Frontend Architecture and System Design for eCommerce Solution

1. Overview

This eCommerce platform is developed using React (a JavaScript library for building user interfaces), TypeScript (for static typing), and Vite (for fast bundling and development with Hot Module Replacement). This architecture is designed for modularity, scalability, and performance.

2. Frontend Architecture

a. Component-Based Design (with a combination of modular design)

The application follows a component-based architecture, where the UI is divided into reusable, isolated components. Each component represents a specific part of the interface (e.g., product listing, cart, checkout, etc.).

- React Functional Components: Leverages React's functional components with hooks for state management and lifecycle events.
- TypeScript: Adds type safety, reducing runtime errors by catching issues during development.

b. Folder Structure

The folder structure is organized to separate concerns and ensure scalability as the app grows:

- src/components: Contains reusable UI components.
- src/pages: Contains specific page-level components, like Home, Product, Checkout, etc.
- src/services: Contains API calls and business logic.
- src/hooks: Reusable logic and custom hooks.
- src/utils: Utilities function to support common and reusable business logic.
- src/store: Necessary configuration to handle app-wide state using Redux and Redux Thunk
- src/tests: Unit test cases for TDD using vitest
- src/i18n: Functionality to handle internationalization (will be applied in real app)
- src/assets: Images, icons, and styling files.

c. Styling

CSS Modules or Styled Components: The project uses a modular styling system to scope styles to components, preventing global style conflicts. Currently, it uses pure CSS, but in real applications, we will use CSS preprocessors like SASS or SCSS and combine utilities for styled-component.

d. Routing

React Router: Implements client-side routing for seamless navigation between pages (e.g., product page, checkout page, etc.), without full-page reloads.

3. State Management

a. Local State

React Hooks: useState and useEffect was used to manage the local component state. For example, product data and user input can be controlled using the local state.

b. Global State Management

For this application, we are using Redux to manage the app-wide state. For smaller components, we could use Context API to handle the states but for our application it was not needed. A few reasons for using Redux with our application are:

Global State Management: In a complex app, we often need to share state across different components and areas of the app (e.g., user data, authentication, shopping cart, etc.). Manually passing this data through props (prop-drilling) becomes tedious and error-prone, especially when passing data through deeply nested components.

Centralized State: Redux provides a single, centralized store for all the app's states. This means all components can access the global state without needing to pass it down through multiple layers, reducing code complexity and improving maintainability.

Predictability: Redux enforces strict unidirectional data flow, meaning all state transitions follow a clear path: from action dispatch to reducer, to the updated store. This makes it easier to debug and predict how the state is updated. In a large app with complex UI interactions, having a predictable state management system like Redux is essential for avoiding unpredictable behavior.

Separation of Concerns: Redux helps separate state management from UI components. While React components focus on rendering the UI, Redux handles business logic and state transitions, leading to cleaner, more modular code. This separation is crucial for large-scale apps where maintainability is key.

DevTools and Debugging: Redux comes with powerful DevTools that allow us to time-travel through state changes, inspect actions, and visualize state transitions. This makes debugging large apps significantly easier compared to managing the state with local component state or context.

Middleware Support: Complex apps often involve asynchronous operations (e.g., API calls). Redux provides middleware (Redux Thunk was used for our case) that allows handling side effects in a clear and scalable manner. This makes handling complex state changes and async flow much more manageable.

4. Data Flow and API Integration

a. REST API

The eCommerce solution integrates with backend services using API calls. The service layer (in src/services and src/module/[module_name]/services) handles:
- Product Fetching: Fetch product details from a backend database.
- Cart & Checkout: Send and receive data for the user's shopping cart and checkout process.

b. Axios

Axios was used for asynchronous data fetching and handling HTTP requests.

5. Performance Optimization

a. Vite

Fast Development: The use of Vite ensures fast development builds and Hot Module Replacement (HMR) to allow real-time updates to the app without page reloads.

b. Code Splitting

Lazy Loading: React's React.lazy and Suspense can be used to load components on demand (e.g., only loading the checkout page when the user navigates to it), improving performance.

c. Caching

API response caching (e.g., through Service Workers) to reduce redundant network requests and improve loading times.

6. Testing

Vitest: The app would benefit from unit tests for components and integration tests for API interactions to ensure reliability. For our case we are using Vitest to write out unit tests to handle TDD.

7. Deployment

The project can be deployed on services like Vercel or Netlify, which are compatible with Vite and offer continuous deployment pipelines from GitHub repositories.

Conclusion

This system design provides a scalable, maintainable, and efficient frontend architecture for an eCommerce solution using modern web technologies like React, TypeScript, and Vite. As the app grows, state management, performance optimization, and testing strategies can be adapted to maintain high-quality user experience.