**Full Stack Development with MERN**

**1. Introduction**

* **Project Title:** **E-Commerce-application using MERN stack**
* **Team Members:** 1) **SANTHOSH.T**(PROJECT LEADER)

2) **HARISH.N**(DEVELOPER-1)

3) **MOHAMMED RIZWAN.F**(DEVELOPER-2)

4) **SURYA PRAKSH.G**(EXECUTION OF PROGRAM)

**2. Project Overview**

* **Purpose:** This project incorporates modern web development practices to deliver a scalable, secure, and responsive application. It highlights the power of the MERN stack in solving real-world problems.
* **Features:** The application boasts a range of features catering to both customers and admins:.

**FURTHER INTRODUCTION OF THE PROGRAM:**

**The key objectives of the E-Commerce Application include:**

**Providing a Platform for Businesses: Enable businesses to list and sell their products online without requiring extensive technical expertise.**

**Enhancing Customer Experience: Deliver a user-friendly interface for browsing, searching, and purchasing products.**

**Ensuring Security: Protect user data and transactions through secure login systems and payment gateways.**

**Scalability and Responsiveness: Build an application that performs efficiently on all devices and can scale with increasing user demand.**

**Efficient Management: Provide admins with tools for managing inventory, orders, and analytics.**

**3. Architecture**

## Frontend: Available Scripts

In the project directory, you can run:

**npm start**

Runs the app in the development mode.  
Open [http://localhost:3000](http://localhost:3000/) to view it in your browser.

The page will reload when you make changes.  
You may also see any lint errors in the console.

**npm test**

Launches the test runner in the interactive watch mode.  
See the section about [running tests](https://facebook.github.io/create-react-app/docs/running-tests) for more information.

**npm run build**

Builds the app for production to the build folder.  
It correctly bundles React in production mode and optimizes the build for the best performance.

The build is minified and the filenames include the hashes.  
Your app is ready to be deployed!

See the section about [deployment](https://facebook.github.io/create-react-app/docs/deployment) for more information.

**npm run eject**

**Note: this is a one-way operation. Once you eject, you can't go back!**

If you aren't satisfied with the build tool and configuration choices, you can eject at any time. This command will remove the single build dependency from your project.

Instead, it will copy all the configuration files and the transitive dependencies (webpack, Babel, ESLint, etc) right into your project so you have full control over them. All of the commands except eject will still work, but they will point to the copied scripts so you can tweak them. At this point you're on your own.

You don't have to ever use eject. The curated feature set is suitable for small and middle deployments, and you shouldn't feel obligated to use this feature. However we understand that this tool wouldn't be useful if you couldn't customize it when you are ready for it.

1. **Backend:** Functional and Non-Functional Requirements 2.1 Functional Requirements:

User Registration and Authentication:

* + Users should be able to register on the website with a unique username and password.
  + Registered users should be able to log in and access their personalized accounts.

Product Catalog:

* + The website should display a comprehensive catalog of products with details such as name, description, price, and images.
  + Products should be categorized and easily searchable to facilitate user navigation.

Shopping Cart and Checkout:

* + Users should be able to add products to their shopping cart and review cart contents before proceeding to checkout.
  + During the checkout process, users should provide shipping and billing information and choose a payment method.

Order Management:

* + The website should allow users to view their order history and track the status of current orders.
  + Admin users should have the ability to manage and process orders, update their status, and generate invoices.

Payment Gateway Integration:

* + The website should integrate with a secure payment gateway to process online transactions.
  + Users should have multiple payment options, such as credit/debit cards, PayPal, or other relevant payment methods.

Customer Reviews and Ratings:

* + Users should be able to leave reviews and ratings for products they have purchased.
  + Average product ratings should be displayed to assist other users in making purchase decisions.

Product Recommendations:

* + The website should provide personalized product recommendations based on user browsing and purchase history.
  + Recommended products should appear on the homepage or product pages.

Wishlist and Favorites:

* + Users should have the option to add products to their wishlist or favorites list for future reference.
  + Wishlist items should be easily managed and shareable.

Inventory Management:

* + The website should track product inventory and display availability to users.
  + Users should be notified when a product is out of stock or low in quantity.

Order Confirmation and Email Notifications:

* + Users should receive an order confirmation email after completing a purchase.
  + Email notifications should be sent for order updates, such as shipping and delivery status.. 2.2 Non-Functional Requirements

Performance:

* + The website should load quickly, with minimal page load times, to enhance user experience.
  + It should be able to handle a large number of concurrent users without significant performance degradation.

**Database:** The database in a MERN (MongoDB, Express, React, Node.js) stack e-commerce project is typically managed using **MongoDB**, a NoSQL database. It is designed to handle and store data in a flexible, document-oriented structure, allowing for seamless integration with the other parts of the MERN stack.

Here’s a detailed description of the **database structure**:

### ****Key Collections and Their Structures****

#### 1. **Users Collection**

This collection manages the data of the platform's users, including customers and admins.

* **Schema Fields**:
  + name: Full name of the user (string, required).
  + email: Unique email ID (string, required, unique).
  + password: Hashed password for security (string, required).
  + isAdmin: Boolean flag to differentiate admins from regular users (default: false).
  + createdAt: Timestamp of when the user registered (date, default: current timestamp).
  + updatedAt: Timestamp of the last update to the user’s record (date).

#### 2. **Products Collection**

Stores information about the products listed on the e-commerce platform.

* **Schema Fields**:
  + name: Name of the product (string, required).
  + description: Detailed description of the product (string).
  + price: Price of the product (number, required).
  + brand: Brand or manufacturer (string).
  + category: Category to which the product belongs (string, required).
  + stock: Available quantity in inventory (number, required).
  + ratings: Average user rating (number, default: 0).
  + reviews: Array of reviews (see Reviews schema below).
  + createdAt: Timestamp of when the product was added (date).

#### 3. **Reviews Sub-Schema**

Nested schema for managing product reviews.

* **Schema Fields**:
  + userId: Reference to the Users collection for reviewer identification (ObjectId).
  + rating: User's rating for the product (number, 1-5).
  + comment: Optional review text (string).
  + createdAt: Timestamp of when the review was added (date).

#### 4. **Orders Collection**

Tracks user orders and their fulfillment statuses.

* **Schema Fields**:
  + userId: Reference to the Users collection (ObjectId, required).
  + orderItems: Array of ordered items:
    - productId: Reference to the Products collection (ObjectId, required).
    - quantity: Quantity of the product ordered (number, required).
    - price: Price at the time of the order (number, required).
  + shippingAddress: Object containing:
    - street: Street address (string).
    - city: City (string).
    - postalCode: Postal or ZIP code (string).
    - country: Country (string).
  + paymentMethod: Payment method used (e.g., PayPal, Credit Card) (string, required).
  + paymentStatus: Status of the payment (e.g., Pending, Completed) (string, default: "Pending").
  + orderStatus: Current order status (e.g., Processing, Shipped, Delivered) (string, default: "Processing").
  + createdAt: Timestamp of order placement (date).

#### 5. **Categories Collection** (Optional)

Stores a hierarchical structure of product categories.

* **Schema Fields**:
  + name: Category name (string, required, unique).
  + description: Brief description of the category (string).
  + parentCategoryId: Reference to a parent category (if applicable, for subcategories).

#### 6. **Cart Collection** (Optional)

Temporarily stores items added to the user’s cart before checkout.

* **Schema Fields**:
  + userId: Reference to the Users collection (ObjectId, required).
  + cartItems: Array of items:
    - productId: Reference to the Products collection (ObjectId, required).
    - quantity: Quantity of the item (number, required).

#### 7. **Payment Collection** (Optional)

Tracks payment details for each transaction.

* **Schema Fields**:
  + orderId: Reference to the Orders collection (ObjectId, required).
  + paymentStatus: Status of the payment (e.g., Success, Failed) (string).
  + paymentDetails: Object storing additional data from the payment gateway.

### ****Relationships****

* **Users ↔ Orders**: One-to-many (a user can place multiple orders).
* **Orders ↔ Products**: Many-to-many (orders can contain multiple products, and products can appear in multiple orders).
* **Products ↔ Categories**: Many-to-one (a product belongs to one category).
* **Users ↔ Reviews**: One-to-many (a user can write multiple reviews).

### ****Features Supported by Database Design****

1. **Scalability**: MongoDB's document-based storage ensures the database can scale horizontally.
2. **Flexibility**: Schemaless design allows easy addition of new fields or collections as requirements evolve.
3. **Fast Querying**: Indexing on fields like email, productId, and orderId ensures efficient querying.
4. **Secure Storage**: Passwords are hashed, and sensitive data is encrypted as needed.

By structuring the database as described, the e-commerce platform can effectively manage users, products, orders, and other key components while ensuring performance, reliability, and scalability.

4) **. Folder Structure**

### ****1. React Frontend (Client)****

The React frontend is structured to facilitate modularity, maintainability, and scalability. Below is a typical folder structure:

bash

Copy code

client/

├── public/

│ ├── index.html # Main HTML file for the React app

│ ├── favicon.ico # App favicon

│ └── assets/ # Static files (e.g., images, fonts)

├── src/

│ ├── components/ # Reusable UI components

│ │ ├── Navbar.js

│ │ ├── Footer.js

│ │ └── ProductCard.js

│ ├── pages/ # Pages corresponding to routes

│ │ ├── HomePage.js

│ │ ├── ProductPage.js

│ │ ├── CartPage.js

│ │ └── CheckoutPage.js

│ ├── redux/ # Redux-related files for state management

│ │ ├── store.js # Configures the Redux store

│ │ ├── slices/ # Individual slices of the state

│ │ │ ├── cartSlice.js

│ │ │ └── productSlice.js

│ ├── services/ # API interaction logic

│ │ └── api.js # Axios or Fetch API setup and functions

│ ├── utils/ # Utility functions (e.g., formatters, validators)

│ ├── App.js # Root component defining routes

│ ├── index.js # Entry point of the React app

│ └── styles/ # Global and modular CSS/SCSS files

│ ├── App.css

│ ├── variables.scss

│ └── mixins.scss

├── package.json # Dependencies and scripts for the React app

├── .env # Environment variables for the frontend

└── README.md # Documentation for the frontend

#### **Key Highlights**

* **Component-based structure**: UI elements are encapsulated in the components/ folder for reuse across multiple pages.
* **State management**: Redux (or Context API) is used for managing application-wide states like cart items or user authentication.
* **API interaction**: The services/api.js file handles all HTTP requests to the backend, keeping API logic separate from the components.
* **Styling**: Centralized in the styles/ folder, ensuring consistency across the application.

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

The backend is structured to handle routing, controllers, models, and middleware. Below is a typical folder structure:

bash

Copy code

server/

├── config/

│ ├── db.js # Database connection setup

│ ├── dotenv.js # Environment variable configuration

│ └── keys.js # API keys and secrets

├── controllers/

│ ├── authController.js # Handles user authentication

│ ├── productController.js # Handles product-related logic

│ └── orderController.js # Manages order processing

├── middleware/

│ ├── authMiddleware.js # Protects routes requiring authentication

│ └── errorHandler.js # Centralized error handling middleware

├── models/

│ ├── User.js # Schema for the users collection

│ ├── Product.js # Schema for the products collection

│ └── Order.js # Schema for the orders collection

├── routes/

│ ├── authRoutes.js # Routes for authentication

│ ├── productRoutes.js # Routes for products

│ └── orderRoutes.js # Routes for orders

├── utils/

│ ├── emailHelper.js # Functions to send emails

│ ├── tokenHelper.js # Functions to generate/validate JWTs

│ └── logger.js # Logging utility

├── server.js # Entry point for the server

├── package.json # Dependencies and scripts for the backend

├── .env # Environment variables for the server

└── README.md # Documentation for the backend

#### **Key Highlights**

* **Separation of Concerns**:
  + **controllers/**: Contains logic for handling requests and responses.
  + **routes/**: Defines RESTful API endpoints, delegating logic to controllers.
  + **models/**: Contains Mongoose schemas for MongoDB collections.
* **Middleware**: The middleware/ folder contains reusable functions for authentication, logging, and error handling.
* **Environment Configuration**: Environment variables (e.g., database URIs, API keys) are centralized in .env and accessed via the config/ folder.
* **Modular API Design**: Routes are grouped by functionality (e.g., authRoutes.js, productRoutes.js).

### ****Workflow Integration****

* **Frontend** makes HTTP requests (e.g., using Axios or Fetch) to the API endpoints defined in the backend.
* **Backend** processes requests, interacts with the MongoDB database, and responds with JSON data.
* The modular structure ensures that the **frontend** and **backend** can be developed, tested, and deployed independently.

5) **Running the Application**

**Step1:**

**<!DOCTYPE html>**

**<html lang="en">**

**<head>**

**<meta charset="UTF-8" />**

**<meta name="viewport" content="width=device-width, initial-scale=1.0" />**

**<meta http-equiv="X-UA-Compatible" content="ie=edge" />**

**<meta name="Description" content="Enter your description here" />**

**<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/5.1.0/css/bootstrap.min.css" />**

**<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.15.4/css/all.min.css" />**

**<link rel="preconnect" href="https://fonts.googleapis.com" />**

**<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin />**

**<link href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;500;700;900&display=swap" rel="stylesheet" />**

**<title>Admin</title>**

**</head>**

**<body>**

**<div id="root"></div>**

**<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/2.9.2/umd/popper.min.js"></script>**

**<script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/5.1.0/js/bootstrap.min.js"></script>**

**</body>**

**</html>**

**Step2:**

**{**

**"short\_name": "React App",**

**"name": "Create React App Sample",**

**"icons": [**

**{**

**"src": "favicon.ico",**

**"sizes": "64x64 32x32 24x24 16x16",**

**"type": "image/x-icon"**

**},**

**{**

**"src": "logo192.png",**

**"type": "image/png",**

**"sizes": "192x192"**

**},**

**{**

**"src": "logo512.png",**

**"type": "image/png",**

**"sizes": "512x512"**

**}**

**],**

**"start\_url": ".",**

**"display": "standalone",**

**"theme\_color": "#000000",**

**"background\_color": "#ffffff"**

**}**

**Step3:**

  "name": "admin",

  "version": "0.1.0",

  "private": true,

  "dependencies": {

    "@ant-design/plots": "^1.2.5",

    "@reduxjs/toolkit": "^1.9.5",

    "@testing-library/jest-dom": "^5.16.5",

    "@testing-library/react": "^13.4.0",

    "@testing-library/user-event": "^13.5.0",

    "antd": "^5.6.1",

    "axios": "^1.4.0",

    "formik": "^2.4.2",

    "react": "^18.2.0",

    "react-dom": "^18.2.0",

    "react-dropzone": "^14.2.3",

    "react-form-stepper": "^2.0.3",

    "react-icons": "^4.9.0",

    "react-multi-select-component": "^4.3.4",

    "react-quill": "^2.0.0",

    "react-redux": "^8.1.0",

    "react-router-dom": "^6.13.0",

    "react-scripts": "5.0.1",

    "react-select": "^5.7.3",

    "react-toastify": "^9.1.3",

    "react-widgets": "^5.8.4",

    "web-vitals": "^2.1.4",

    "yup": "^1.2.0"

  },

  "scripts": {

    "start": "react-scripts start",

    "build": "react-scripts build",

    "test": "react-scripts test",

    "eject": "react-scripts eject"

  },

  "eslintConfig": {

    "extends": [

      "react-app",

      "react-app/jest"

    ]

  },

  "browserslist": {

    "production": [

      ">0.2%",

      "not dead",

      "not op\_mini all"

    ],

    "development": [

      "last 1 chrome version",

      "last 1 firefox version",

      "last 1 safari version"

    ]

  }

}

**Step4:**

**abel/helper-plugin-utils": "^7.14.5"**

**},**

**"engines": {**

**"node": ">=6.9.0"**

**},**

**"peerDependencies": {**

**"@babel/core": "^7.0.0-0"**

**}**

**},**

**"node\_modules/@babel/plugin-syntax-decorators": {**

**"version": "7.22.5",**

**Output picture:**













