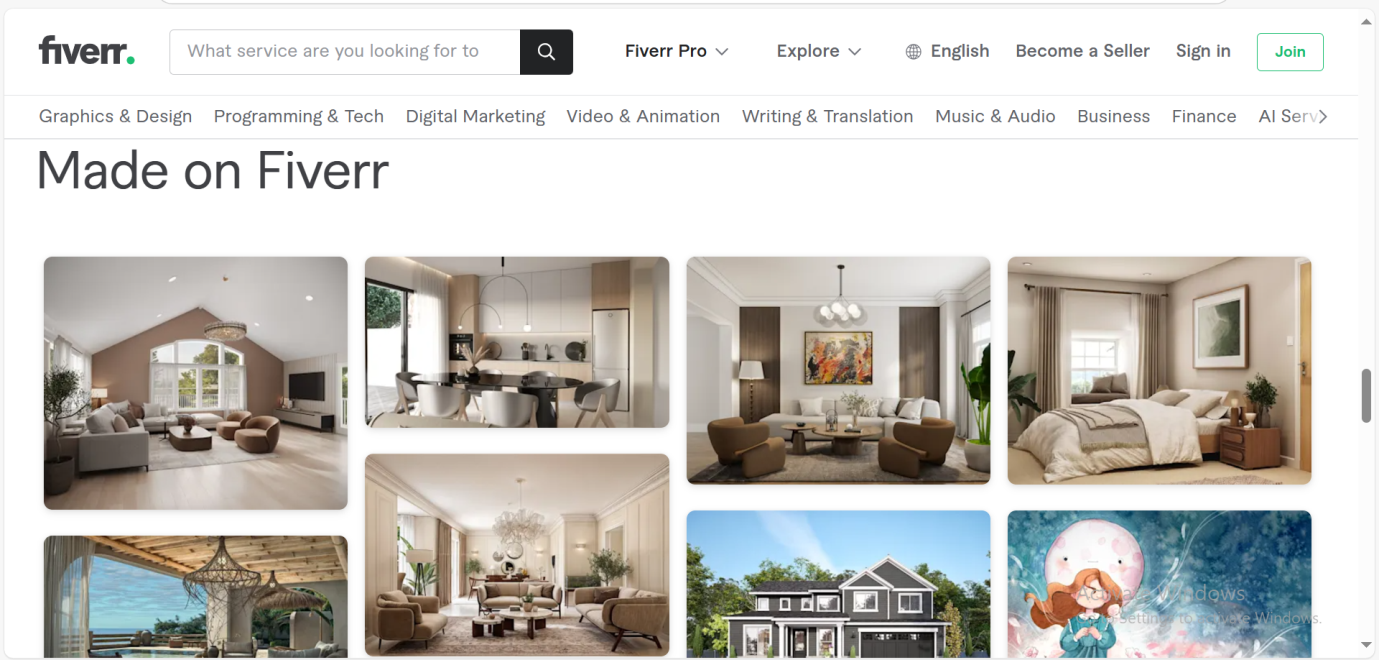
Home page:



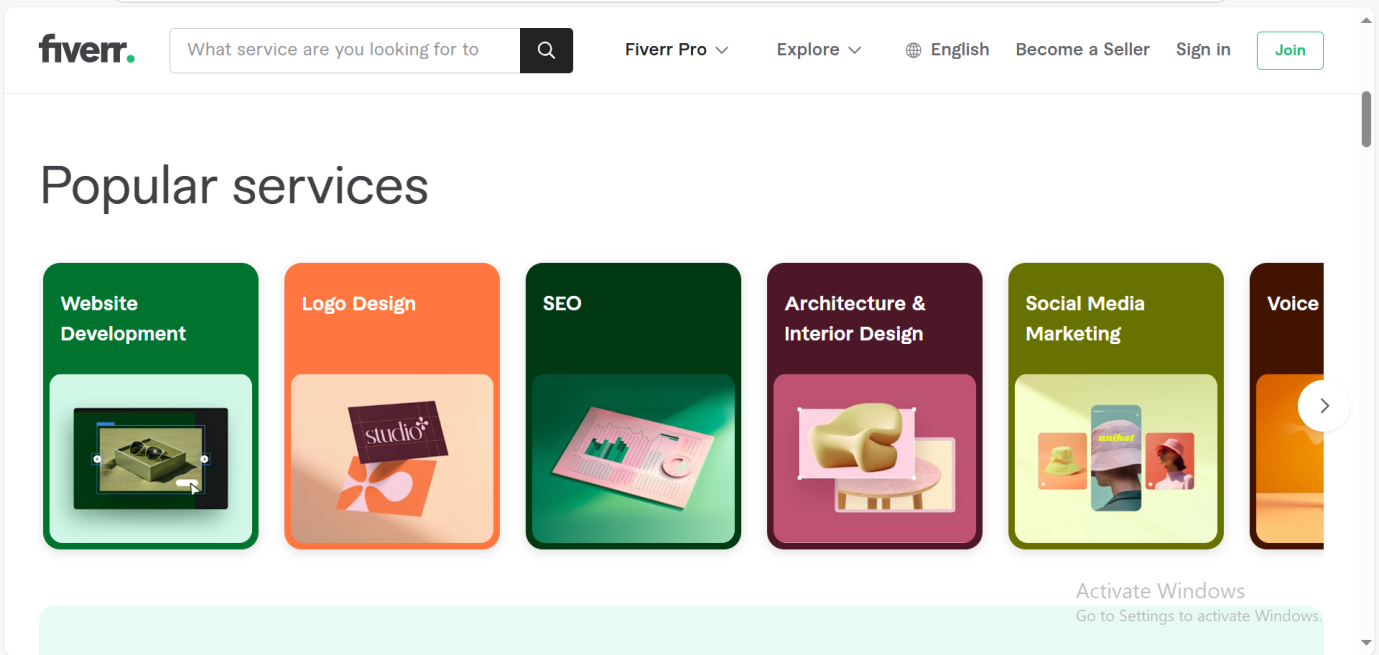
Login page:



Made on fiverr:



Services provided :



**10. Testing**

**Testing Strategy**

1. **Unit Testing**:
   * Unit testing focuses on testing individual components and functions to ensure they behave as expected in isolation.
   * This testing is particularly useful for smaller, reusable parts of the codebase, such as utility functions, helper methods, and API service functions.
   * By testing these components in isolation, we can ensure that each part of the code works correctly before integrating it into larger systems.
2. **Integration Testing**:
   * Integration testing verifies that different modules and services interact as intended.
   * This includes testing interactions between the frontend and backend systems, ensuring they exchange data and communicate effectively.
   * It also covers the integration of various backend services, such as database queries, third-party API calls, and internal service communication.
3. **End-to-End (E2E) Testing**:
   * E2E testing simulates a user’s journey through the application to identify issues in real-world scenarios.
   * Core workflows such as user registration, login, browsing books, placing orders, and completing payments are tested in this phase.
   * This type of testing ensures that all parts of the application work together seamlessly from the user's perspective.

**Tools Used**

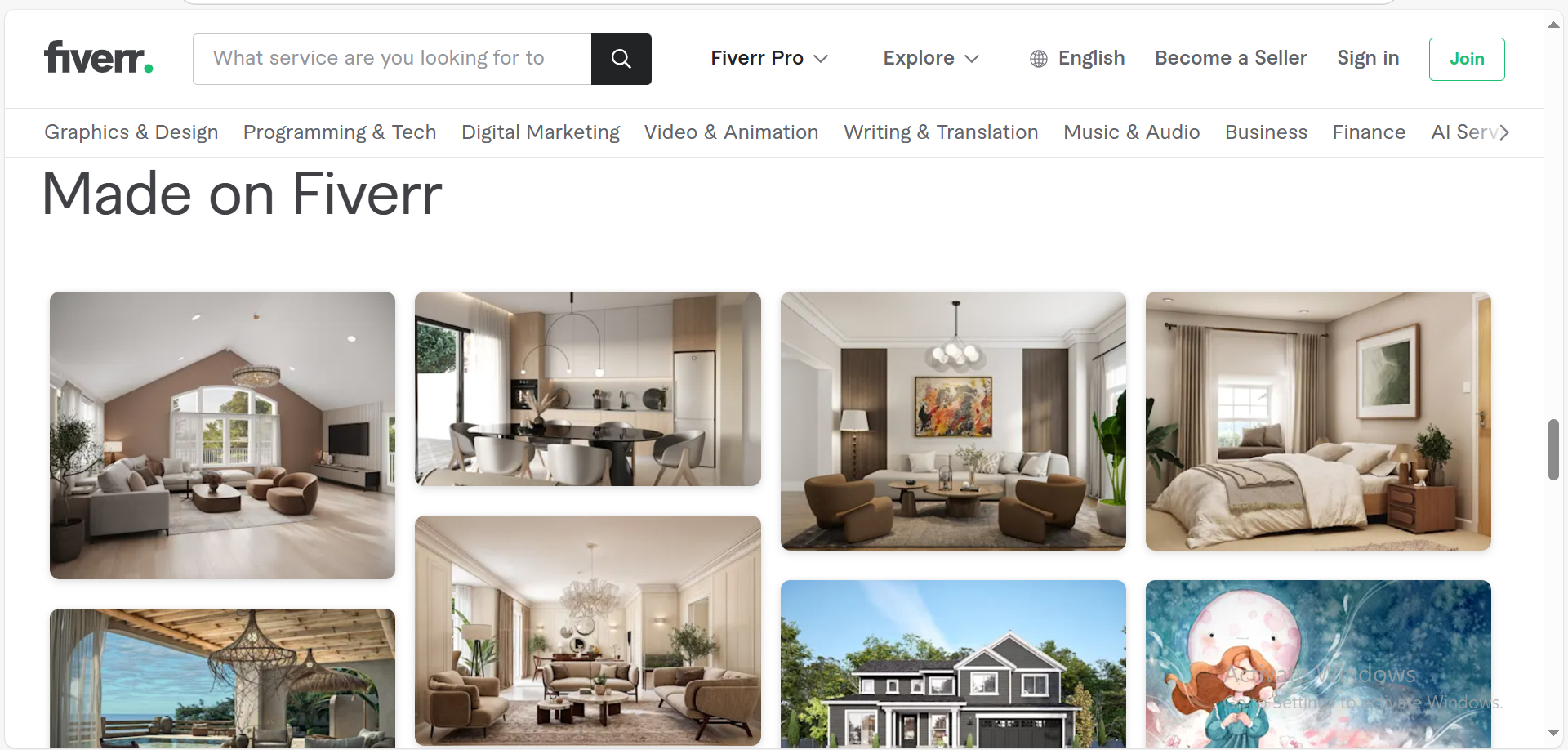
* **Jest**:
  + Jest is used for unit testing, especially on the backend. It offers robust testing capabilities, including built-in mocking functions to simulate dependencies and isolate units for testing.
  + It is especially useful for testing individual components of the backend such as utility functions, database queries, and authentication processes.
* **Mocha and Chai**:
  + Mocha and Chai are utilized for both unit and integration testing. Mocha provides a flexible framework for structuring tests, while Chai offers an expressive syntax for assertions.
  + These tools are particularly helpful for testing REST API endpoints to ensure they are functioning correctly and returning the expected results.
* **Selenium**:
  + Selenium is used for end-to-end (E2E) testing, particularly for automating frontend UI interactions.
  + It allows for simulating real user actions, such as logging in, adding items to the cart, and completing a purchase, to verify the application’s behavior across different browsers.
* **Postman**:
  + Postman is used for API testing, where we verify that API endpoints work as expected in various scenarios (e.g., different request types, headers, or response conditions).
  + It’s a powerful tool for manually testing and automating the validation of API responses.

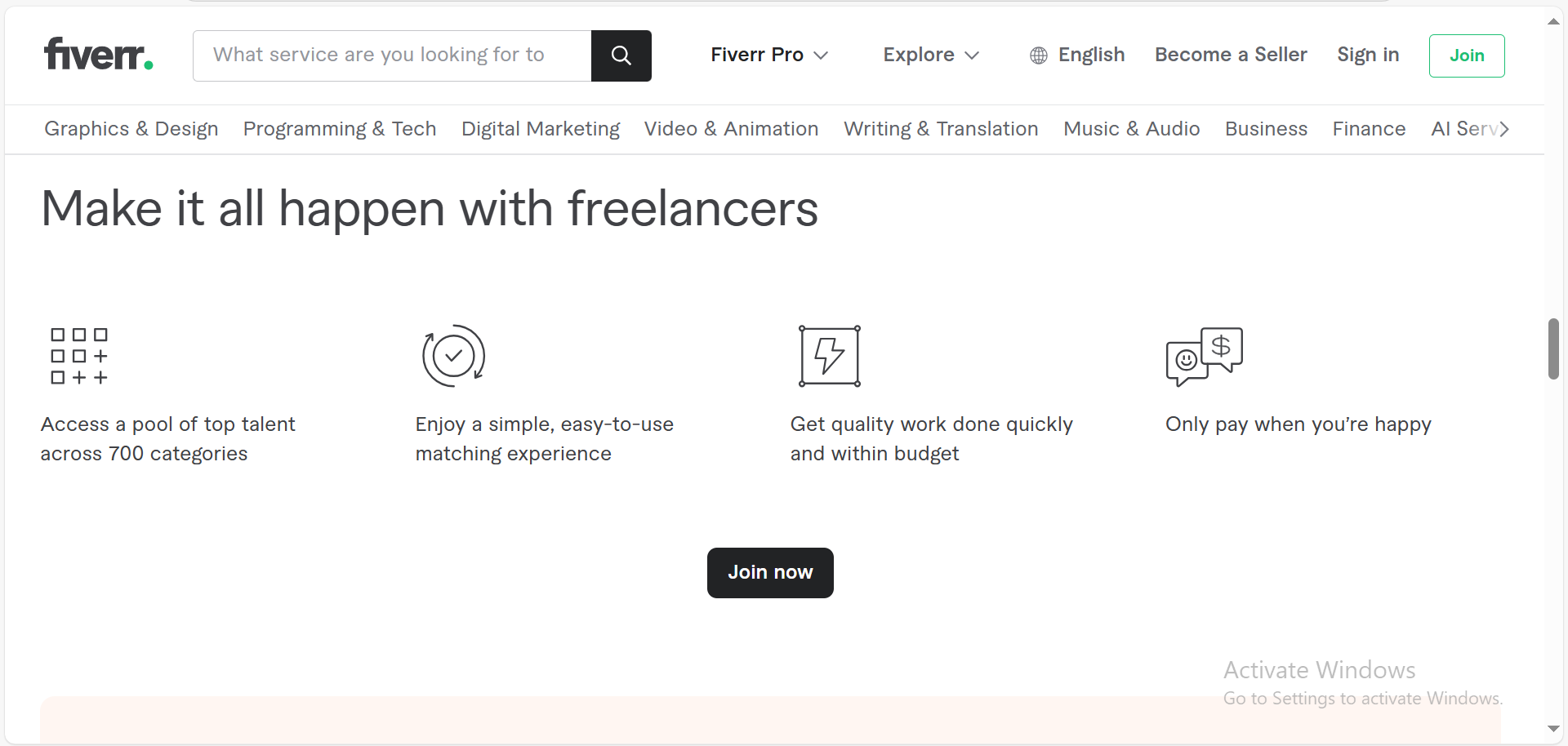
**Continuous Integration (CI)**

* **GitHub Actions**:
  + GitHub Actions is configured to automatically run tests on every push and pull request.
  + This ensures that all changes to the codebase are tested and verified before being merged into the main branch.
  + By integrating this CI tool, we can catch bugs early in the development cycle, maintain code quality, and prevent regressions, all while speeding up the development process.

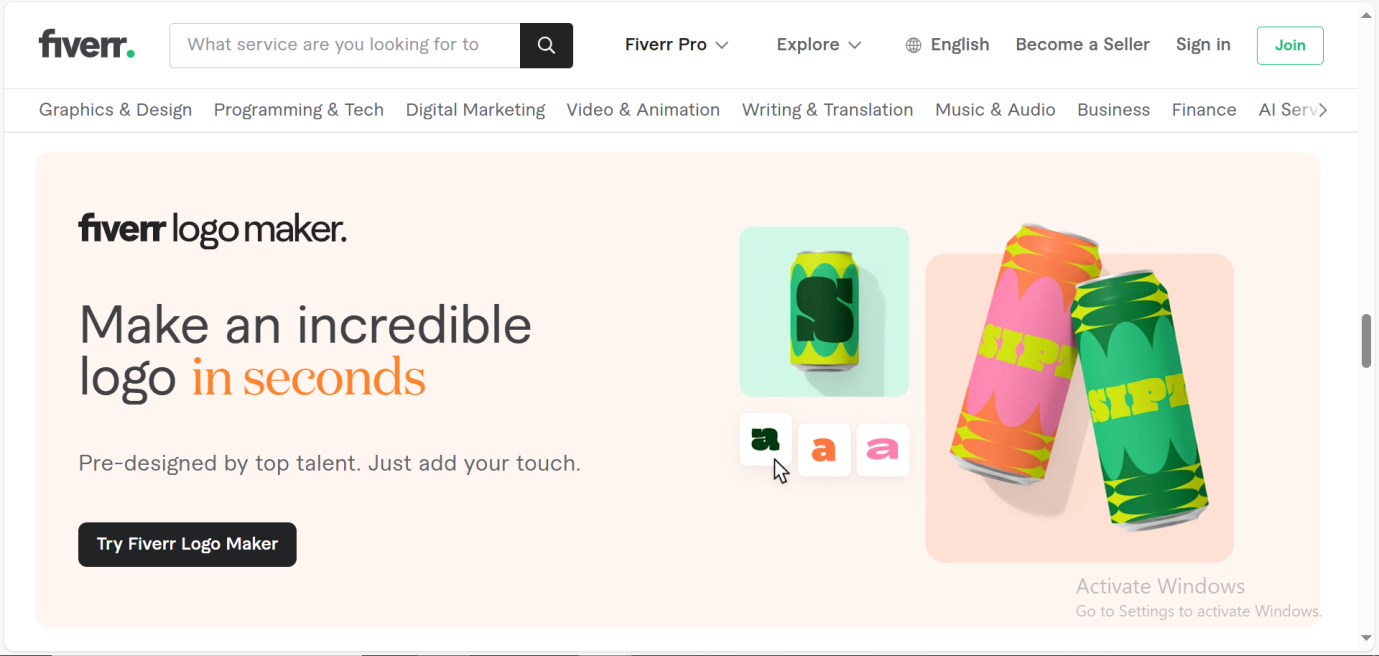
**11. Screenshots or Demo**

Made on fiverr:



Make it all happen with freelancers:

Fiverr – logo maker



**12. Known Issues**

#### **Data Persistence Delay in MongoDB**

* **Description**: Occasionally, users may experience a delay in updating or retrieving data from MongoDB. This can result in users seeing outdated information or facing slower response times when interacting with the application.
* **Impact**: Users may experience delays when accessing the latest data, such as newly added items or recent changes to their profiles.
* **Workaround**:
  + Refresh the page or retry the action to force the system to retrieve the latest data.
  + Investigate MongoDB performance, optimize database queries, and consider implementing caching strategies for faster data retrieval.

#### **2. Intermittent Authentication Timeout**

* **Description**: Users may experience timeouts during authentication, particularly when the server is under high load or during periods of heavy traffic.
* **Impact**: This issue prevents users from successfully logging in or signing up, causing frustration and potentially blocking access to the application.
* **Workaround**:
  + Increase the timeout threshold in the authentication service to allow longer processing times during peak usage periods.
  + Consider optimizing the backend infrastructure, such as adding more resources or load balancing, to handle higher traffic loads.

#### **3. API Gateway Routing Conflicts**

* **Description**: Some routes in the API gateway may conflict, leading to incorrect redirections, especially between login and signup services. This can cause issues such as users being redirected to the wrong page or encountering error messages.
* **Impact**: Users might face unexpected errors or be redirected to incorrect pages, such as landing on a login page when they intended to sign up.
* **Workaround**:
  + Review the API gateway configuration to ensure that routes are correctly mapped.
  + Use more specific and unique route paths to prevent conflicts and ensure correct redirection.

#### **4. Session Expiration Not Consistently Handled**

* **Description**: In some cases, session expiration is not consistently handled, which may lead to users encountering unauthorized access errors even though they are still active within the application.
* **Impact**: Users may experience interruptions or unexpected logouts when their session expires, affecting their ability to continue their work or use the platform smoothly.
* **Workaround**:
  + Implement a session timeout warning that prompts users to reauthenticate before the session expires, ensuring they do not lose progress or encounter unexpected logouts.
  + Regularly review and test session management logic to ensure consistency across the platform.

.

**13. Future Enhancements**

#### **1. Enhanced Search and Filter Options**

* **Description**: Implement advanced search filters that allow users to find books by specific criteria such as genre, author, publication date, language, price range, and user ratings.
* **Impact**:
  + Improves the search experience by enabling users to quickly narrow down their options from a large collection of books.
  + Helps users find books that match their interests more easily, enhancing satisfaction and retention.

#### **2. Recommendation System**

* **Description**: Develop a personalized recommendation engine that suggests books based on users' past purchases, browsing history, and preferences. This system would analyze user behavior to offer tailored book suggestions.
* **Impact**:
  + Increases user engagement by promoting relevant content and encouraging users to explore more books.
  + Drives sales by suggesting books users might not have discovered otherwise, leading to a more personalized shopping experience.

#### **3. Rating and Review System**

* **Description**: Enable users to rate and review books, allowing them to share their thoughts and experiences with the community. Reviews can help other users make informed decisions and contribute to the overall curation of the platform's content.
* **Impact**:
  + Fosters community engagement by enabling interaction between users through ratings and reviews.
  + Helps improve the quality of the platform's book selection, as users can make decisions based on others' feedback and ratings.

#### **4. Social Sharing Options**

* **Description**: Integrate social media sharing functionality to allow users to share their favorite books, reviews, or reading lists directly on platforms like Facebook, Twitter, Instagram, and other social media outlets.
* **Impact**:
  + Increases brand visibility and organic marketing by encouraging users to share their experiences with their social circles.
  + Promotes user-generated content, which can attract new users to the platform and enhance user engagement.