Project Documentation

Project Overview

This full-stack eCommerce project consists of three main components: an Admin Frontend, an Ecommerce Frontend, and a Backend. It is designed to provide a comprehensive solution for online retail, allowing administrators to manage products, orders, and users, while offering customers a seamless shopping experience.

Technology Stack

This project leverages a modern and robust technology stack, ensuring scalability, maintainability, and a rich user experience across all its components.

Admin Frontend (React.js)

The administrative interface is built using React.js, a declarative, component-based JavaScript library for building user interfaces. It provides a dynamic and responsive dashboard for managing the eCommerce operations.

- **Framework:** React.js (version 19.1.0)
- **Routing:** React Router DOM (version 7.5.0) for declarative navigation.
- **API Communication:** Axios (version 1.8.4) for making HTTP requests to the backend.
- **State Management:** Local component state and React Context API (implicit, as no dedicated state management library like Redux or Zustand was found in `package.json`).
- **UI Components:** Custom components, supplemented by `react-icons` (version 5.5.0) for vector icons.
- **Data Handling:** `papaparse` (version 5.5.2) for CSV parsing, `react-csv` (version 2.2.2) for CSV export, `react-chartjs-2` (version 5.3.0) for data visualization,

`sharp` (version 0.34.1) for image processing (likely used for client-side image manipulation before upload).

- **Notifications:** `react-toastify` (version 11.0.5) for displaying user feedback and notifications.
- **Authentication Utilities:** `jwt-decode` (version 4.0.0) for decoding JSON Web Tokens on the client-side.

Ecommerce Frontend (React.js)

The customer-facing storefront is also developed with React.js, focusing on an intuitive and engaging shopping journey. It integrates various libraries to enhance user interaction and provide essential eCommerce functionalities.

- **Framework:** React.js (version 19.0.0)
- **Routing:** React Router DOM (version 7.2.0) for client-side routing.
- **API Communication:** Axios (version 1.8.2) for backend API interactions.
- **State Management:** React Context API (`AuthContext` , `CartContext` , `ProductContext`) for global state management related to user authentication, shopping cart, and product data.
- **UI Framework & Styling:** Bootstrap (version 5.3.3) via `react-bootstrap` (version 2.10.9) for responsive design and pre-built UI components. Font Awesome (`@fortawesome/fontawesome-free` version 6.7.2) for scalable vector icons.
- **Animations:** `framer-motion` (version 12.5.0) for smooth UI animations and transitions.
- **Payment Integration:** Stripe (`@stripe/stripe-js` version 6.0.0) for secure client-side payment collection.
- **Notifications:** `react-toastify` (version 11.0.5) for user feedback.
- **Authentication Utilities:** `jwt-decode` (version 4.0.0) for JWT parsing.
- **Emailing (Client-side):** `nodemailer` (version 6.10.0) (Note: While `nodemailer` is typically a backend library, its presence here might indicate client-side email triggering or a misconfiguration. Further investigation would be needed if this causes issues).

Backend (Node.js + Express.js + MongoDB)

The backend serves as the core of the application, handling all business logic, data persistence, and API services. It is built on Node.js with the Express.js framework and uses MongoDB as its primary database.

- **Runtime Environment:** Node.js
- **Web Framework:** Express.js (version 4.21.2) for building robust RESTful APIs.
- **Database:** MongoDB (via `mongodb` driver version 6.10.3) as the NoSQL database.
- **Object Data Modeling (ODM):** Mongoose (version 8.14.0) for elegant MongoDB object modeling.
- **Authentication:** `bcrypt` (version 5.1.1) and `bcryptjs` (version 3.0.2) for password hashing and comparison. `jsonwebtoken` (version 9.0.2) for generating and verifying JSON Web Tokens.
- **API Utilities:** `cors` (version 2.8.5) for enabling Cross-Origin Resource Sharing. `body-parser` (version 1.20.3) for parsing incoming request bodies.
- **File Uploads & Cloud Storage:** `multer` (version 1.4.5-lts.1) for handling multipart/form-data (file uploads). `cloudinary` (version 2.6.0) and `sharp` (version 0.34.1) for cloud-based image storage and manipulation.
- **Emailing:** `nodemailer` (version 6.10.0) for sending transactional emails.
- **Payment Integration:** Stripe (version 17.7.0) for server-side payment processing and webhook handling.
- **Environment Variables:** `dotenv` (version 16.5.0) for loading environment variables from a `.env` file.
- **Development Utilities:** `nodemon` (version 3.1.9) for automatic server restarts during development.

Folder Structure

Admin Frontend

```
admin-frontend/
├── public/
├── src/
│   ├── assets/
│   ├── components/
│   ├── context/
│   ├── pages/
│   ├── styles/
│   ├── App.js
│   ├── index.js
│   └── ... (other files)
├── package.json
├── package-lock.json
└── README.md
```

Ecommerce Frontend

```
ecommerce-frontend/
├── public/
├── src/
│   ├── assets/
│   ├── components/
│   ├── context/
│   ├── pages/
│   ├── styles/
│   ├── utils/
│   ├── App.js
│   ├── index.js
│   └── ... (other files)
├── package.json
├── package-lock.json
└── README.md
```

Backend

```
ecommerce-backend/
├── config/
├── controllers/
├── middleware/
├── models/
├── routes/
├── uploads/
│   ├── products/
│   └── avatars/
├── utils/
├── server.js
├── package.json
├── package-lock.json
└── README.md
```

Installation & Setup Instructions

To set up and run this project locally, follow these steps:

Prerequisites

- Node.js (LTS version recommended)
- npm (Node Package Manager) or Yarn
- MongoDB (Community Server)

General Setup

1. **Clone the repository:**

   `bash git clone <repository_url> cd REACT-Ecommerce-main`

2. **Install dependencies for each component:**

   Navigate into each project directory (`admin-frontend`, `ecommerce-frontend`, `ecommerce-backend`) and install the dependencies:

   ```bash cd admin-frontend npm install

   cd ../ecommerce-frontend npm install

   cd ../ecommerce-backend npm install ```
   ```

Backend Setup

1. **Environment Variables:**

   Create a `.env` file in the `ecommerce-backend` directory with the following variables:

   ```
   PORT=5000 MONGO_URI=mongodb://localhost:27017/ecommerce
   JWT_SECRET=your_jwt_secret_key
   STRIPE_SECRET_KEY=your_stripe_secret_key
   CLOUDINARY_CLOUD_NAME=your_cloudinary_cloud_name
   CLOUDINARY_API_KEY=your_cloudinary_api_key
   CLOUDINARY_API_SECRET=your_cloudinary_api_secret
   EMAIL_USER=your_email@example.com EMAIL_PASS=your_email_password
   ```

   Replace placeholder values with your actual credentials.

2. **Start the Backend Server:**

   ```
   bash cd ecommerce-backend npm start
   ```

   The backend server will start on `http://localhost:5000` (or the port you specified).

Admin Frontend Setup

1. **API Endpoint Configuration:**

   Ensure your admin frontend is configured to connect to your backend API. This is typically done in an environment file or a configuration file within the `admin-frontend` project. (Further details require code analysis).

2. **Start the Admin Frontend:**

   ```
   bash cd admin-frontend npm start
   ```

   The admin frontend will typically open in your browser at `http://localhost:3000`.

Ecommerce Frontend Setup

1. **API Endpoint Configuration:**

   Similar to the admin frontend, ensure the ecommerce frontend is configured to connect to your backend API. (Further details require code analysis).

2. **Start the Ecommerce Frontend:**

   `bash cd ecommerce-frontend npm start`

   The ecommerce frontend will typically open in your browser at `http://localhost:3001` (or another available port).

Features Module-wise

Admin Features

- **Dashboard:** Overview of key metrics (sales, orders, users, products).
- **Product Management:** Add, edit, delete, and view products. Manage product details, images, categories, and stock.
- **Order Management:** View, update status, and manage customer orders.
- **User Management:** View, edit, and manage user accounts and roles.
- **Analytics & Reporting:** Generate reports on sales, product performance, and user activity.
- **Category Management:** Create, update, and delete product categories.
- **Review Management:** Moderate and manage product reviews.

User Features (Ecommerce Frontend)

- **Product Listing & Search:** Browse products by category, search for products, and view product details.
- **Shopping Cart:** Add, remove, and update quantities of items in the cart.
- **Wishlist:** Save products for later purchase.
- **User Authentication:** Register, login, and logout functionality.

- **User Profile Management:** View and update personal information, shipping addresses.
- **Order History:** View past orders and their statuses.
- **Checkout Process:** Secure multi-step checkout with payment integration.
- **Product Reviews & Ratings:** Submit reviews and ratings for purchased products.
- **Contact & Support:** Contact form and FAQ section.

Backend APIs Summary

The backend provides a comprehensive set of RESTful APIs, meticulously designed to support both the administrative and customer-facing frontends. These APIs adhere to REST principles, ensuring stateless, scalable, and maintainable communication. Below is a summary of key API categories and their functionalities:

- **Authentication APIs:**
    - `POST /api/auth/register`: User registration (for both regular users and admins).
    - `POST /api/auth/login`: User login, returning a JWT upon successful authentication.
    - `POST /api/auth/logout`: Invalidates the user's session (if applicable, or simply clears client-side token).
    - `POST /api/auth/forgot-password`: Initiates password reset process.
    - `PUT /api/auth/reset-password`: Resets user password with a valid token.
- **User Management APIs (Admin Only):**
    - `GET /api/users`: Retrieves a list of all users.
    - `GET /api/users/:id`: Retrieves details of a specific user.
    - `PUT /api/users/:id`: Updates user information (e.g., role, status).
    - `DELETE /api/users/:id`: Deletes a user account.
- **Product APIs:**

    - `GET /api/products`: Fetches all products, with optional filtering, sorting, and pagination.
    - `GET /api/products/:id`: Retrieves details of a single product.
    - `POST /api/products`: Creates a new product (Admin only), including image uploads.
    - `PUT /api/products/:id`: Updates an existing product (Admin only).
    - `DELETE /api/products/:id`: Deletes a product (Admin only).
- **Order APIs:**
    - `POST /api/orders`: Creates a new order.
    - `GET /api/orders/my`: Retrieves orders for the authenticated user.
    - `GET /api/orders/:id`: Retrieves details of a specific order.
    - `GET /api/orders`: Retrieves all orders (Admin only).
    - `PUT /api/orders/:id/status`: Updates the status of an order (Admin only).
- **Category APIs:**
    - `GET /api/categories`: Retrieves a list of all product categories.
    - `POST /api/categories`: Creates a new category (Admin only).
    - `PUT /api/categories/:id`: Updates an existing category (Admin only).
    - `DELETE /api/categories/:id`: Deletes a category (Admin only).
- **Review APIs:**
    - `POST /api/products/:productId/reviews`: Submits a new review for a product.
    - `GET /api/products/:productId/reviews`: Retrieves reviews for a specific product.
    - `DELETE /api/reviews/:id`: Deletes a review (Admin or review owner).
- **Payment APIs:**
    - `POST /api/payment/create-intent`: Creates a Stripe Payment Intent.

    - `POST /api/payment/webhook`: Endpoint for Stripe webhooks to handle payment status updates.

- **Dashboard/Analytics APIs (Admin Only):**
    - `GET /api/dashboard/summary`: Provides aggregated data for the admin dashboard (e.g., total sales, order count, user count).
    - `GET /api/dashboard/sales-data`: Retrieves sales data for charting and reporting.

This detailed breakdown of API endpoints ensures clarity for developers integrating with the backend and provides a comprehensive overview of the system's capabilities.

Authentication & Authorization Flow

Authentication

User authentication is handled using JSON Web Tokens (JWT). Upon successful login, the backend issues a JWT to the client (both admin and ecommerce frontends). This token is then stored client-side (e.g., in local storage or HTTP-only cookies) and sent with subsequent requests to protected routes in the `Authorization` header as a Bearer token.

Authorization

Authorization is role-based. Users are assigned roles (e.g., 'admin', 'user'). Middleware on the backend checks the user's role extracted from the JWT to determine if they have the necessary permissions to access specific routes or perform certain actions. For example, only users with the 'admin' role can access product management APIs.

Database Models Overview

The MongoDB database schema is managed using Mongoose, with the following key models:

- **User:** Represents a user of the system, including both customers and administrators.

- Fields: `username`, `email`, `password`, `role` (`user`, `admin`), `address`, `phone`, etc.
- **Product:** Stores information about products available in the store.
- Fields: `name`, `description`, `price`, `category`, `stock`, `images`, `reviews`, etc.
- **Order:** Represents a customer's order.
- Fields: `user`, `products` (array of product references with quantities), `totalAmount`, `status`, `shippingAddress`, `paymentDetails`, `createdAt`, etc.
- **Cart:** Represents a user's shopping cart.
- Fields: `user`, `items` (array of product references with quantities), `createdAt`, `updatedAt`.
- **Wishlist:** Represents a user's wishlist.
- Fields: `user`, `products` (array of product references), `createdAt`, `updatedAt`.
- **Review:** Stores product reviews submitted by users.
- Fields: `user`, `product`, `rating`, `comment`, `createdAt`.
- **Admin:** Represents an administrator user with specific privileges.
- Fields: `username`, `email`, `password`, `isAdmin` (boolean, defaults to true).

Payment Integration Details

Payment processing is integrated using **Stripe**. The backend handles the creation of payment intents and confirmation, while the frontend (Ecommerce Frontend) interacts with the Stripe API to collect payment information securely.

- **Backend:** Uses `stripe` npm package to create `PaymentIntent` objects, manage webhooks for payment status updates, and handle successful transactions.
- **Frontend:** Utilizes `@stripe/stripe-js` to securely collect card details and confirm payments on the client-side, ensuring PCI compliance.

Deployment Guidelines

Local Deployment

Follow the Installation & Setup Instructions section to deploy the project locally. Ensure all environment variables are correctly configured for your local environment.

Live Deployment

For live deployment, consider the following:

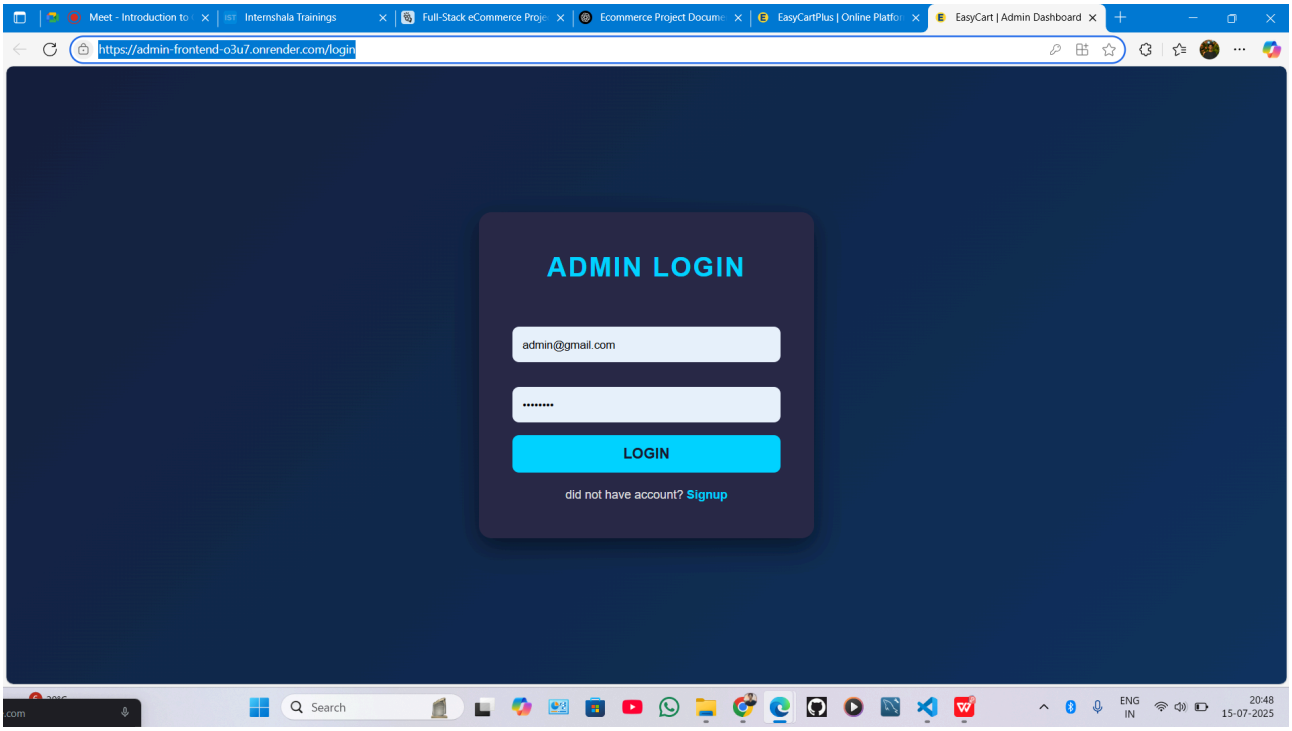
- **Backend:**
    - **Hosting:** Cloud platforms like Heroku, AWS EC2, Google Cloud Run, or DigitalOcean Droplets.
    - **Process Manager:** Use PM2 to keep the Node.js application running continuously and manage restarts.
    - **Environment Variables:** Securely configure environment variables (e.g., `MONGO_URI`, `JWT_SECRET`, `STRIPE_SECRET_KEY`, Cloudinary credentials) in your hosting environment.
    - **Database:** Use a managed MongoDB service like MongoDB Atlas for production-grade database hosting.
- **Frontend (Admin and Ecommerce):**
    - **Hosting:** Static site hosting services like Netlify, Vercel, AWS S3 with CloudFront, or Firebase Hosting.
    - **Build Process:** Run `npm run build` in each frontend directory to create optimized production builds. These static files can then be deployed.
    - **Environment Variables:** Configure environment variables (e.g., API endpoint URLs) during the build process or at runtime, depending on the hosting service.
- **Domain & SSL:** Configure a custom domain and enable SSL/TLS certificates for secure communication (HTTPS).
- **CI/CD:** Implement Continuous Integration/Continuous Deployment pipelines (e.g., using GitHub Actions, GitLab CI, Jenkins) for automated testing and

deployment.

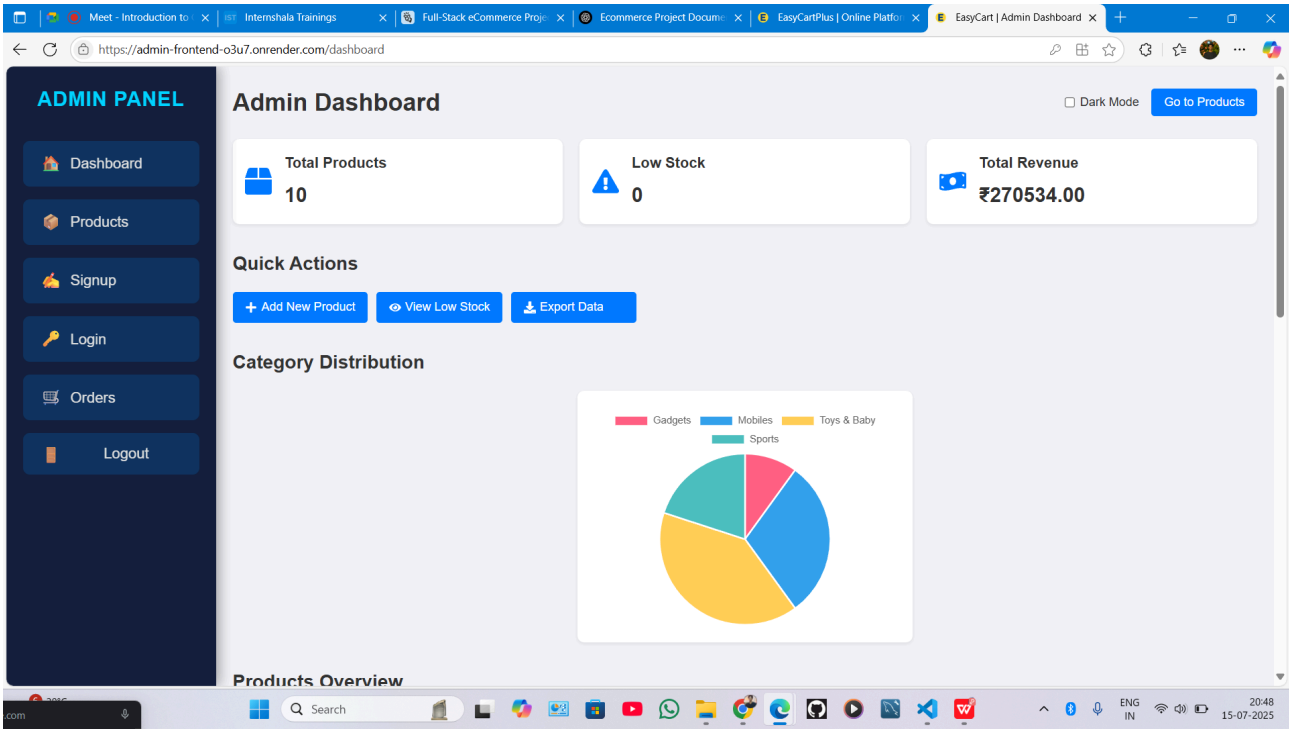
Screenshots or UI Summary

Admin Frontend

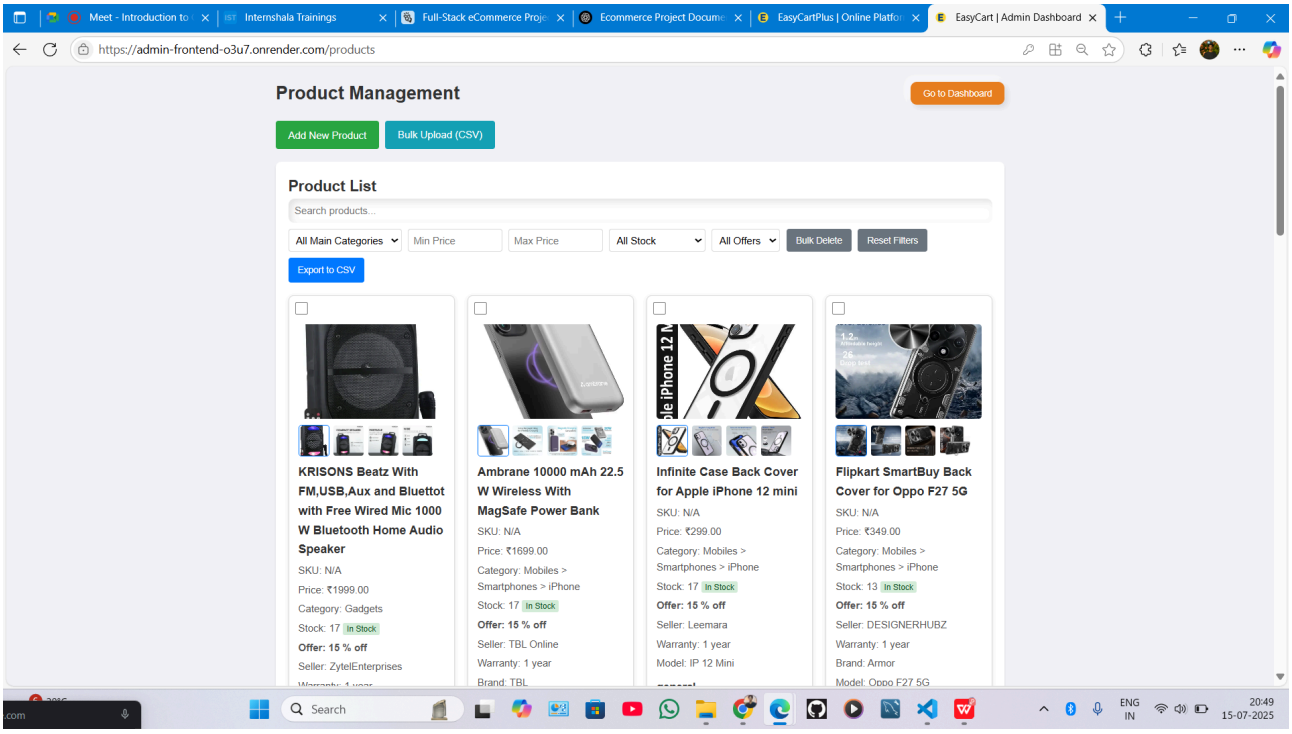
- **Dashboard:** Clean and intuitive layout with charts and graphs displaying key metrics (e.g., total sales, number of orders, active users, product stock levels). Navigation sidebar for quick access to different management sections.



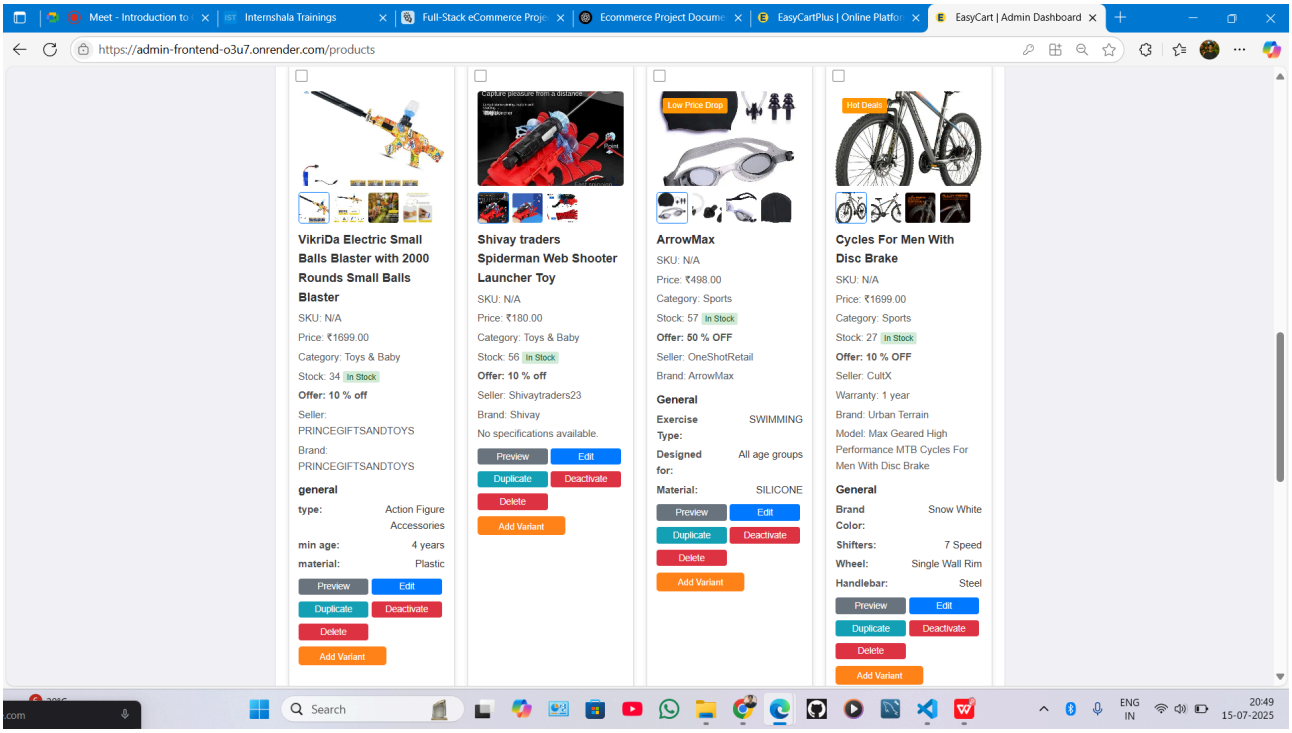
- **Product Management:** Table-based view of products with options to add new products, edit existing ones, and delete. Forms for product details will include fields for name, description, price, stock, category selection, and image uploads.



- **Order Management:** List of orders with filters for status, date, and customer. Each order detail page will show customer information, ordered items, total amount, and options to update order status.

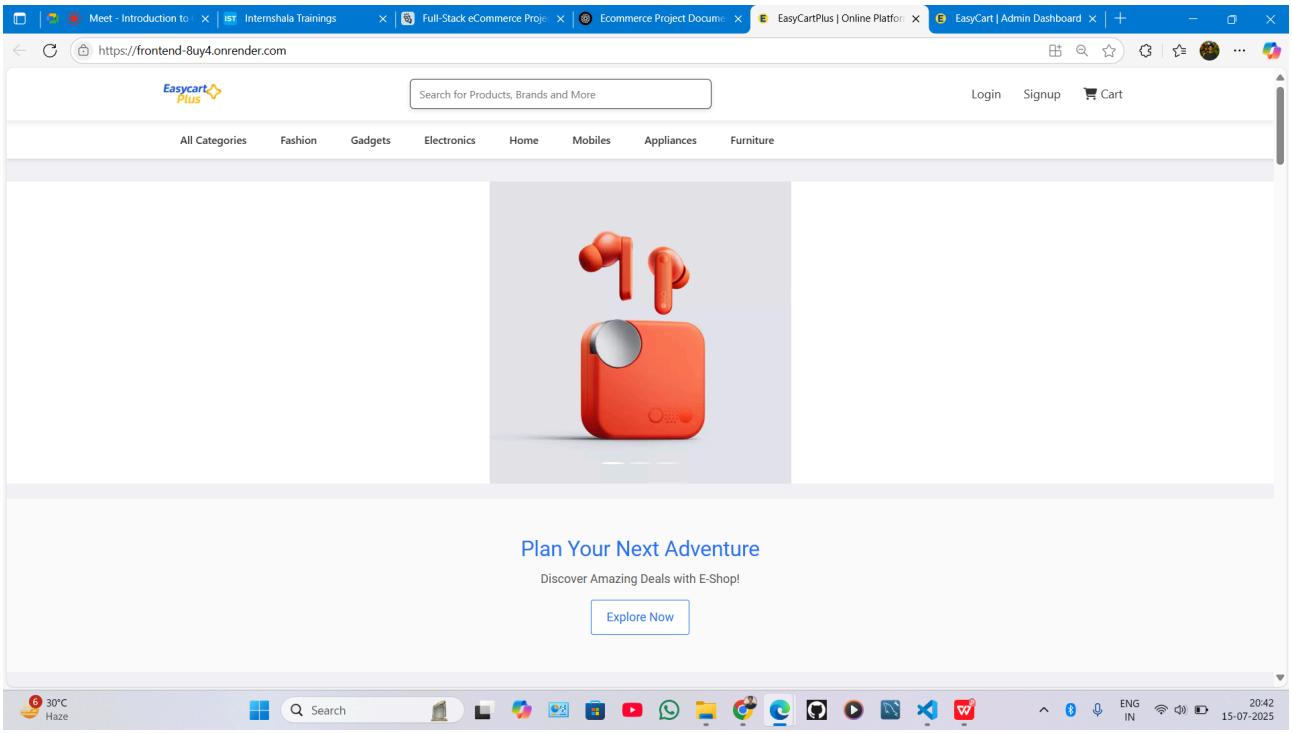


- **User Management:** Table of registered users with options to view details, edit roles, or deactivate accounts.

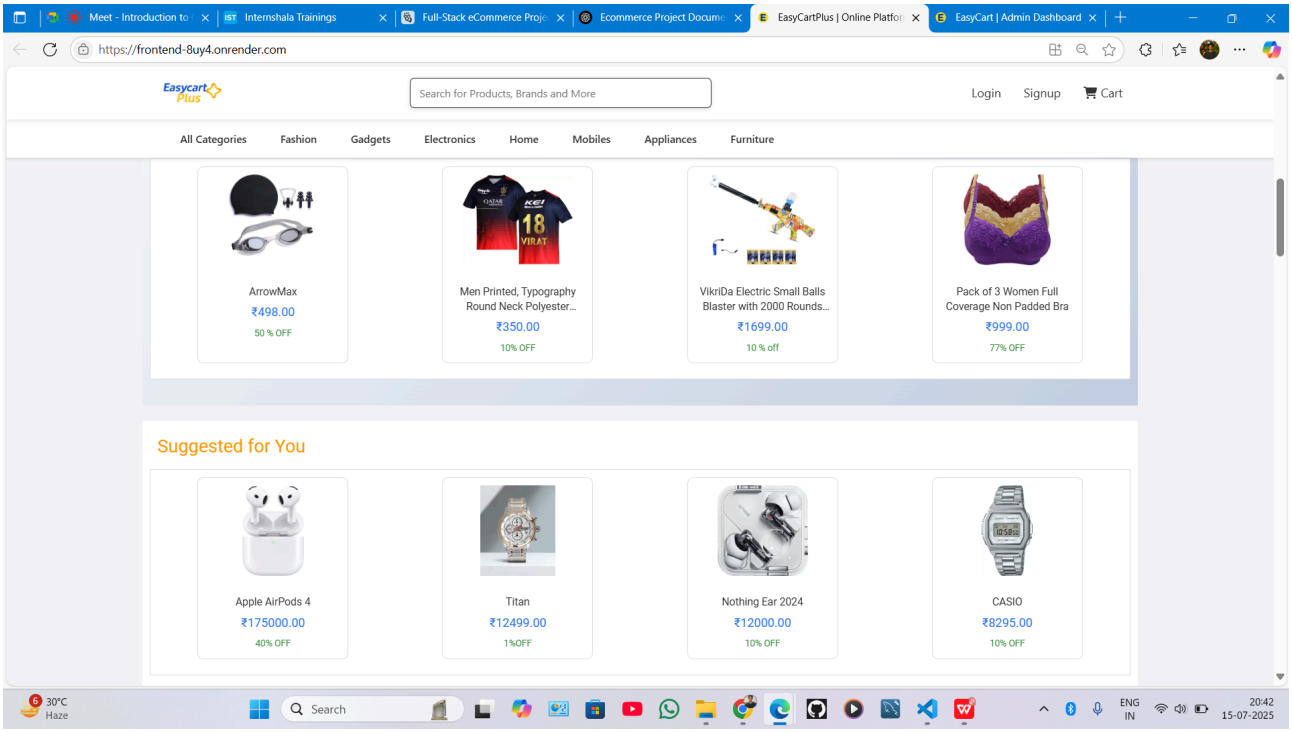


Ecommerce Frontend

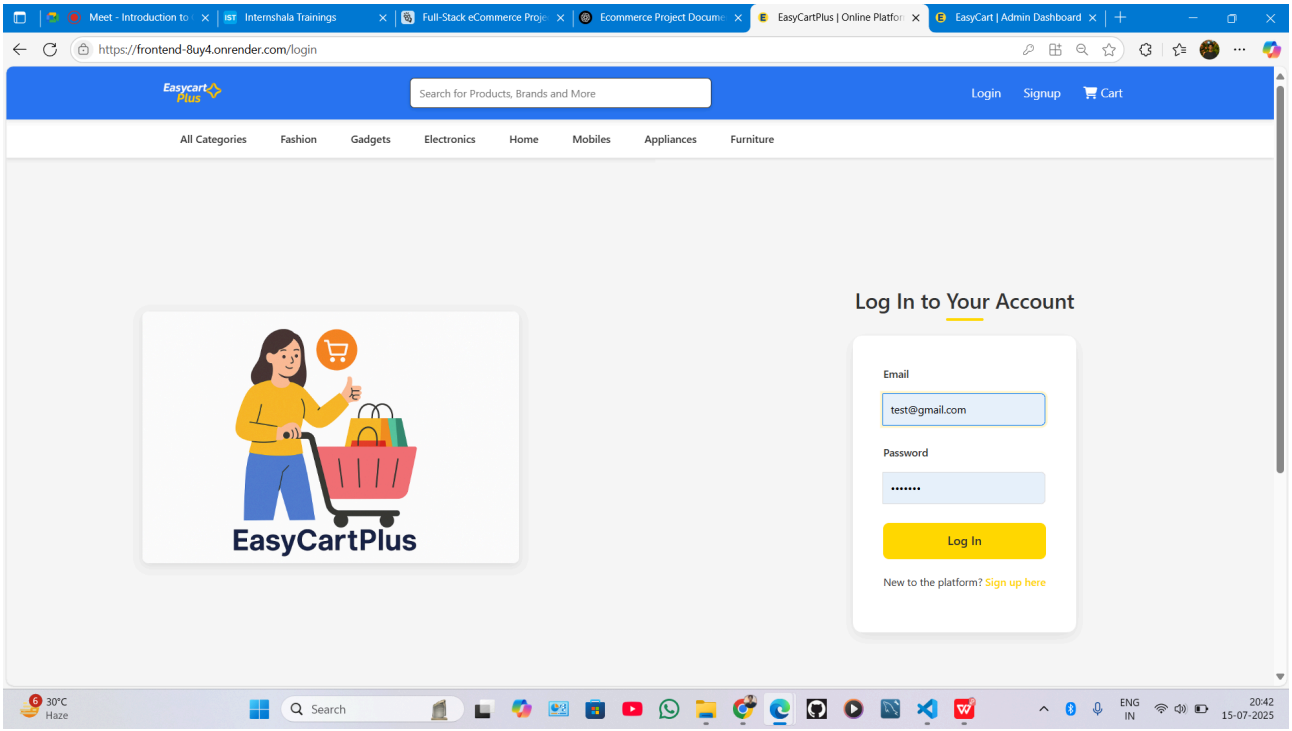
- **Homepage:** Visually appealing layout featuring new arrivals, popular products, categories, and promotional banners. Clear navigation bar with links to categories, cart, wishlist, and user profile.



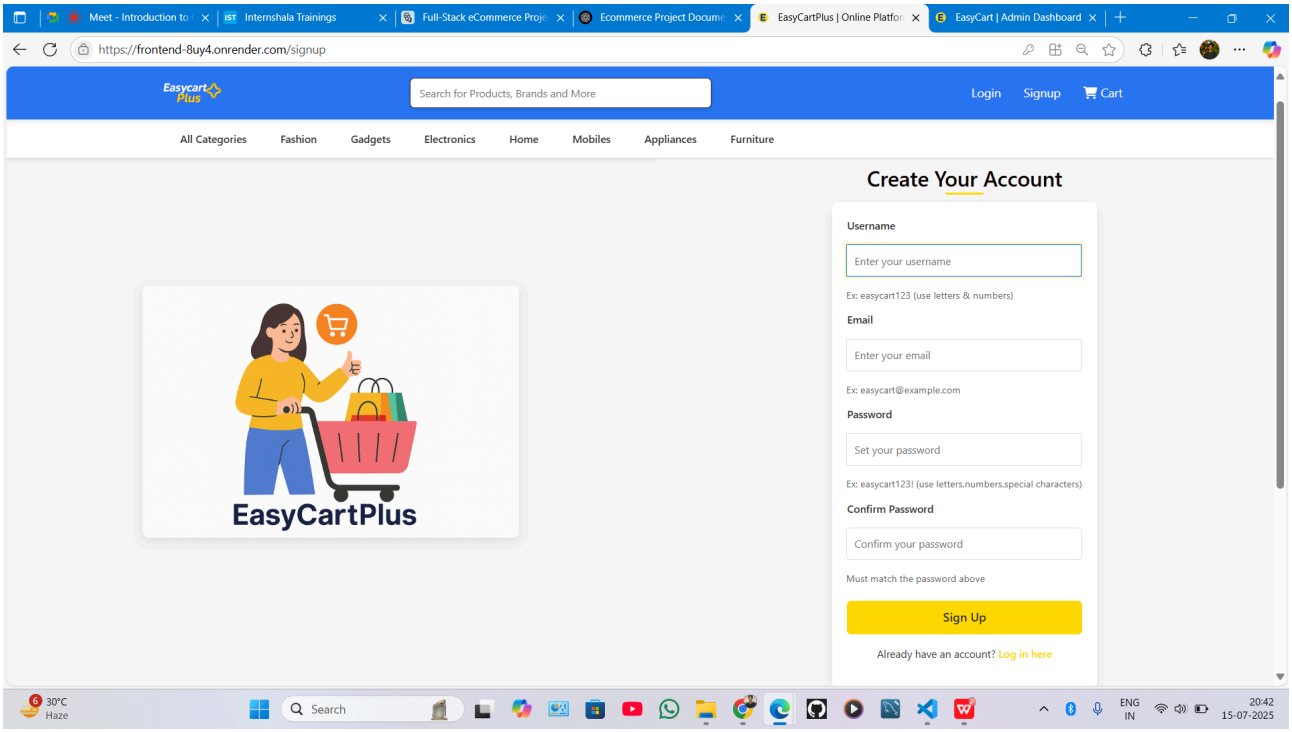
- **Product Listing Page:** Grid or list view of products with filtering and sorting options (e.g., by price, category, popularity). Pagination for large product sets.



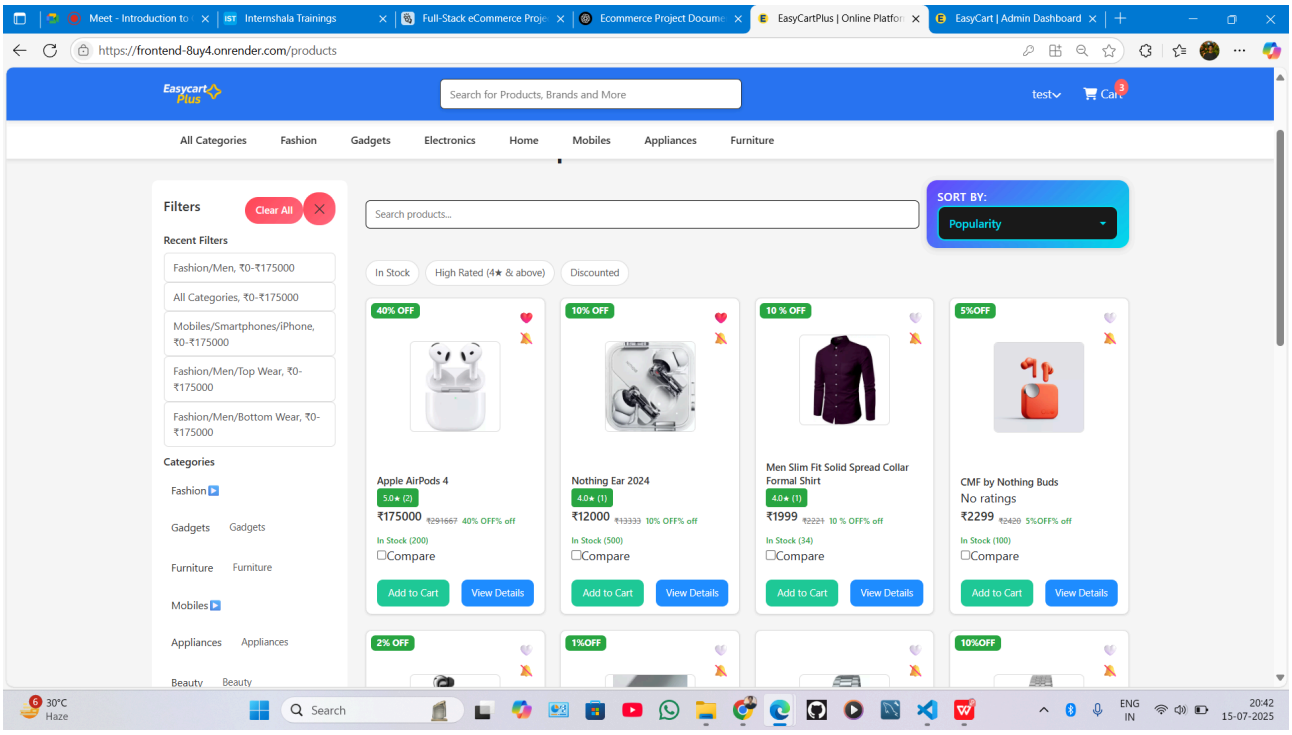
- **Product Detail Page:** Dedicated page for each product with high-resolution images, detailed description, price, add-to-cart button, quantity selector, and customer reviews section.



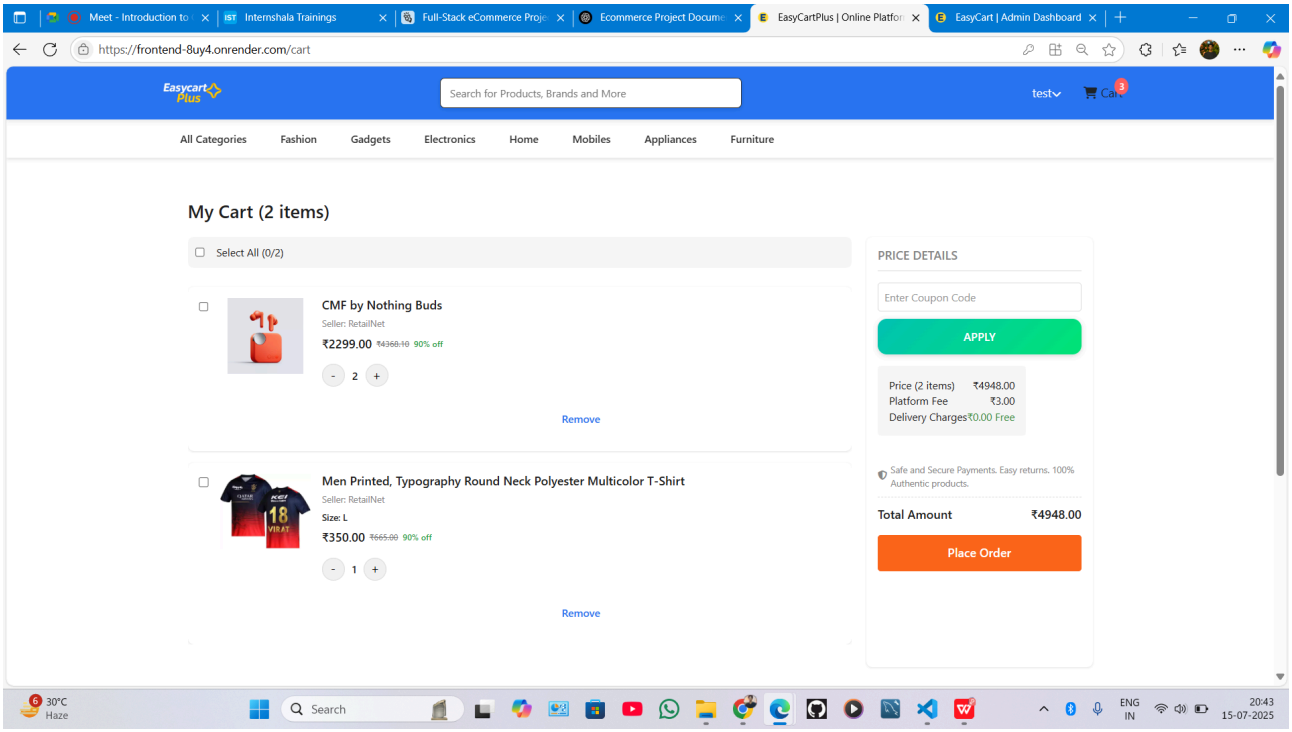
- **Shopping Cart:** Clear display of items in the cart with quantities, prices, and subtotal. Options to update quantities or remove items. Prominent checkout button.



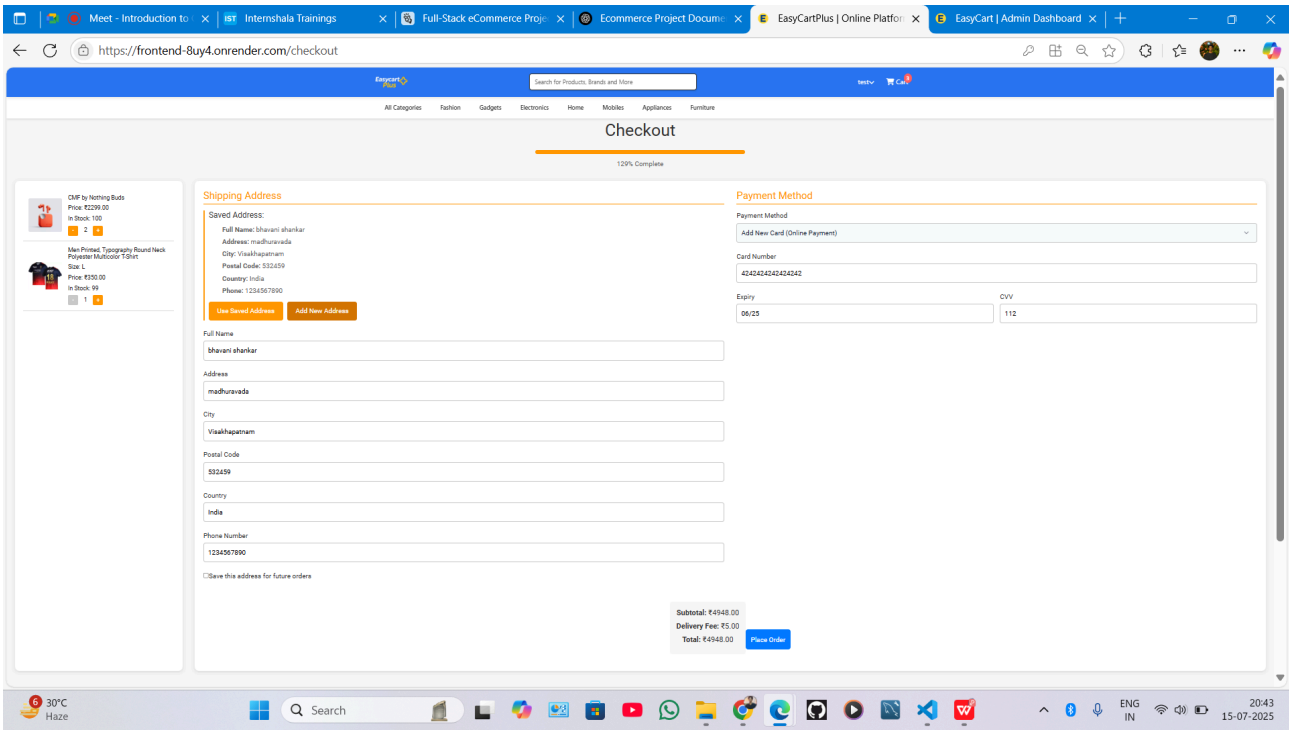
- **Checkout Process:** Multi-step form for shipping address, payment information (Stripe integration), and order review.



- **User Profile:** Dashboard for users to view and update their personal information, manage addresses, and access order history.



- **Order History:** List of past orders with links to detailed order pages.



Best Practices Followed

- **Modular Architecture:** The project is divided into distinct frontend and backend components, promoting separation of concerns, easier maintenance, and scalability.

- **RESTful API Design:** The backend follows REST principles for clear, stateless communication between client and server.
- **Component-Based UI:** React.js is used to build reusable UI components, enhancing development speed and consistency.
- **Environment Variable Management:** Sensitive information and configuration settings are managed through environment variables, improving security and deployment flexibility.
- **Authentication with JWT:** Secure and scalable authentication is implemented using JSON Web Tokens.
- **Role-Based Access Control (RBAC):** Authorization is managed by assigning roles to users, ensuring that only authorized users can access specific resources or functionalities.
- **Asynchronous Operations:** Efficient handling of API calls and other I/O operations using asynchronous programming patterns.
- **Error Handling:** Robust error handling mechanisms are implemented in the backend to provide meaningful error messages and prevent application crashes.
- **Input Validation:** Server-side input validation is performed to ensure data integrity and security.
- **Payment Gateway Integration:** Secure and reliable payment processing is achieved through direct integration with Stripe.
- **Cloud Storage for Assets:** Utilizing Cloudinary for image storage and delivery ensures scalability and efficient media management.

Future Scope or Improvements

- **Advanced Search & Filtering:** Implement more sophisticated search capabilities with faceted search, autocomplete, and advanced filtering options.
- **Real-time Notifications:** Integrate WebSockets for real-time order updates, chat support, or new product notifications.
- **User Reviews & Q&A:** Enhance the review system with features like review moderation, helpfulness voting, and a Q&A section for products.
- **Product Recommendations:** Implement a recommendation engine based on user behavior, purchase history, or product similarity.

- **Multi-language Support:** Add internationalization (i18n) to support multiple languages for a broader audience.
- **Multi-currency Support:** Allow users to view prices and checkout in different currencies.
- **Admin Dashboard Enhancements:** Develop more advanced analytics, custom reporting, and potentially a CMS for managing static content.
- **Promotions & Discounts:** Implement a module for creating and managing various types of promotions, coupons, and discounts.
- **Inventory Management System:** Integrate a more robust inventory management system with features like low-stock alerts, supplier management, and automated reordering.
- **Shipping & Tax Calculation:** Integrate with third-party APIs for real-time shipping rate calculation and automated tax calculation.
- **Customer Support Integration:** Integrate with a customer support platform or implement a live chat feature.
- **Performance Optimization:** Further optimize frontend and backend performance through code splitting, lazy loading, caching strategies, and database indexing.
- **Unit and Integration Testing:** Implement comprehensive unit and integration tests for both frontend and backend to ensure code quality and stability.
- **Containerization:** Containerize the application using Docker and orchestrate with Kubernetes for easier deployment and scaling.
- **Serverless Deployment:** Explore serverless options for the backend (e.g., AWS Lambda, Google Cloud Functions) to reduce operational overhead.

Detailed Code Component Documentation

This section provides an in-depth look into the key components of each part of the eCommerce project, explaining their purpose, functionality, and how they interact within the larger system.

Admin Frontend Components

The Admin Frontend, built with React.js, provides a robust interface for managing various aspects of the eCommerce platform. Its components are designed for reusability and clear separation of concerns.

`App.js`

`App.js` serves as the main entry point for the Admin Frontend application. It typically sets up the main routing structure using `react-router-dom` and defines the overall layout. It might also include global context providers or authentication checks that apply across the entire application.

Key Responsibilities: - Defines the top-level application structure. - Manages routing to different admin pages (e.g., Dashboard, Product Management, Order Management). - Potentially handles global state or authentication context.

`routes/AdminRoutes.js`

`AdminRoutes.js` defines the protected routes for the admin panel. It typically uses `react-router-dom` to create routes that are only accessible to authenticated and authorized admin users. This file often includes logic to check for an authenticated admin session before rendering the child routes.

Key Responsibilities: - Defines the routing paths for various admin functionalities. - Implements route protection to ensure only authenticated administrators can access specific pages. - Redirects unauthorized users to the admin login page.

`pages/Dashboard.js`

The `Dashboard.js` component is the central hub for administrators, providing an overview of the eCommerce platform's performance. It typically fetches and displays key metrics such as total sales, number of orders, active users, and product statistics. This component often integrates charting libraries (like `react-chartjs-2`) to visualize data.

Key Responsibilities: - Fetches and displays aggregated data for sales, orders, users, and products. - Presents data visually using charts and graphs. - Provides quick links or summaries to other management sections.

`pages/ProductManagement.js`

`ProductManagement.js` is the core component for administrators to manage the product catalog. It provides functionalities to add new products, edit existing product details, and delete products. This component typically interacts with the backend API for CRUD (Create, Read, Update, Delete) operations on product data, including handling image uploads (likely via Cloudinary integration on the backend).

Key Responsibilities: - Displays a list of all products, often with pagination and search/filter capabilities. - Provides forms for adding new products with fields for name, description, price, stock, category, and image uploads. - Enables editing of existing product details. - Handles product deletion. - Manages product images.

Ecommerce Frontend Components

The Ecommerce Frontend, also built with React.js, focuses on providing a user-friendly shopping experience. Its components are designed to be intuitive and efficient for customers.

`App.js`

Similar to the Admin Frontend, `App.js` in the Ecommerce Frontend is the main entry point. It sets up the primary routing for customer-facing pages and often wraps the application with context providers (e.g., `AuthContext`, `CartContext`, `ProductContext`) to manage global state related to user authentication, shopping cart, and product data.

Key Responsibilities: - Defines the overall structure and navigation for the customer-facing application. - Manages routing for pages like Home, Product Listings, Product Details, Cart, Checkout, Login, Signup, Profile, and Orders. - Integrates global context providers to share state across components.

`context/AuthContext.js`, **`context/CartContext.js`**, **`context/ProductContext.js`**

These files define React Contexts, which are used for global state management across the Ecommerce Frontend. They provide a way to share data like user authentication status, shopping cart contents, and product lists without prop-drilling.

- **`AuthContext.js`:** Manages user authentication state (e.g., logged in/out status, user information, JWT token). Provides functions for login, logout, and

registration.

- **`CartContext.js`:** Manages the state of the user's shopping cart. Provides functions to add items, remove items, update quantities, and calculate totals.
- **`ProductContext.js`:** Manages the state of product data, potentially including fetched product lists, categories, and filters. Provides functions to fetch products from the API.

Key Responsibilities: - Centralized state management for specific domains. - Provides data and functions to consuming components. - Reduces prop-drilling and simplifies component trees.

Backend Components

The Backend, built with Node.js and Express.js, serves as the central API for both frontends and handles all data persistence, business logic, and third-party integrations.

`server.js`

`server.js` is the main entry point for the backend application. It initializes the Express.js application, connects to the MongoDB database using Mongoose, configures middleware (e.g., for parsing JSON requests, handling CORS), and mounts the various API routes. It also typically starts the server and listens for incoming requests.

Key Responsibilities: - Application bootstrapping and configuration. - Database connection establishment. - Middleware setup (CORS, body parsing, etc.). - Route registration. - Server startup.

`controllers/productController.js`

Controller files like `productController.js` contain the business logic for handling requests related to a specific resource (in this case, products). These functions interact with the Mongoose models to perform database operations (CRUD), handle request parameters, and send appropriate responses back to the client. They also often include logic for image uploads (e.g., using Cloudinary) and validation.

Key Responsibilities: - Handles incoming HTTP requests for product-related operations. - Interacts with the `Product` Mongoose model to query, create, update, and delete product data. - Manages product image uploads and integration with

Cloudinary. - Implements validation and error handling for product data. - Sends JSON responses back to the client.

`pages/AdminLogin.js` and `pages/AdminSignup.js`

These components handle the authentication processes for administrators. `AdminSignup.js` allows new administrators to register (though in a production environment, admin registration might be restricted or handled internally), while `AdminLogin.js` facilitates existing administrators to log into the system. They interact with the backend's authentication APIs to send user credentials and receive JWT tokens.

Key Responsibilities: - Provide user interfaces for administrator registration and login. - Send authentication requests to the backend API. - Handle successful login by storing the received JWT and redirecting to the dashboard. - Display error messages for failed authentication attempts.

`pages/AdminOrders.js`

`AdminOrders.js` provides administrators with a comprehensive view and management interface for all customer orders. This component fetches order details from the backend, displays them in a structured format (e.g., a table), and allows administrators to update order statuses (e.g., from 'Pending' to 'Shipped' or 'Delivered').

Key Responsibilities: - Fetches and displays a list of all orders. - Provides filtering and sorting options for orders. - Allows administrators to view detailed information for each order. - Enables updating the status of orders.

`pages/Home.js`

`Home.js` is the landing page for the customer-facing eCommerce site. It typically showcases featured products, new arrivals, popular categories, and promotional banners to engage users. It aggregates data from various sources (e.g., product listings, categories) and presents them in an appealing layout.

Key Responsibilities: - Displays a curated selection of products and categories. - Highlights promotions or new collections. - Provides a welcoming and informative entry point for users.

`pages/Products.js`

`Products.js` displays a list of products, often allowing users to browse by category, apply filters (e.g., price range, brand), and sort the results. This component fetches product data from the backend and renders individual product cards or items. It often includes pagination to handle large datasets efficiently.

Key Responsibilities: - Fetches and displays product listings based on user selection or search queries. - Implements filtering, sorting, and pagination functionalities. - Navigates to `ProductDetails.js` when a product is selected.

`pages/ProductDetails.js`

`ProductDetails.js` provides a detailed view of a single product. It displays comprehensive information such as product name, description, price, available stock, multiple images, and customer reviews. Users can typically add the product to their cart or wishlist from this page.

Key Responsibilities: - Fetches and displays detailed information for a specific product. - Manages product image gallery. - Allows users to select quantity and add the product to the shopping cart or wishlist. - Displays existing customer reviews and provides an interface for submitting new ones.

`pages/Cart.js`

`Cart.js` displays the contents of the user's shopping cart. It lists all items added by the user, their quantities, and individual prices, along with the calculated subtotal. Users can update item quantities, remove items, and proceed to checkout from this page. It heavily relies on `CartContext` for state management.

Key Responsibilities: - Displays items currently in the shopping cart. - Allows users to modify item quantities or remove items. - Calculates and displays the total cart value. - Provides a clear path to the checkout process.

`pages/Checkout.js`

`Checkout.js` guides the user through the final steps of placing an order. This typically involves multiple stages, such as entering shipping information, selecting a payment method, and reviewing the order before final confirmation. It integrates with the Stripe API for secure payment processing.

Key Responsibilities: - Collects shipping address and contact information. - Integrates with Stripe for secure credit card or other payment method input. - Summarizes the order details before final submission. - Communicates with the backend to create and process the order.

`pages/Login.js` and `pages/Signup.js`

These components handle user authentication for the customer-facing application. `Signup.js` allows new users to create an account, while `Login.js` enables existing users to sign in. They interact with the backend's authentication APIs and manage the user's session state via `AuthContext`.

Key Responsibilities: - Provide user interfaces for account registration and login. - Send authentication requests to the backend API. - Handle successful authentication by updating `AuthContext` and redirecting the user. - Display appropriate error messages for failed attempts.

`pages/Profile.js`

`Profile.js` allows authenticated users to view and manage their personal information, including their name, email, shipping addresses, and potentially password changes. It fetches user data from the backend and provides forms for updating this information.

Key Responsibilities: - Displays the user's personal details. - Enables users to update their profile information and addresses. - Provides access to other user-specific functionalities like order history.

`pages/Orders.js`

`Orders.js` displays a list of all past orders placed by the authenticated user. It provides a summary of each order (e.g., order ID, date, total amount, status) and allows users to navigate to a detailed view of a specific order.

Key Responsibilities: - Fetches and displays a history of the user's orders. - Provides a summary for each order. - Links to `OrderDetails.js` for more in-depth information.

`pages/OrderDetails.js`

`OrderDetails.js` provides a comprehensive view of a single order. It displays all items included in the order, their quantities and prices, shipping information, payment details, and the current status of the order. This component fetches specific order data from the backend.

Key Responsibilities: - Displays detailed information for a selected order. - Shows ordered products, quantities, and prices. - Presents shipping and payment information. - Indicates the current status of the order.

`pages/Wishlist..js`

`Wishlist..js` (likely `Wishlist.js`) allows users to manage a list of products they are interested in but are not ready to purchase immediately. Users can add products to their wishlist from product detail pages and later move them to the cart or remove them from the wishlist. This component interacts with the backend's wishlist APIs and potentially `WishlistContext` if implemented.

Key Responsibilities: - Displays products saved in the user's wishlist. - Allows users to remove items from the wishlist. - Provides an option to move items from the wishlist to the shopping cart.

Backend Models

The backend models define the schema for the data stored in the MongoDB database. Each `.js` file in the `models` directory typically corresponds to a Mongoose schema and model, providing a structured way to interact with the database.

`models/User.js`

The `User` model represents a user of the eCommerce system. This model is crucial for authentication and authorization, storing user credentials, roles, and personal information. It typically includes fields for username, email, password (hashed), and a role to differentiate between regular customers and administrators.

Schema Fields (Example): - `username`: String, unique, required. - `email`: String, unique, required, indexed. - `password`: String, required (stored as a hashed value). - `role`: String, enum (`user`, `admin`), default `user`. - `address`: Object (or sub-

document) containing shipping and billing addresses. - `phone`: String. - `createdAt`: Date, default `Date.now`. - `updatedAt`: Date.

Key Responsibilities: - Defines the structure and validation rules for user data. - Handles password hashing before saving to the database (pre-save hook). - Provides methods for user authentication (e.g., comparing passwords).

`models/Product.js`

The `Product` model defines the structure for all products available in the store. It includes details necessary for displaying products on the frontend, managing inventory, and associating products with orders and reviews.

Schema Fields (Example): - `name`: String, required, unique. - `description`: String, required. - `price`: Number, required, min 0. - `category`: String, required (or reference to a Category model). - `stock`: Number, required, min 0. - `images`: Array of Strings (URLs to product images, likely stored on Cloudinary). - `reviews`: Array of Object IDs referencing `Review` documents. - `createdAt`: Date, default `Date.now`. - `updatedAt`: Date.

Key Responsibilities: - Stores comprehensive product information. - Manages inventory levels. - Facilitates linking products to categories and customer reviews.

`models/Order.js`

The `Order` model captures all details related to a customer's purchase. It links to the `User` who placed the order and includes an array of products purchased, along with their quantities and the total amount. This model is central to tracking sales and managing order fulfillment.

Schema Fields (Example): - `user`: ObjectId, required, references `User`. - `products`: Array of Objects, each containing: - `product`: ObjectId, required, references `Product`. - `quantity`: Number, required, min 1. - `totalAmount`: Number, required. - `status`: String, enum (`pending`, `processing`, `shipped`, `delivered`, `cancelled`), default `pending`. - `shippingAddress`: Object (or sub-document) with address details. - `paymentDetails`: Object (or sub-document) with payment transaction details. - `createdAt`: Date, default `Date.now`. - `updatedAt`: Date.

Key Responsibilities: - Records complete transaction details for each customer order. - Tracks the status of orders through their lifecycle. - Links orders to specific users and products.

`models/Cart.js`

The `Cart` model represents a user's shopping cart. It stores the items that a user intends to purchase before completing the checkout process. This model is typically associated with a `User` and contains an array of products with their respective quantities.

Schema Fields (Example): - `user`: ObjectId, required, unique, references `User`. - `items`: Array of Objects, each containing: - `product`: ObjectId, required, references `Product`. - `quantity`: Number, required, min 1. - `createdAt`: Date, default `Date.now`. - `updatedAt`: Date.

Key Responsibilities: - Persists the contents of a user's shopping cart. - Allows for adding, removing, and updating quantities of items. - Facilitates the transition of items from cart to order during checkout.

`models/Wishlist.js`

The `Wishlist` model allows users to save products they are interested in for future consideration. It provides a convenient way for users to keep track of desired items without immediately adding them to the cart.

Schema Fields (Example): - `user`: ObjectId, required, unique, references `User`. - `products`: Array of Object IDs referencing `Product` documents. - `createdAt`: Date, default `Date.now`. - `updatedAt`: Date.

Key Responsibilities: - Stores a list of products a user wishes to save. - Enables users to add or remove products from their wishlist.

`models/Review.js`

The `Review` model stores customer feedback and ratings for products. This model is essential for building trust and providing valuable insights to other potential buyers. It links a review to a specific user and product.

Schema Fields (Example): - `user`: ObjectId, required, references `User`. - `product`: ObjectId, required, references `Product`. - `rating`: Number, required, min 1, max 5. - `comment`: String. - `createdAt`: Date, default `Date.now`. - `updatedAt`: Date.

Key Responsibilities: - Stores product ratings and textual reviews. - Associates reviews with the user who submitted them and the product being reviewed.