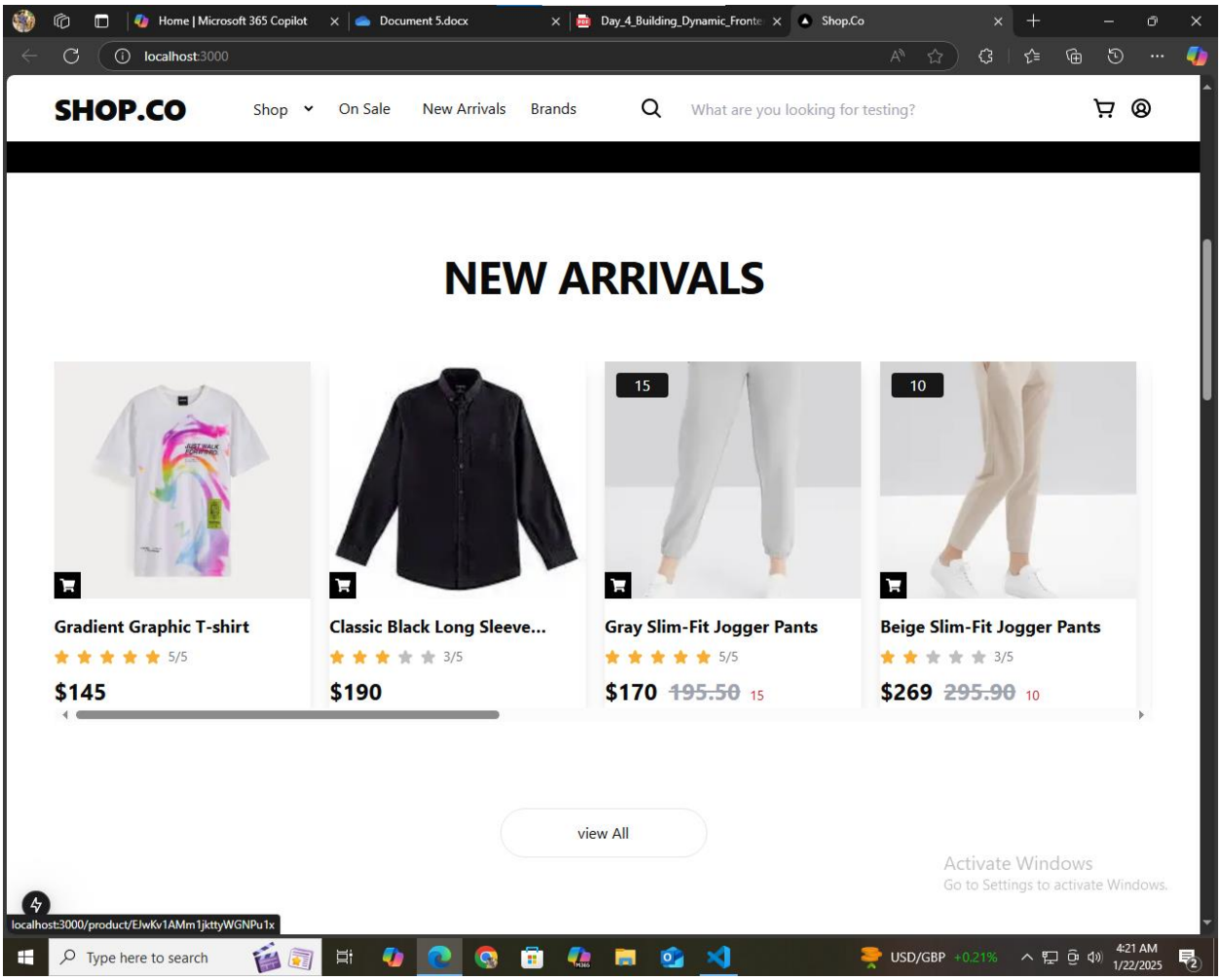
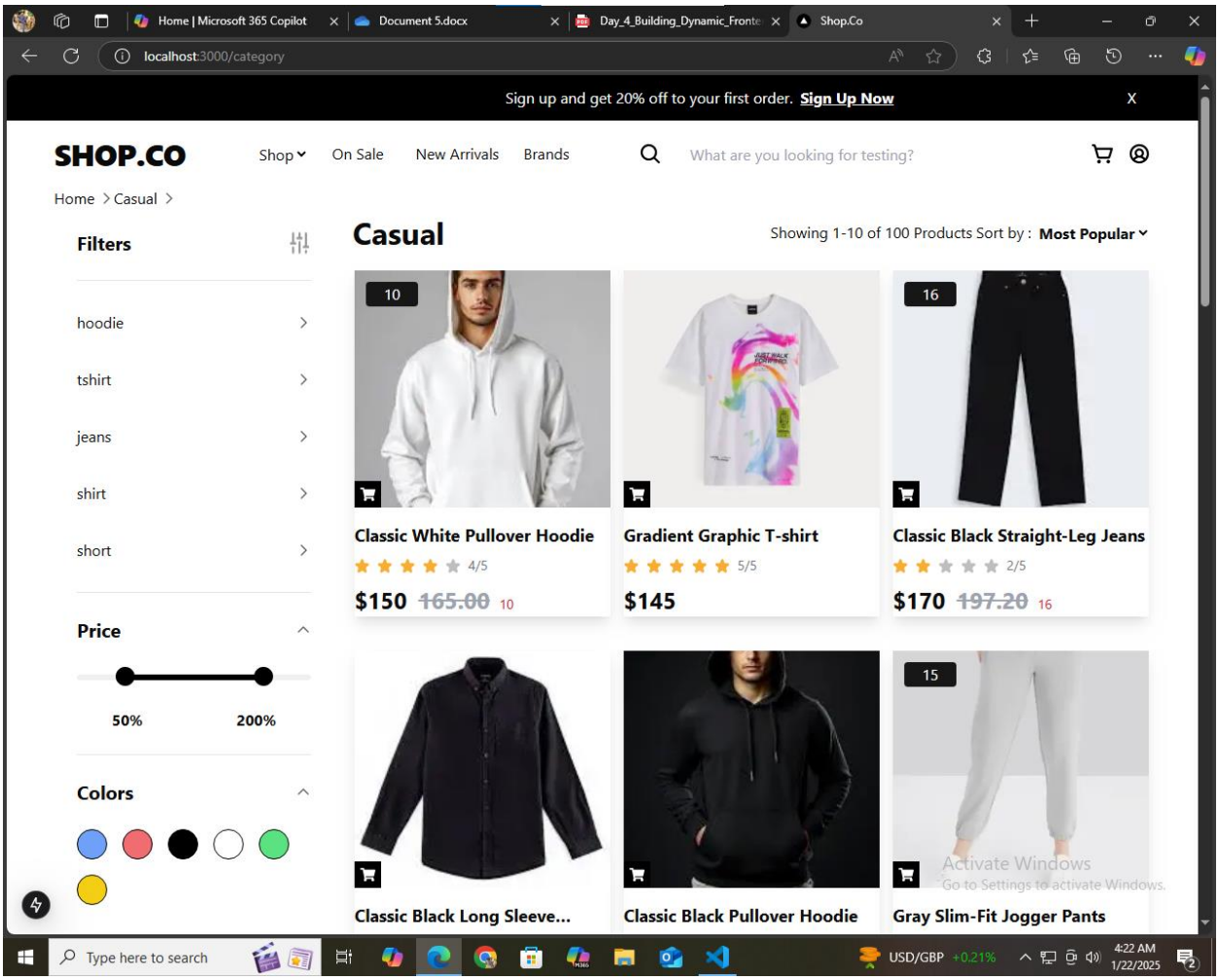
Day 4: Building Dynamic Front-end Components

1. Functional Deliverables:

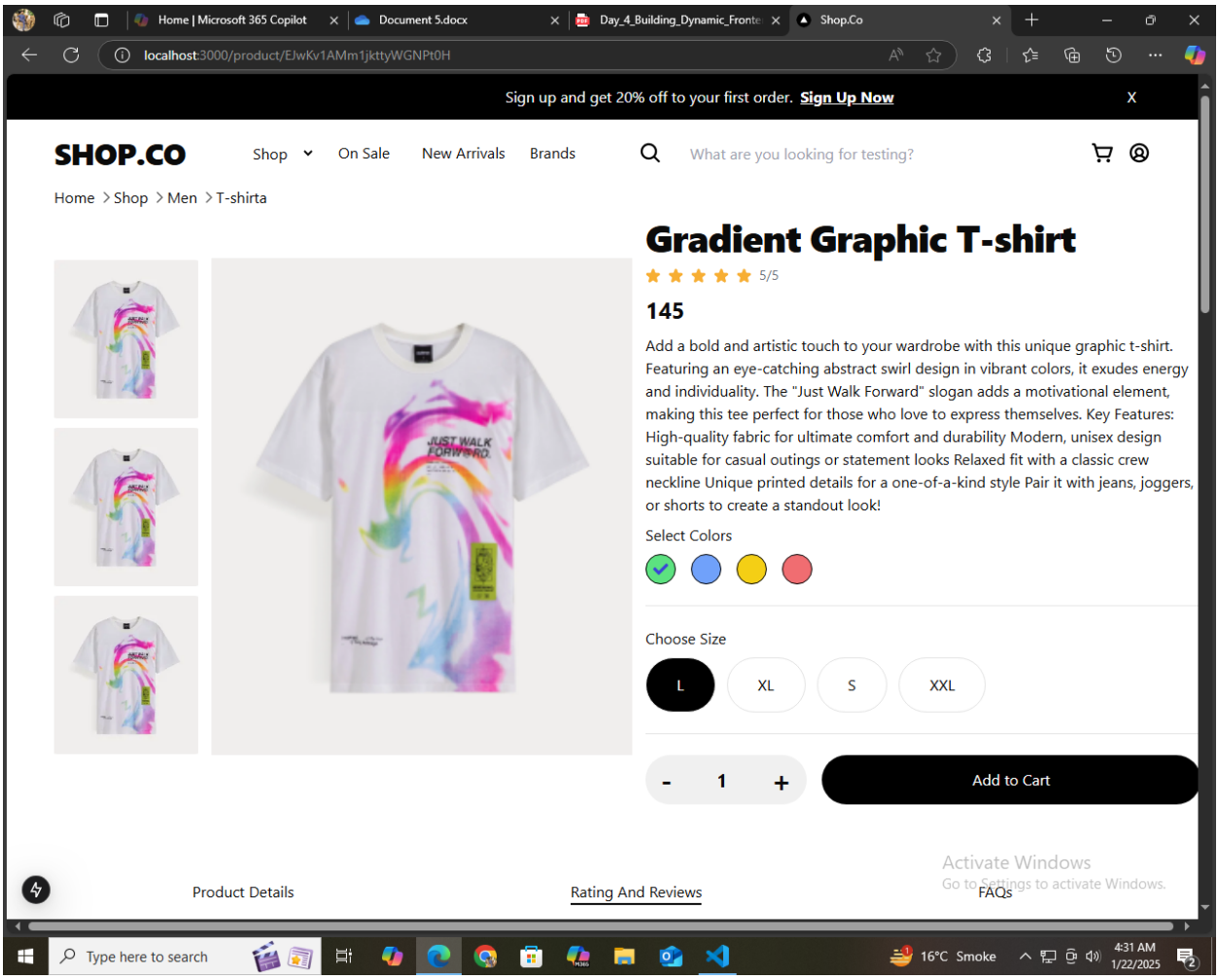
Product Listing showing Data at Home Page:



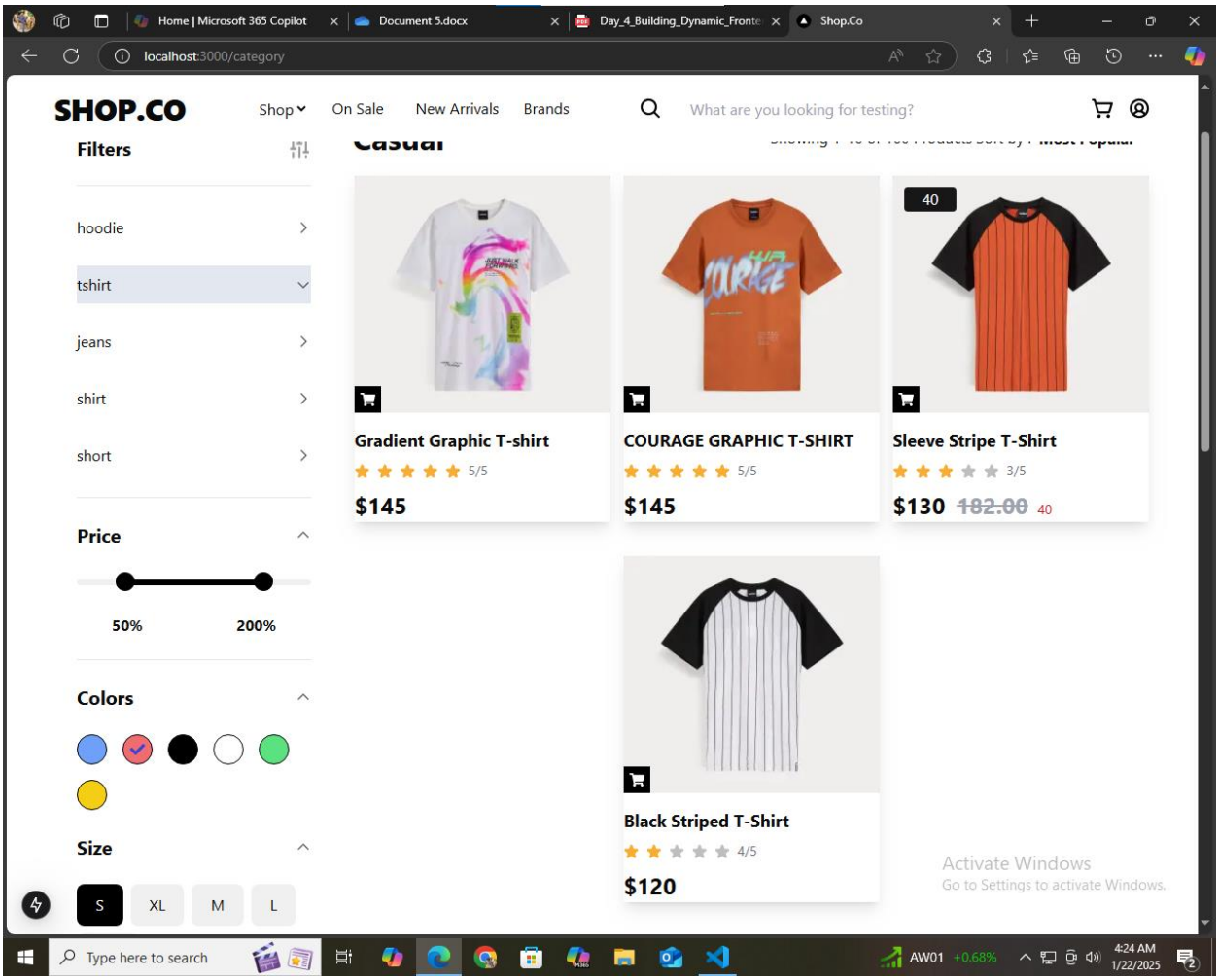
Product Listing Showing Categories Page:



Product Listing Showing Dynamic Details Page of Individual Product:

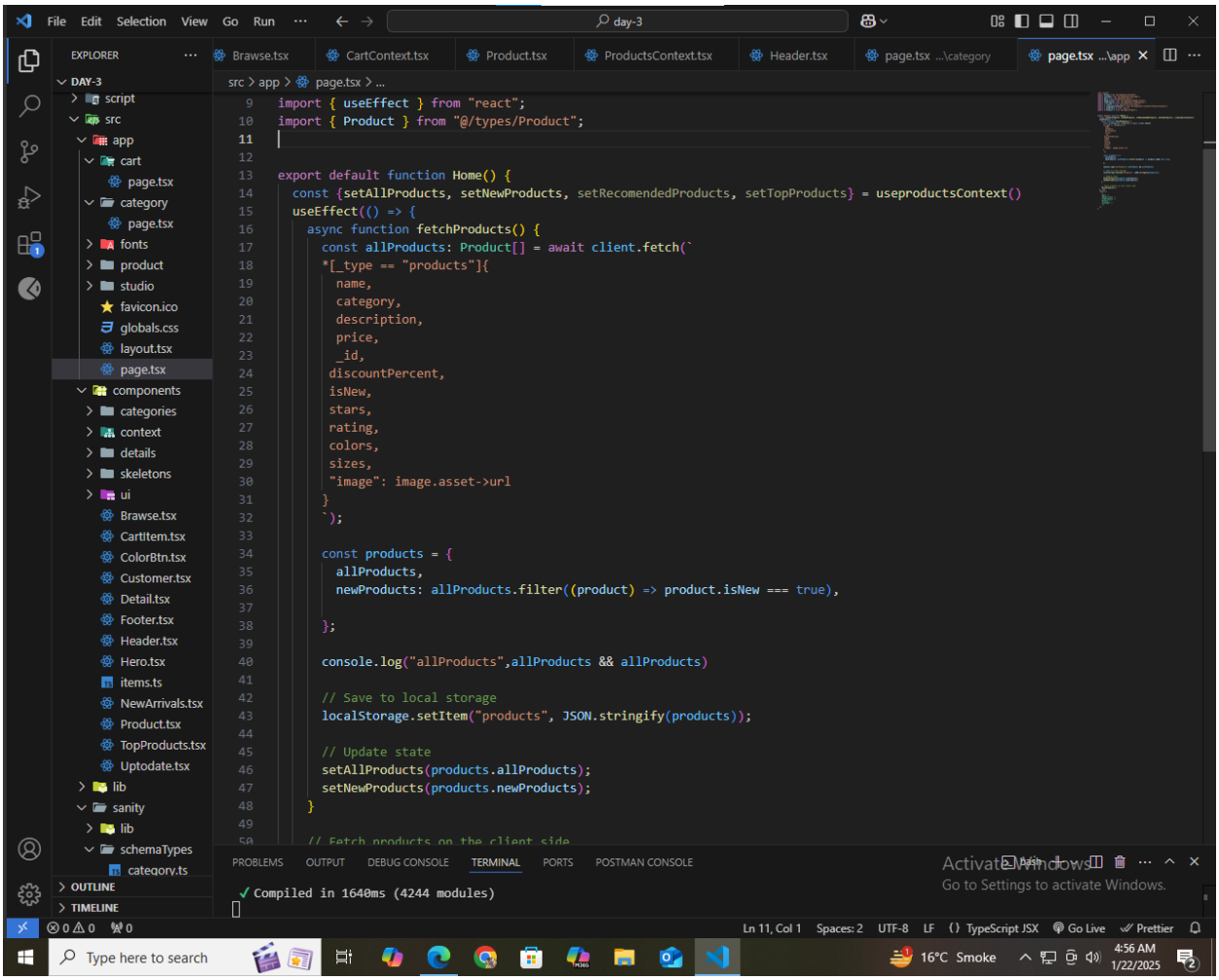


Product Listing of Filtered Data:

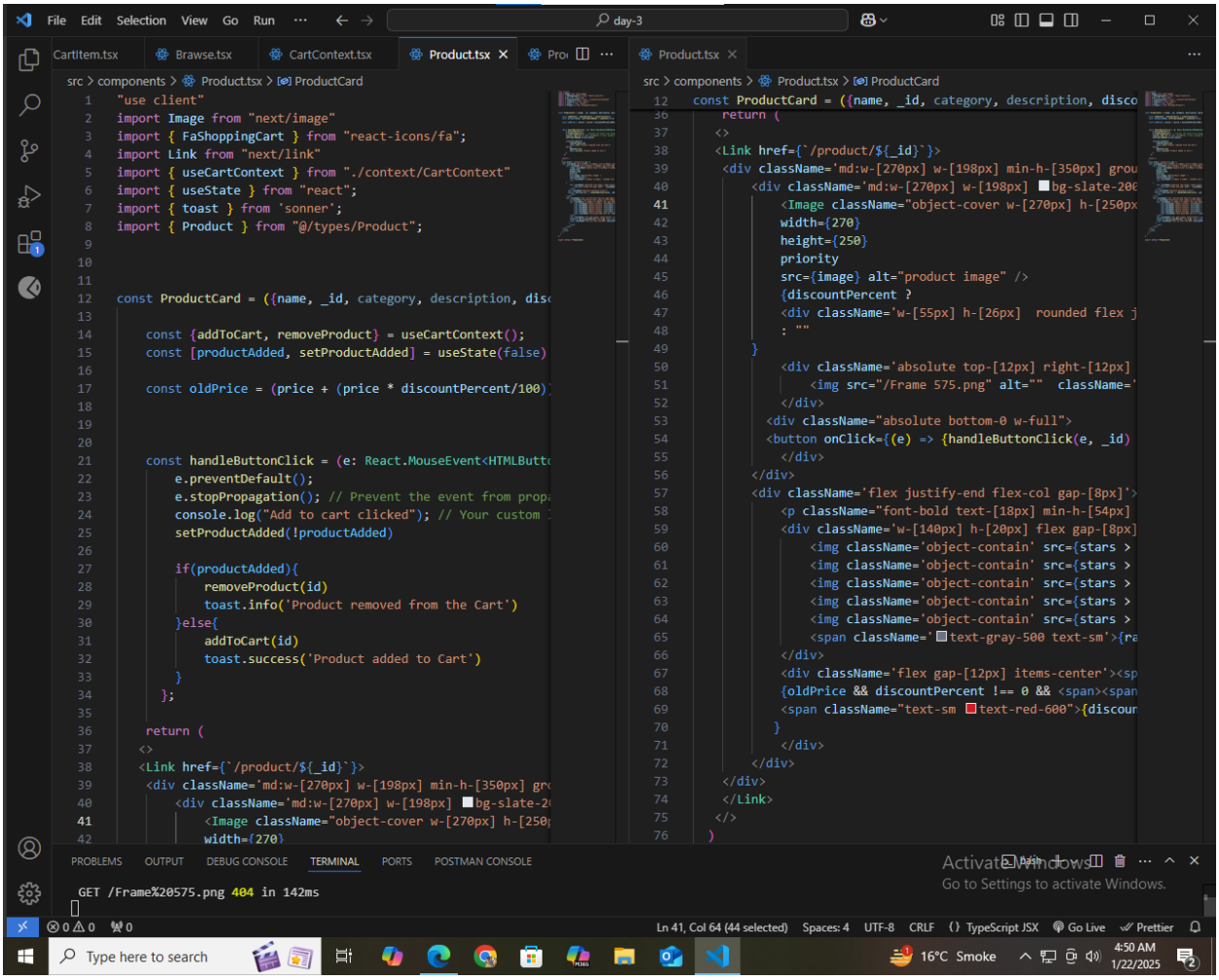


2. Code Deliverables:

Code Snippet Api Integration:



Code Snippet Product Card Component:



Code Snippet Product Context Component:

```
"use client"
import { createContext, Dispatch, FC, ReactNode, SetStateAct
import type { Product } from "@/types/Product";

interface productsContextType{
      allProducts : Product[];
      setAllProducts : Dispatch<SetStateAction<Product[]>>
      newProducts : Product[];
      setNewProducts : Dispatch<SetStateAction<Product[]>>
      topProducts : Product[];
      setTopProducts : Dispatch<SetStateAction<Product[]>>
      recomendedProducts : Product[];
      setRecomendedProducts : Dispatch<SetStateAction<Product[
      allAvaibleTypes : string[];
      searchFilter:string
      setSearchFilter : Dispatch<SetStateAction<string>>
}


//create context
export const productsContext = createContext<productsContext

export const useproductsContext = () => {
  const context = useContext(productsContext);
  if (!context) {
    throw new Error("useCartContext must be used within a Ca
  }
  return context;
};

const ProductsContextProvider : FC<{children : ReactNode}> =
  const [allProducts, setAllProducts] = useState<Product[]>(
  const [newProducts, setNewProducts] = useState<Product[]>(
  const [topProducts, setTopProducts] = useState<Product[]>(
  const [recomendedProducts, setRecomendedProducts] = useSta
  const [allAvaibleTypes, setallAvaibleTypes] = useState<str
  const [colorCategory, setColorCategory] = useState<string[
  const [sizeCategory, sizeColorCategory] = useState<string[
  const [searchFilter, setSearchFilter] = useState('')


  useEffect(()=>{
    if(allProducts.length !== 0){
    function getCategories(){
      const categories = Array.from( new Set(allProducts.map
      const colorCategory = Array.from( new Set(allProducts.
      setallAvaibleTypes(categories)
    }
    getCategories()
  }
  },[allProducts])

    return(
        <productsContext.Provider value={{setSearchFilter, s
            {children}
        </productsContext.Provider>
    )
}

export default ProductsContextProvider;
```

Code Snippet Cart Context Component:

```
"use client"
import {Cart} from "@/types/Cart"
import { useproductsContext } from "@/components/context/Pro
import { createContext, Dispatch, FC, ReactNode, SetStateAct



interface CartContextType {

  addToCart: (id: string) => void;
  removeProduct: (id: string) => void;
  incrementAmount : (id : string) => void;
  decrementAmount : (id : string) => void;
  setAmount: Dispatch<SetStateAction<number>>;
  amount : number
  cart: Cart[]
  setCart : Dispatch<SetStateAction<Cart[]>>
  totalCostPrice : number
  netPrice : number
  discount : number
}



//create context
export const cartContext = createContext<CartContextType | u

export const useCartContext = () => {
  const context = useContext(cartContext);
  if (!context) {
    throw new Error("useCartContext must be used within a Ca
  }
  return context;
};


const CartContextProvider : FC<{children : ReactNode}> = ({cl
  const {allProducts} = useproductsContext()
  const [cart, setCart] = useState<Cart[]>([])
  const [color, setColor] = useState("Blue")
  const [size, setSize] = useState("medium")
  const [totalCostPrice, settotalCostPrice] = useState(0)
```

```
const CartContextProvider : FC<{children : ReactNode}> = ({cl
  const [totalCostPrice, settotalCostPrice] = useState(0)
  const [netPrice, setNetPrice] = useState(0)
  const [discount, setDiscount] = useState(0)
  const [amount, setAmount] = useState(1)

  function addToCart(id : string) : void {
    if(allProducts.length !== 0){
   const addProduct  = allProducts.find((product) => product
   const productExists = cart.find((product) => product._id
  if(!productExists && addProduct){
    setCart((prev)=>(
      [
        ...prev,
        {...addProduct, size, amount, color}
      ]
    ))
  }
}
  }

  function removeProduct(id: string){
    const newList = cart.filter((product) => product._id !==
    setCart(newList)
  }

  function incrementAmount(id : string){
    setCart((prev) =>(
      prev.map((product) => (product._id === id ? {...produc
    ))
  }

  function decrementAmount(id : string){
    setCart((prev) =>(
      prev.map((product) => (product._id === id ? {...produc
    ))
  }



function handleTotalPrice(){
```

```
GET /Frame%20575.png 404 in 81ms
```

3. Documentations:

Steps Taken to Build and Integrate Components:

In my project, I have developed several components to streamline the application development process. Key components include:

- **ProductCard**: Displays product details such as images, titles, and prices.
- **ProductContext and CartContext**: Manage application state for product data and shopping cart functionality, respectively.
- **Header and Footer**: Provide consistent navigation and branding across the application.

These components are designed with the goals of reusability, improved organization, and easier maintenance, ensuring a cleaner and more scalable codebase.

Process of Component Development:

1. **Structuring the Component Design:** Each component's role and responsibilities were carefully planned:
    a. **UI Components**: Elements like ProductCard, Header, and Footer focus on presenting information and user interaction.
    b. **State Management Components**: ProductContext and CartContext centralize data handling to avoid prop drilling and ensure efficient state management.
2. **Prioritizing Reusability:** To enhance reusability, components were designed with flexibility in mind:
    a. **Dynamic Props**: Components like ProductCard accept props for data, enabling their use in various contexts without modification.

   b. **Generic Elements**: Styles and layouts were kept generic to support diverse use cases.
3. **Efficient Integration:** Integration was carried out methodically:
   a. **Context Wrapping**: State management contexts were wrapped around the application's component tree to provide data access across components.
   b. **Consistent Layouts**: Header and Footer were integrated as persistent components, framing dynamic content rendered between them.
4. **Testing and Refinement:** Each component underwent rigorous testing:
   a. **Unit Testing**: Verified individual functionality.
   b. **Integration Testing**: Ensured smooth interaction between components.
   c. **Feedback-Driven Adjustments**: Incorporated improvements based on testing outcomes and user feedback.

Key Advantages of This Approach:

- **Improved Code Organization**: Clear separation of responsibilities enhances readability and maintainability.
- **Ease of Updates**: Modular components allow changes to specific parts without affecting the entire system.
- **Collaborative Development**: Components can be developed and tested independently, supporting team-based workflows.
- **Future Scalability**: The reusable and maintainable design supports the application's growth and adaptation to new requirements.
- **Ease of Maintenance**: Changes in one component can be implemented without impacting others, reducing the likelihood of introducing bugs.
- **Improved Collaboration**: Modular components make it easier for team members to work on different parts of the project simultaneously.

- **Scalability**: The use of reusable components and context management ensures that the application can grow and adapt to new requirements efficiently.

Through this approach, I have achieved a well-structured, maintainable, and scalable codebase that supports both current needs and future enhancements.

Challenges Faced and Solutions Implemented:

One of the primary challenges encountered during the development process was managing the application state effectively across various components. Ensuring that data could be accessed and updated seamlessly in a scalable way was critical to the project's success.

State Management Challenges:

Initially, passing data between components using props led to unnecessary complexity, increased redundancy, and made the code harder to maintain. Prop drilling not only cluttered the code but also limited the scalability of the application as new features were added.

Solutions Implemented:

To address these issues, I leveraged the `useContext` hook for efficient state management. Two key context files were created:

- **CartContext.tsx**: This context provides shopping cart data to all components across the project. By centralizing cart state, components such as the checkout page, cart display, and product detail views can access and update the cart data seamlessly.
- **ProductContext.tsx**: This context supplies product data to components. The data is fetched at the root level in `page.tsx` from Sanity CMS, ensuring that all child components can access the same

dataset without redundant API calls. This reduces overhead and improves performance.

Integration Process:

1. **Creating Contexts**: The CartContext and ProductContext were structured to encapsulate state logic and provide easy access through context providers.
    a. Example: The CartContext manages cart items, providing methods to add, remove, or update items.
2. **Wrapping Application with Providers**: The context providers were integrated into the application's component tree, ensuring that all components could consume the context data.

This approach eliminated prop drilling, simplifying the code and enhancing maintainability.

Testing and Optimization:

- Verified the functionality of each context using mock data.
- Monitored performance to ensure that context usage did not introduce unnecessary re-renders.

Outcome and Benefits:

- **Streamlined Data Flow**: Centralized state management improved the consistency and reliability of data across components.
- **Reduced Code Complexity**: Eliminating prop drilling resulted in a cleaner and more organized codebase.
- **Improved Scalability**: The context-based solution allows for easy addition of new features or components without requiring significant refactoring.

- **Enhanced Performance**: By fetching product data at the root level and sharing it via ProductContext, redundant API calls were minimized, reducing load times.

This approach to state management not only resolved the initial challenges but also laid the foundation for a scalable and maintainable application architecture.

Best Practices Followed During Development:

Adhering to best practices was pivotal in ensuring the success of the project. Key practices included:

- **Skeleton Components for Loading States**: To improve user experience during data fetching, skeleton components were implemented as placeholders. These visually indicate loading states, ensuring the interface remains interactive and visually consistent while data is being retrieved.
    - Example: Skeleton loaders were used in the ProductCard component to show a placeholder for product details until the actual data is fetched and rendered.
- **Dynamic Component Development**: Components were designed with a dynamic approach, creating as many reusable and modular components as necessary to reduce duplication and enhance maintainability.
    - Example: Components such as ProductCard and Header were developed to handle varying data inputs through props, allowing them to be reused across different sections of the application.

Benefits of These Practices:

- **Enhanced User Experience**: Skeleton components keep users informed and engaged during loading periods.

- **Improved Code Maintainability**: The dynamic approach minimizes redundancy and simplifies updates or modifications.
- **Scalability**: Modular and reusable components enable easy integration of new features without significant rework.

By adhering to these practices, the project achieved a polished, professional interface with a robust and maintainable codebase.