Tech Assignment: Develop a Simplified Full-Stack Product Display Web Application.

Project Documentation-

1. Project Overview

- **Project Name**: Briskk
- **Purpose**: Briskk is designed as a product catalog and review platform where users can browse products and submit reviews, offering a user-friendly shopping experience with detailed product listings.
- **Technologies Used**:
    - Frontend: React, CSS
    - Backend: Node.js, Express
    - Database: MongoDB

2. Key Features

- **Product Display**: Users can view list of all products with images, price, barand.
- **Product Details**: User can view details of a particular project.
- **Review System**: Users can submit ratings and comments on products, which are stored in a database and displayed dynamically.

4. Database Design

- **Product Schema**: {name, brand, price, image, review{rating, comment}}
- **Explanation**: Each product document has an array for storing reviews, with each review containing a rating and comment.

5. API Endpoints

GET /products

- Purpose: Retrieve list of all products showing image, price, brand, name.
- Request Example: GET http://localhost:3000/products
- Response: Array of JSON objects, each representing a product.

GET /products/:id

- Purpose: Retrieve details and reviews of a specific product by its ID.
- Request Example: GET http://localhost:3000/products/12345
- Response: JSON object with product details and associated reviews.

POST /products

- Purpose: Add a new product to the database.
- Request Body: {name, brand, image, price, description, reviews}
- Response: JSON object of the created product with an ID and any assigned review IDs.

POST /products/:id/review

- Purpose: Add a new review to a specific product by its ID.
- Request Body: {rating, comment}
- Response: Updated JSON object of the product, now including the new review.

6. User Guide

- **Adding a Review**: Fill in the rating (1-5) and comment, then click "Submit Review."
- **Navigating Products**: Select a product from the main list to view details and read other users' reviews.

7. Future Enhancements

- **User Authentication**: To manage user-specific review capabilities.
- **Review Edit/Delete Options**: Allow users to manage their reviews.
- **Enhanced Search/Filter**: Improve product discovery.

8. Testing and Deployment

- **Testing**: Unit tests for API endpoints and UI tests for the review form and product display.
- **Deployment**: Hosted on a local server for testing. Future versions could be deployed on platforms like Heroku or AWS for broader accessibility.

Incorporating Low-Code Tools

While low-code (LC) tools offer many advantages, including rapid development and ease of use, I chose to approach this project with traditional coding methods. Here’s why:

1. **Project Requirements and Complexity** : Traditional coding enabled me to tailor the application precisely to requirements, rather than working within the constraints of a low-code platform.
2. **Customization and Flexibility** : Building the project from the ground up gave me full control over each component, ensuring a seamless and unique user experience.
3. **Skill Development and Technical Depth** : Working through each aspect of the codebase improved my skills in various technologies and problem-solving, providing a deeper understanding of each component.

Impact on Project’s Functionality and Design

Using traditional coding instead of low-code tools resulted in:

- **Enhanced Control**: I could implement specific requirements without being limited by the capabilities of a low-code platform.
- **Customization**: Full coding allowed me to design a unique interface and feature set, optimized for the target audience.
- **Better Performance**: I could directly manage and optimize resource usage, improving overall efficiency.

Design Decisions, Challenges, and Solutions

Design Decisions

1. **Modular Code Structure**: I structured the project into two main modules: **frontend** and **backend**. For the backend, I separated the database models and product routes, keeping the code clean and organized. Similarly, in the frontend, I created individual components, each with a clear responsibility, and organized CSS files separately for streamlined styling management. This approach makes it easier to work on specific parts without impacting the entire codebase.
2. **Technologies Choice**:
   - **React for Frontend**: I went with React because it's efficient for creating dynamic and responsive user interfaces and Single Page Application. React's component-based structure made it easy to break down the UI into smaller, reusable pieces. I also wanted the frontend to be fast and interactive, and React's Virtual DOM handles this well by updating only the components that change, keeping performance smooth even as data updates frequently.
   - **Node.js and Express for Backend**: For the backend, I chose Node.js along with Express to keep things simple and efficient. Since Node is non-blocking and event-driven, it works well for handling multiple requests without slowing down. Express adds a clean structure to the routes, making the API endpoints more organized and easier to expand or debug in the future.
3. **Database Choice (MongoDB)**: MongoDB was a natural choice due to its flexibility with JSON-like documents. It lets me store products with nested data, like reviews, without the hassle of rigid schemas. Plus, MongoDB's scalability is an advantage if the project grows and needs to handle more complex data relationships.
4. **RESTful API Design**: I designed the API endpoints with REST principles, using a structure like /products and /products/:id/review. This makes the API intuitive and easy to extend, ensuring it's maintainable and accessible for future needs, like adding extra features or making the project scalable.

Challenges and Solutions

1. **Challenge: Dynamic Handling of Reviews**
   - **Problem**: Adding reviews to each product while keeping the data structure consistent and relational.
   - **Solution**: Created a separate Review model and referenced it in the Product model. This approach allows for a flexible, scalable way of handling reviews, enabling each product to have an array of reviews that can be populated dynamically.
2. **Challenge: Error Handling for Non-Existent Products**
   - **Problem**: When querying for a product that doesn't exist
   - **Solution**: Implemented checks within the route functions to return a 404 Not Found status and an error message when the requested product ID is invalid.
3. **Challenge: Fetching Products with Reviews**
   - **Problem**: Fetching all products with their associated reviews.
   - **Solution**: Used Mongoose's populate method to retrieve and merge related review data in a single query.
4. **Challenge: Frontend Integration**
   - **Problem**: Cross-Origin Resource Sharing (CORS) issues arose when the frontend (on a different port) attempted to access backend resources.
   - **Solution**: Configured CORS middleware to allow requests from the frontend's URL (http://localhost:5173). This setup allowed the frontend to interact with the backend while maintaining security by restricting access to specific origins only.

How to set up and run?

1. **Clone the Repository**: Run the following command to clone the repository:

   ```
   git clone https://github.com/janmesh1814/Briskk.git
   ```

2. **Change to the Product Directory**: Navigate to the cloned repository's product directory:

   ```
   cd product
   ```

3. **Set Up the Backend**: Change to the backend directory and install the necessary dependencies:

   ```
   cd backend
   npm install express mongoose cors dotenv body-parser nodemon
   ```

4. **Set Up the Frontend**: Change back to the frontend directory and install the required packages:

   ```
   cd ..
   cd frontend
   npm install react-router-dom axios dotenv
   ```

5. **Run the Application**: Open two terminal windows to run the frontend and backend simultaneously:

   - **In the first terminal**, navigate to the frontend directory and start the React application:

     ```
     cd frontend
     npm run dev
     ```

   - **In the second terminal**, navigate to the backend directory and start the Node.js server using nodemon:

     ```
     cd backend
     nodemon app.js
     ```

Now, your Briskk application should be up and running, and you can access it through your web browser.