Task

Develop a full-stack shopping cart using **React + Redux Toolkit** for frontend state management and **NestJS (Node.js + TypeScript + PostgreSQL)** for backend API.

Brief

The shopping cart should allow fetching products from a PostgreSQL database through a backend API, adding/removing items, and dynamically displaying the total number of items and total price. The implementation should follow modern best practices for **Redux Toolkit** and **NestJS with PostgreSQL** to ensure scalability and maintainability.

Requirements

1. **Backend (NestJS + Node.js + TypeScript)**

    1. **Database (PostgreSQL)**
        - Use PostgreSQL for storing products and cart_items.
        - Each product should include: id, name, price, image.
        - Each cart item should reference a product and include: id, productId, quantity.

    2. **API Endpoints**
        - GET /products → Returns list of products from PostgreSQL. (id, name, price, image).
        - GET /cart → Returns cart items with total count and price.
        - POST /cart → Add an item to the cart (insert/update in PostgreSQL).
        - DELETE /cart/:id → Remove an item from the cart in PostgreSQL.

    3. **TypeScript & Validation**
        - Use DTOs for request/response validation.
        - Define types/interfaces for Product and Cart models.
        - Strong typing for API responses.

    4. **Database Integration**
        - Use **TypeORM** or **Prisma** with NestJS.
        - Provide migration/seeding script to insert sample products into PostgreSQL.

    5. **Error Handling**
        - Handle invalid requests with proper HTTP status codes.

2. **Frontend (React + Redux Toolkit + TypeScript + Vite)**

    1. **Product Listing**

- Fetch products from the backend API and display them (name, price, image).

2. **Shopping Cart Functionality**
   - Add items to cart (via API).
   - Remove items from cart (via API).
   - Display cart summary with total items and total price (via API).

3. **State Management**
   - Use Redux Toolkit slices for products and cart.
   - Implement async thunks or RTK Query for API integration.

4. **User Experience**
   - Show loading and error states for API calls.
   - Use reusable components (e.g., ProductCard, CartItem, CartSummary).

5. **Persistence (Optional)**
   - Use Redux Persist or localStorage so cart data survives refresh.

Expectations

- **Component-Based Architecture**: Design should use reusable components to ensure maintainability.
- **Scalable Redux Setup**: Follow Redux Toolkit best practices.
- **Database First**: PostgreSQL schema with seed data.
- **TypeScript Everywhere**: Proper typings across frontend and backend.
- **Error Handling**: Show clear message when requests fail.
- **Efficiency**: Minimize unnecessary re-renders and optimize state updates.
- **User Experience**: Ensure smooth and intuitive interaction for adding/removing items.
- **Clean Code**: Maintain well-structured and readable code following best practices.

Tools & Libraries

- **Frontend**: React, Redux Toolkit, TypeScript, Vite
- **Backend**: NestJS, Node.js, TypeScript
- **Database**: PostgreSQL, TypeORM or Prisma
- **Optional**: Jest, React Testing Library, Redux Persist

Timeline

Complete the implementation within **6–8 hours** of receiving this task. Submit the project with a GitHub repository link or share a ZIP file of the codebase. Include a short README with setup instructions and API details.

Evaluation Criteria

1. **Redux or Redux Toolkit Implementation**: Efficient and correct usage of Redux or Redux Toolkit for managing state.
2. **NestJS Implementation**: Proper modular structure and DTO(Data Transfer Object) usage.
3. **PostgreSQL Integration**: Well-defined schema, migrations, and queries.
4. **Functionality**: The cart should work as expected with add/remove functionality with API integration.
5. **TypeScript Usage**: Strong typing across codebase.
6. **Error Handling**: Proper validation and fallback states.
7. **Code Quality**: Clean, well-structured, and maintainable code.
8. **User Interface**: A simple yet intuitive UI that enhances user experience.
9. **Performance Optimization**: Efficient state updates and minimal re-renders.