VIETNAM NATIONAL UNIVERSITY HCMC

INTERNATIONAL UNIVERSITY

WEB APPLICATION DEVELOPMENT PROJECT

ONLINE SHOES STORE

By

Pham Minh Thuc - ITITWE23033

Nguyen Huynh Dang Khoa - ITITWE22152

I. INTRODUCTION

1. Project Overview

This project is a Shoe Store Web Application developed to provide a complete online shopping experience for customers and an efficient management system for administrators. The application allows users to create accounts, browse shoes, manage shopping carts, place orders, and view order history. Administrators can manage user accounts, products, and orders, including updating order statuses.

The system is designed as a full-stack web application with clear separation between front-end, back-end, and database layers. It focuses on usability, scalability, and maintainability while applying modern web development practices.

2. Development

The application is built using the MERN stack:

- **MongoDB**: NoSQL database for storing users, products, carts, and orders
- **Express.js**: Back-end framework for building RESTful APIs
- **React.js**: Front-end library for building user interfaces
- **Node.js**: Server-side runtime environment

Additional technologies:

- **Tailwind CSS**: Used for responsive and modern UI styling
- **Cloudinary**: Used to store and manage product images
- **JWT Authentication**: Used for secure user and admin authentication

Overall System Workflow**:**

User / Admin

↓

Browser

↓

Server

↓

Database

↓

Response

II. REQUIREMENT ANALYSIS AND DESIGN

1. **REQUIREMENT ANALYSIS**

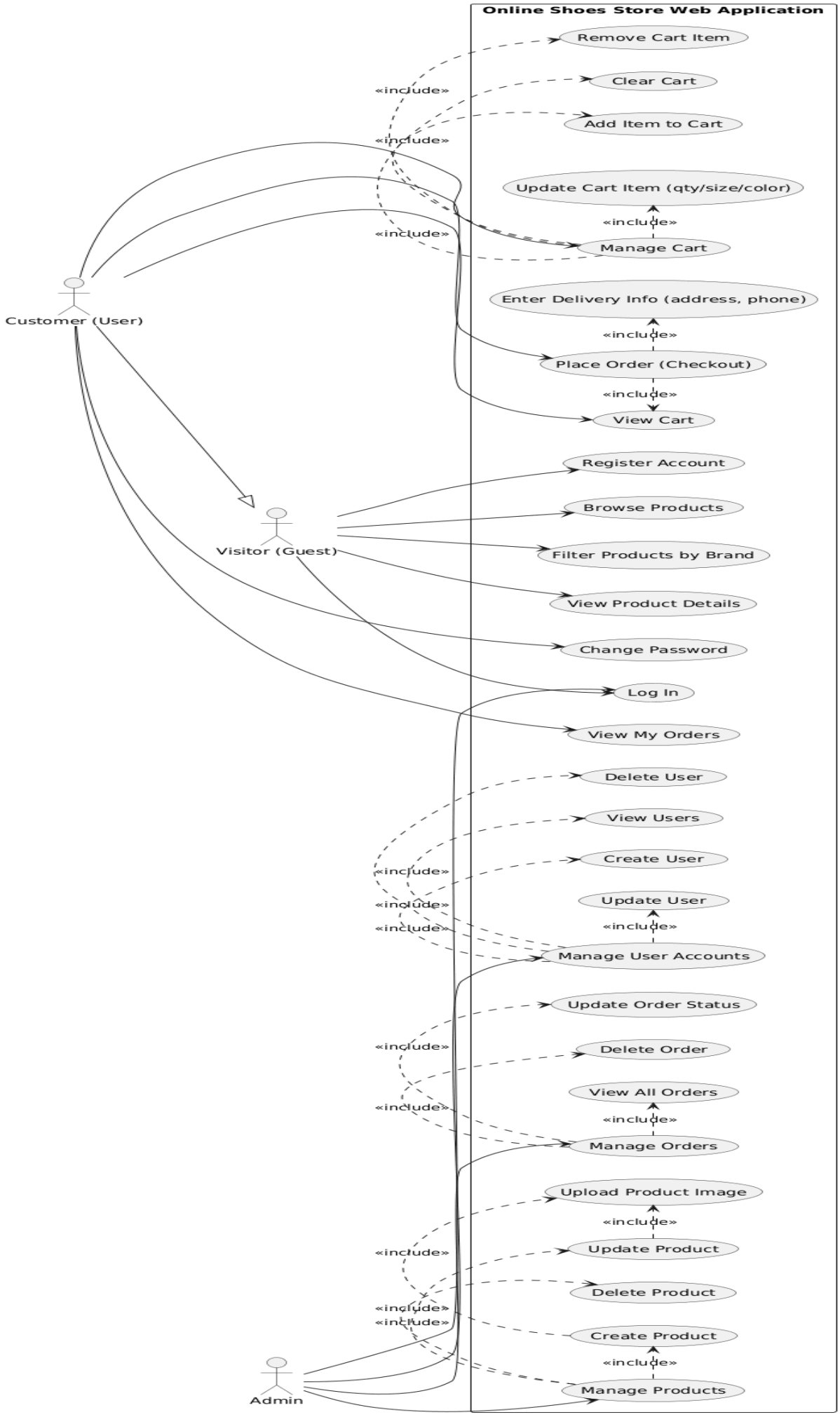
Use Case Diagram

This Use Case Diagram summarizes the main functions of the **Online Shoes Store Web Application** and shows how different users interact with the system.

There are **three actors**:

- **Visitor (Guest):** can **browse products**, **filter products by brand**, **view product details**, **register an account**, and **log in**.
- **Customer (User):** a logged-in user who can do all guest actions and additionally **manage the shopping cart**, **place an order (checkout)**, **view their own orders**, and **change password**.
  The cart functions are modeled using `<<include>>`: **Manage Cart** includes **Add Item**, **Update Item (qty/size/color)**, **Remove Item**, and **Clear Cart**.
  The checkout process is also decomposed using `<<include>>`: **Place Order (Checkout)** includes **View Cart** and **Enter Delivery Info (address, phone)**.
- **Admin:** manages the system data and operations. Admin can **manage products**, **manage user accounts**, and **manage orders**.
  Each admin management use case is broken down with `<<include>>` into smaller actions:
  - **Manage Products** includes **Create Product**, **Update Product**, **Delete Product**, and **Upload Product Image**.
  - **Manage User Accounts** includes **View Users**, **Create User**, **Update User**, and **Delete User**.
  - **Manage Orders** includes **View All Orders**, **Update Order Status**, and **Delete Order**.

Overall, the diagram illustrates the system boundary, the responsibilities of each actor, and how complex features are decomposed into reusable sub-functions using the `<<include>>` relationship.


Online Shoes Store Web Application
Remove Cart Item
Clear Cart
Add Item to Cart
Update Cart Item (qty/size/color)
Manage Cart
Enter Delivery Info (address, phone)
Place Order (Checkout)
View Cart
Register Account
Browse Products
Filter Products by Brand
View Product Details
Change Password
Log In
View My Orders
Delete User
View Users
Create User
Update User
Manage User Accounts
Update Order Status
Delete Order
View All Orders
Manage Orders
Upload Product Image
Update Product
Delete Product
Create Product
Manage Products
«include»
Customer (User)
Visitor (Guest)
Admin

2. DESIGN

ERD

This ERD (Entity–Relationship Diagram) describes the core database structure of the **Online Shoes Store Web Application** and how data is connected across users, products, carts, and orders.

Main entities

- **User** stores account information: `user_id`, `name`, `username`, `password`, and `role` (e.g., customer/admin).
- **Product** stores product catalog data: `product_id`, `name`, `brand`, available `size`/`color`, `price`, `description`, `image`, `stock`, and timestamps (`createdAt`, `updatedAt`).
- **Cart** represents a user's shopping cart: `cart_id`, `user_id`, and timestamps. Each cart belongs to one user.

Weak/Detail entities (line items)

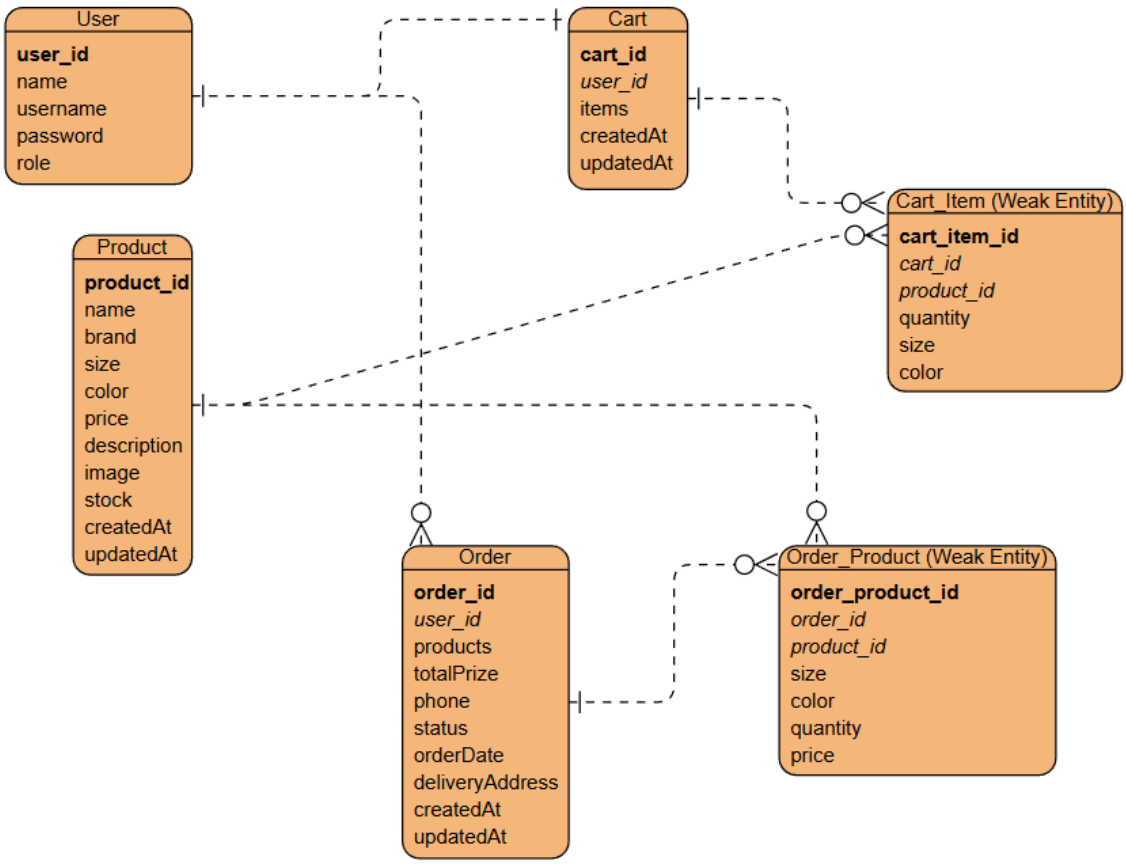
- **Cart_Item (Weak Entity)** represents each item inside a cart. It links a cart to a product and stores purchase options:
  - `cart_item_id`, `cart_id`, `product_id`, `quantity`, `size`, `color`
    This design allows a cart to contain multiple products, and the same product can appear in many carts.
- **Order** stores checkout information after a user places an order:
  - `order_id`, `user_id`, `totalPrize` (total price), `phone`, `status`, `orderDate`, `deliveryAddress`, and timestamps.
    A user can create multiple orders over time.
- **Order_Product (Weak Entity)** stores the detailed items of each order (order line items):
  - `order_product_id`, `order_id`, `product_id`, `size`, `color`, `quantity`, `price`
    The `price` is stored here to preserve the product price at the time of purchase (even if the product price changes later).

Relationships summary

- **User → Cart:** one user has a cart (cart is linked by `user_id`).
- **Cart → Cart_Item:** one cart contains many cart items (each cart item belongs to one cart).
- **Cart_Item → Product:** each cart item refers to one product, while a product can appear in many cart items.
- **User → Order:** one user can place many orders (each order belongs to one user).

- **Order → Order_Product:** one order contains many order products (each line item belongs to one order).
- **Order_Product → Product:** each order line item refers to one product, and a product can appear in many orders.

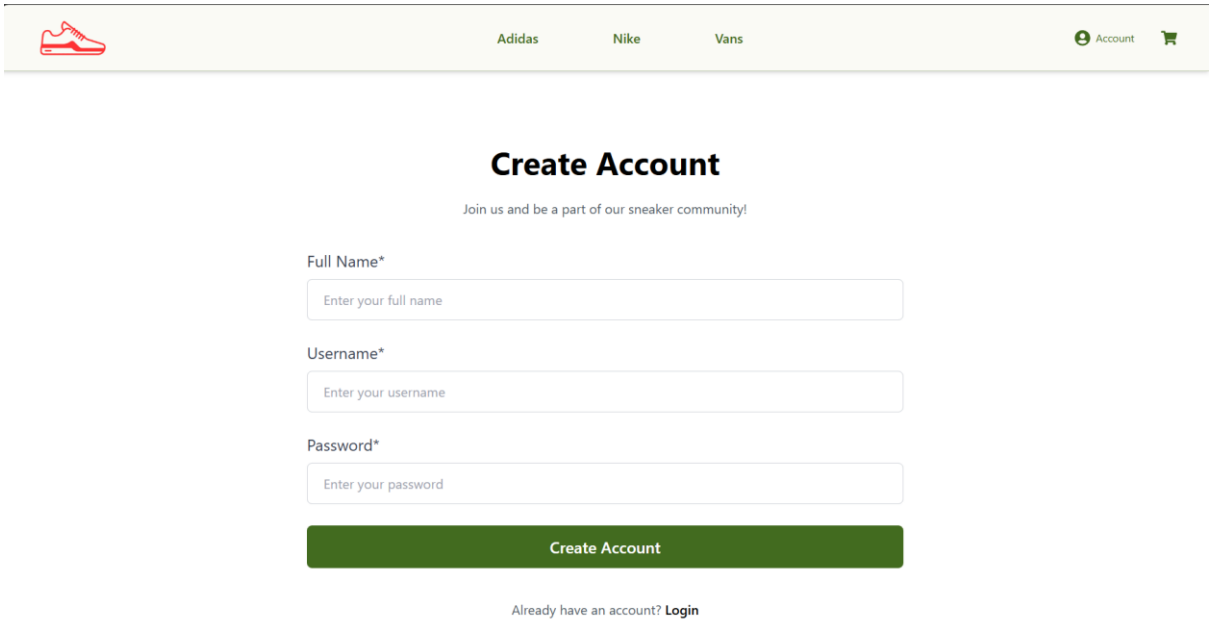
Overall, this ERD separates “master data” (User, Product) from “transaction data” (Cart/Order) and uses weak entities (Cart_Item, Order_Product) to model **many-item lists** inside carts and orders in a clean, normalized way.



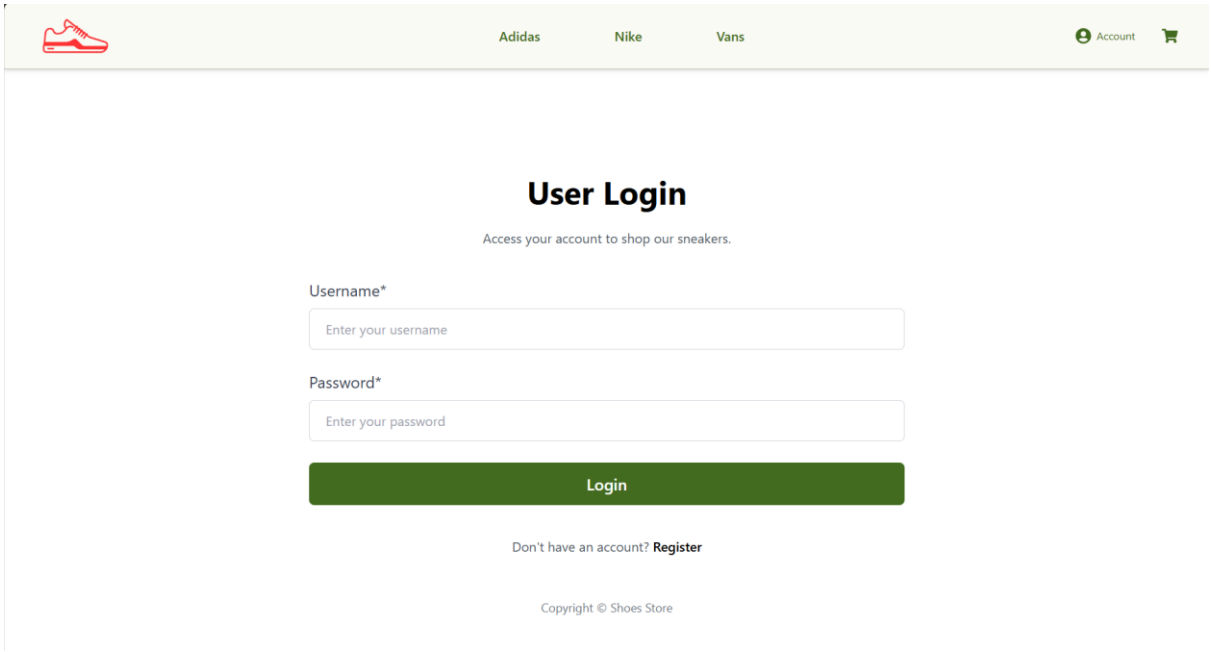
III. IMPLEMENTATION

1 User Functions

a. Register and Login

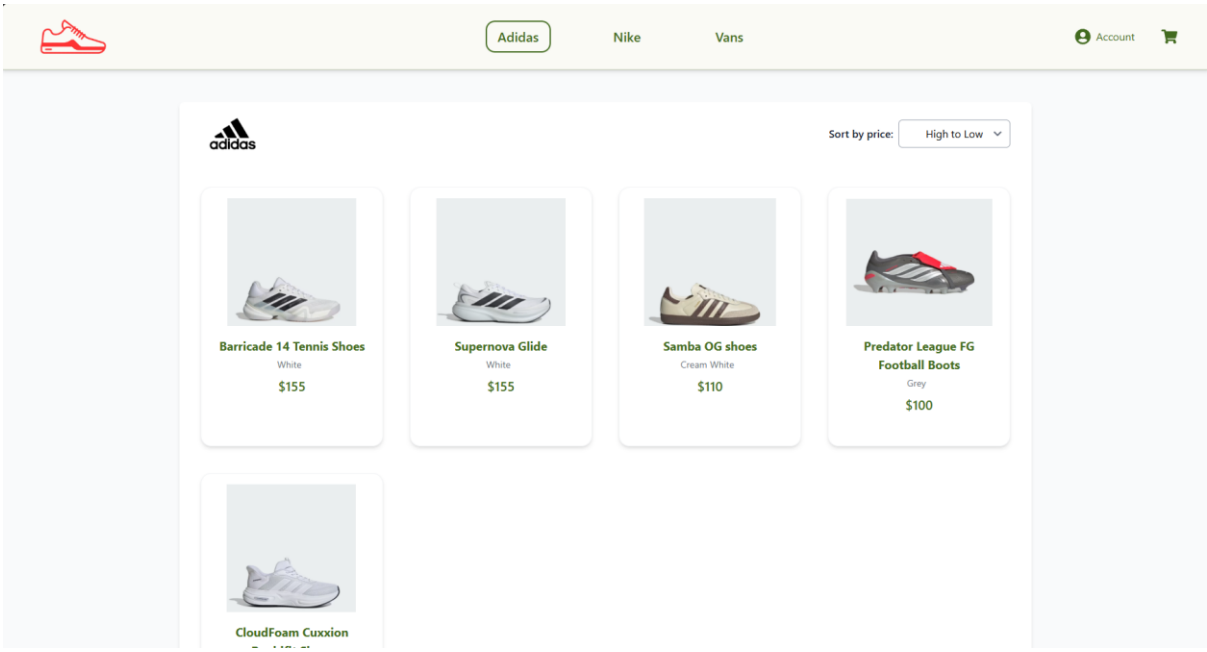


Clicking on the register button	/register -> pickup by Route.jsx -> Register page.
Filling in the form and press Create Account	authApi.jsx /auth/register -> picked up by server.js -> authRoute -> authController checking validation and check for duplicate username -> user model save a new user in the database -> response message and -> /login pickup by Route.jsx and redirect to login page.



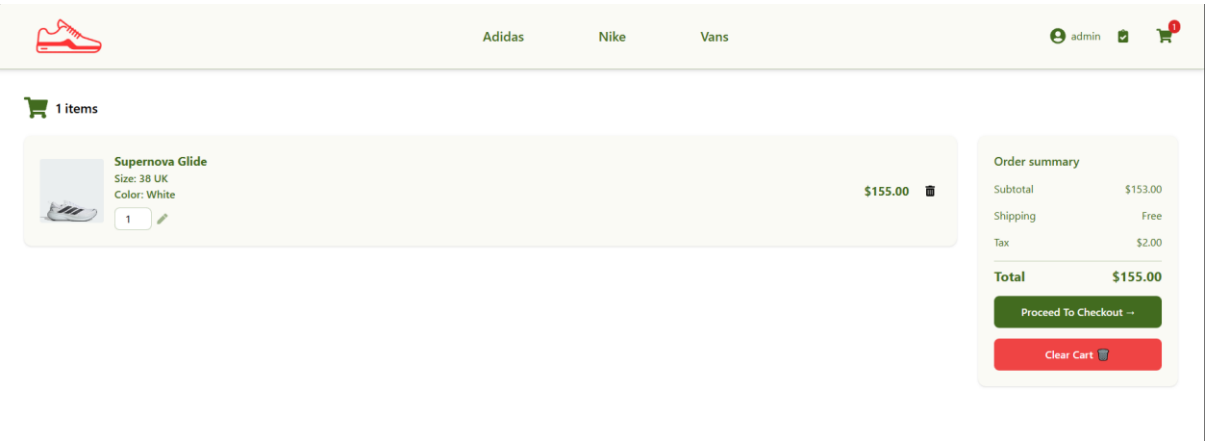
Filling in the form and click Login	authApi.jsx /auth/login -> server pickup -> authRoute -> authController.loginUser -> check validation -> create token which expires in 1 hour -> log in and response message.
Log out	Log out erase the token and log out.

b. Browsing Shoes



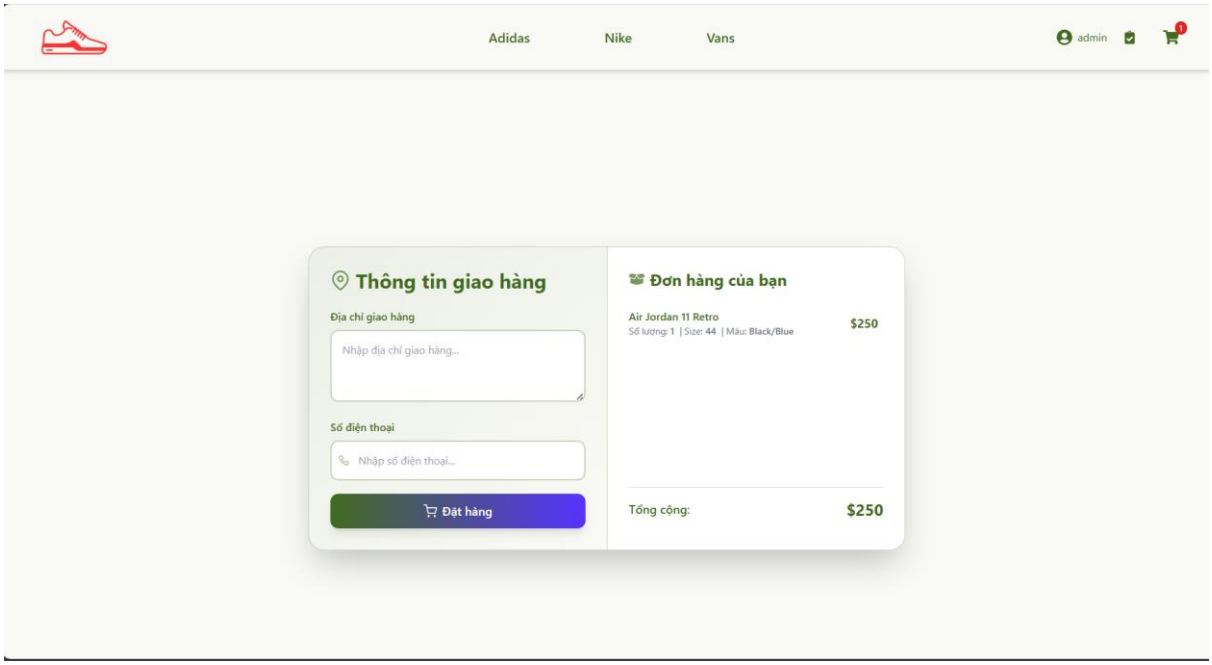
Click on the brand	/brand -> productApi.jsx GET /products/brand -> pick up by server -> productRoute -> middleware -> productController.getProductsByBrand -> fetching products -> show the product in BrandProduct.jsx.
Click on a product	/product/:id -> productApi.jsx GET /products/id -> pick up by server -> productRoute -> middleware -> productController.getProductById -> fetching product details -> show in ProductDetail.jsx

c. Cart Management

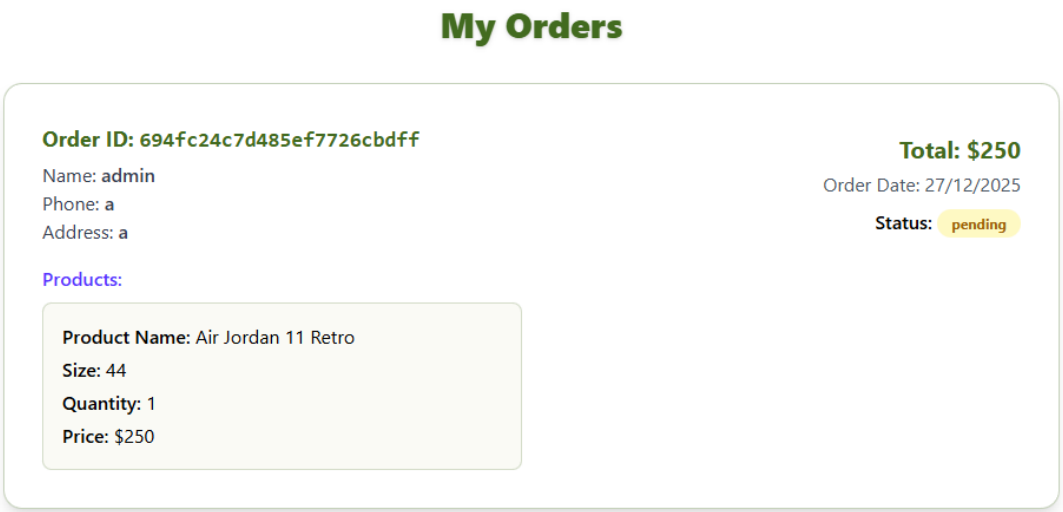


Click on Cart icon:	/cart -> cartApi.jsx GET /cart/:userId -> picked up by server -> cartRoute -> middleware -> cartController.getCartByUser -> showing items in Cart.jsx
Click "Add to Cart" on ProductDetail page:	cartApi.jsx PUT /cart/:userId -> picked up by server -> cartRoute -> middleware -> cartController.updateCart -> update cart
Change quantity input in Cart page:	Cart.jsx.onQuantityChange -> cartApi.jsx sends PUT /api/cart/:userId -> middleware -> cartController.updateCart -> return updated cart
Click delete icon on cart item:	Cart.jsx.onRemove -> cartApi.jsx sends DELETE/api/cart/:userId/item/:productId?size=&color= -> picked up by server -> cartRoute -> middleware -> cartController.deleteCartItem -> return updated cart
Click "Clear Cart" button:	Cart.jsx.onClearCart -> cartApi.jsx sends DELETE /api/cart/:userId -> picked up by server -> cartRoute -> middleware -> cartController.deleteCart -> return empty cart

d. Placing and Viewing Orders



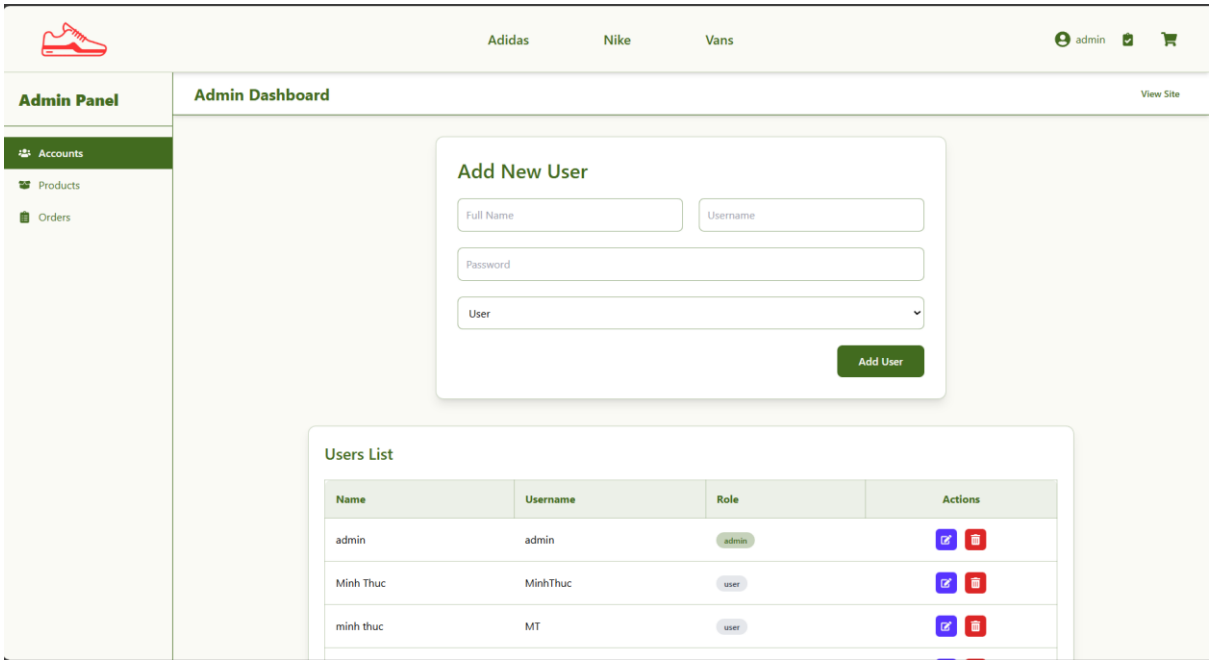
Click "Proceed To Checkout":	Cart.jsx /order -> OrderForm.jsx
Submit order form:	orderApi.jsx sends POST /api/orders -> picked up by server -> orderRoute -> middleware -> orderController.createOrder -> Order model save to database -> redirect user to /my-orders



redirect user to /my-orders or clicking on my orders icon	MyOrders.jsx calls useGetMyOrdersQuery() -> orderApi.jsx sends GET /api/orders/user/:userId -> picked up by server -> orderRoutes -> middleware -> orderController.getOrdersByUser -> show orders.

2 Admin Functions

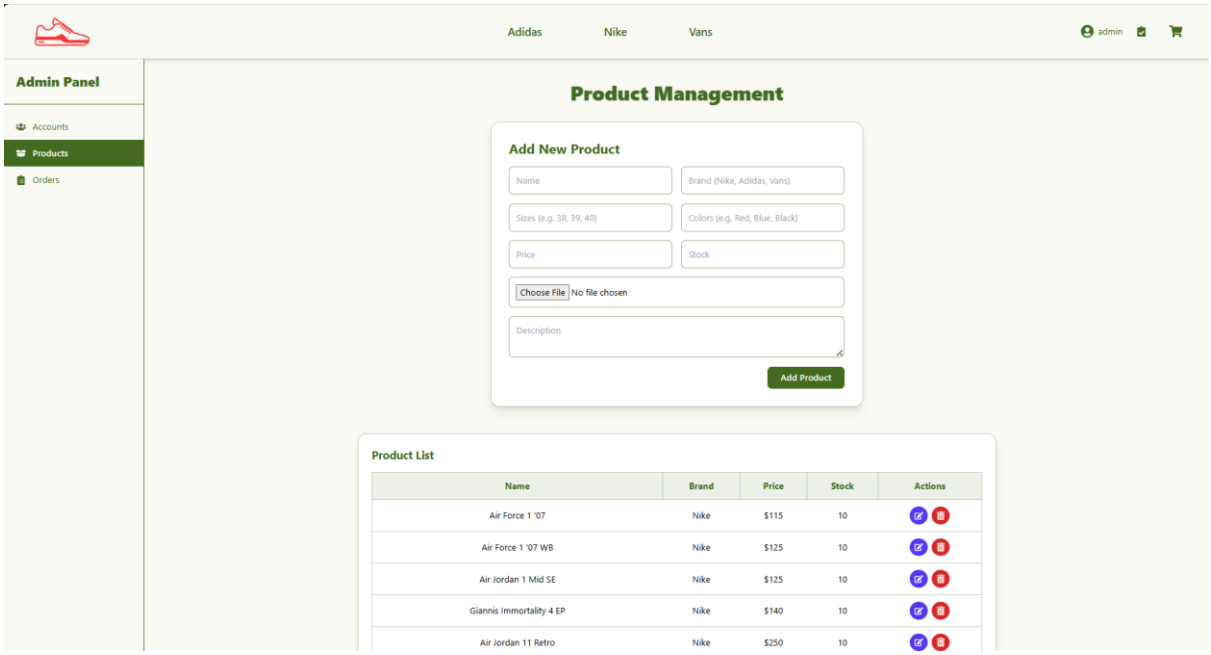
a. Account Management



Admin Accounts page	PrivateRouteRole checks admin role -> AdminAccounts.jsx -> userApi.jsx sends GET /api/auth/users -> picked up by server -> authRoutes -> authenticateUser + admin authorization -> authController.getAllUsers -> return user list
Edit account	After update information -> -> userApi.jsx sends PUT /api/auth/users/:id -> picked up by server -> authRoutes -> authenticateUser + admin authorization ->

	authController.updateUser-> User.findbyIdandUpdate -> return updated user
Delete account	userApi.jsx sends DELETE /api/auth/users/:id -> picked up by server -> authRoutes -> authenticateUser + admin authorization -> authController.deleteUser -> User.findByIdAndDelete -> return

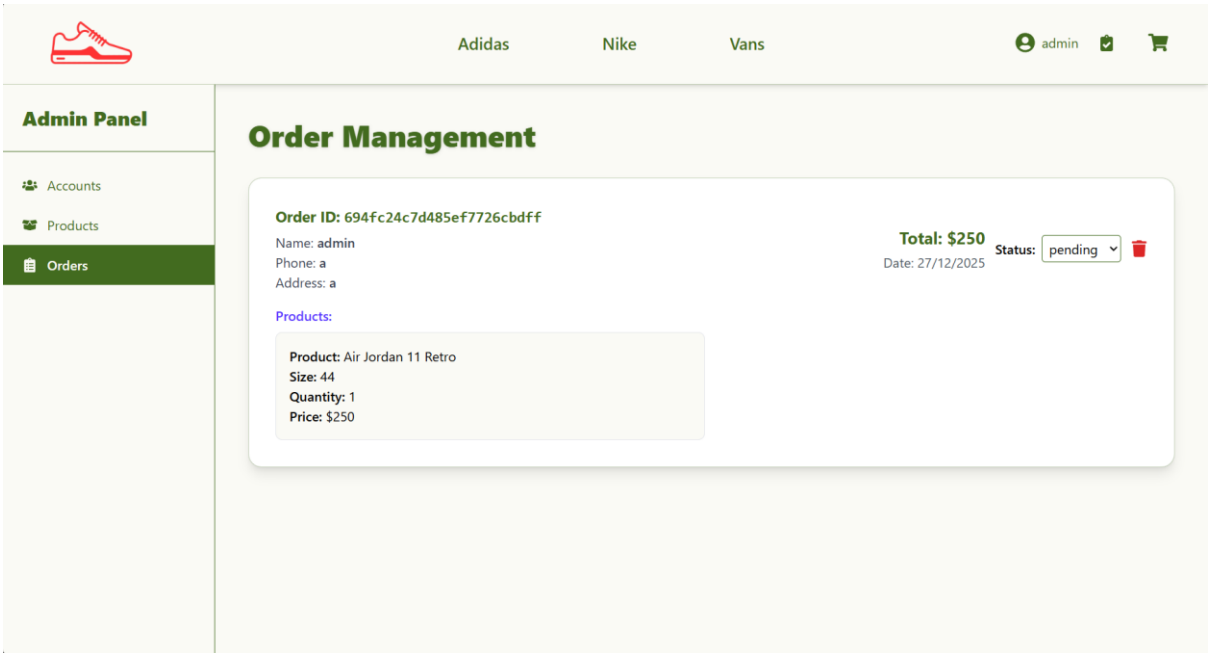
b. Product Management



Admin Products page	PrivateRouteRole checks admin role -> ProductManager.jsx -> productApi.jsx sends GET /api/products -> picked up by server -> productRoutes -> productController.getAllProducts -> return product list
Add new product	Fill in the form -> productApi.jsx sends POST /api/products -> picked up by server -> productRoutes -> authenticateUser + admin authorization ->

	productController.createProduct -> product model save to database -> return created product
Edit product	Select the product to edit -> edit -> productApi.jsx sends PUT /api/products/:id -> picked up by server -> productRoutes -> authenticateUser + admin authorization -> productController.updateProduct -> return updated product
Delete product	productApi.jsx sends DELETE /api/products/:id -> picked up by server -> productRoute -> authenticateUser + admin authorization -> productController.deleteProduct -> return success response

c. Order Management



Admin Orders page	PrivateRouteRole checks admin role -> AllOrders.jsx -> orderApi.jsx sends GET /api/orders -> picked up by server ->

	orderRoutes -> authenticateUser + admin authorization -> orderController.getAllOrders -> return order list
Update order status	Select order -> change status -> orderApi.jsx sends PUT /api/orders/:id -> picked up by server -> orderRoutes -> authenticateUser + admin authorization -> orderController.updateOrderStatus -> return updated order
Delete order	orderApi.jsx sends DELETE /api/orders/:id -> picked up by server -> orderRoutes -> authenticateUser + admin authorization -> orderController.deleteOrder -> return success response

IV. DISCUSSION AND CONCLUSION

1 Discussion

During development, challenges included designing a NoSQL database structure, handling authentication and authorization, and integrating image storage with Cloudinary. These challenges were addressed through proper schema design, middleware usage, and third-party service integration.

2 Conclusion

In conclusion, the Shoe Store Web Application successfully fulfills its objectives by providing a functional and user-friendly e-commerce platform. The system demonstrates effective use of the MERN stack, secure authentication, and scalable database design. This project reflects a solid understanding of full-stack web application development and provides a strong foundation for future enhancements such as online payment integration and advanced analytics.