

Day 4

Dynamic Frontend Components– [Furniro] By M.Anees

Furniro is a top-class furniture website dedicated to providing ease and convenience for both shopkeepers and customers.

For local sellers, Furniro offers a platform to reach a broader audience by leveraging the power of websites and the internet. This helps them showcase their furniture collections online, increasing visibility and sales opportunities.

On the other hand, customers can enjoy the convenience of browsing a wide variety of furniture options from the comfort of their homes. Furniro eliminates the hassle of visiting 30 to 50 shops by offering an extensive catalog, allowing users to compare prices, styles, and quality effortlessly.

Our goal is to bridge the gap between sellers and buyers, creating a seamless shopping experience that saves time, effort, and money.

Whether you're looking for budget-friendly options or premium furniture, Furniro is here to help you make the best choices without stepping out of your home.

Challenge of the day

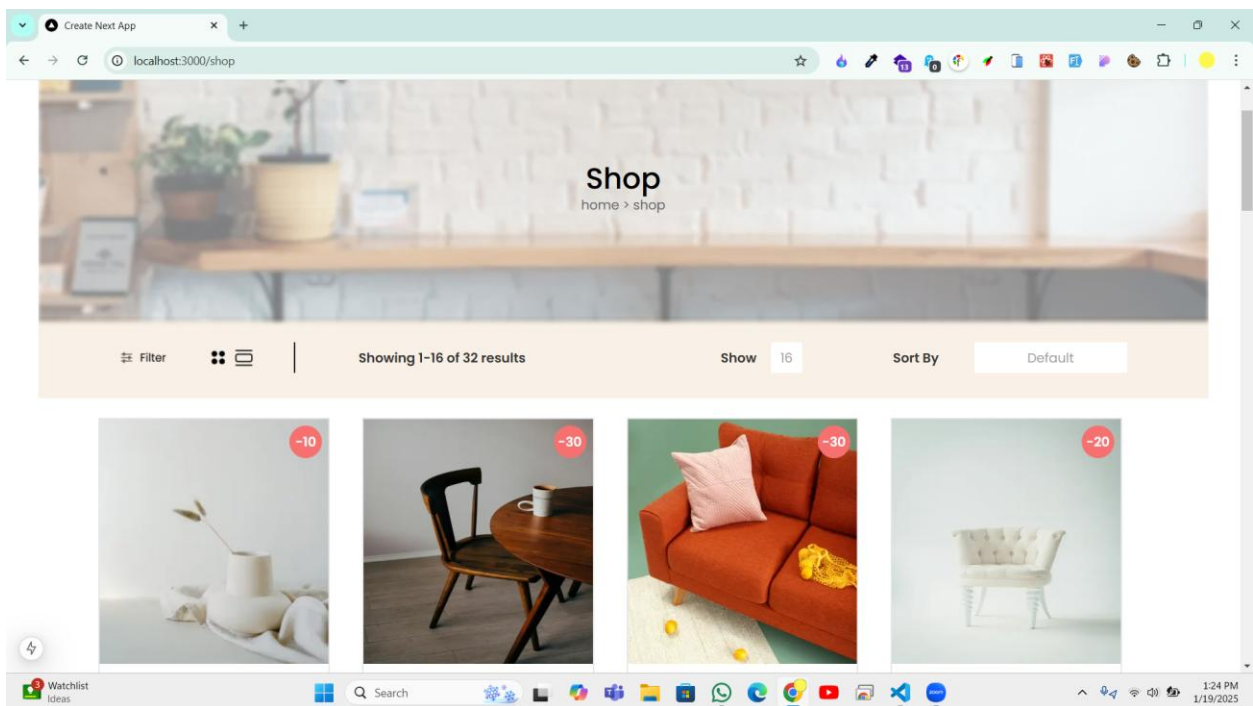
Today, I focused on developing the dynamic frontend components for my marketplace application as part of my assigned task for Day 4. The goal was to ensure the implementation of interactive and functional features that enhance the user experience. Below, I've detailed the progress and deliverables.

Main Components of the Day

The focus for Day 4 was on building and refining dynamic frontend components to enhance the marketplace application. The tasks included implementing core features such as product listing, detailed product pages, filtering options, Cart Functionality, Wishlist Sidebar and seamless navigation, ensuring a functional and user-friendly interface.

1. Product Listing Page

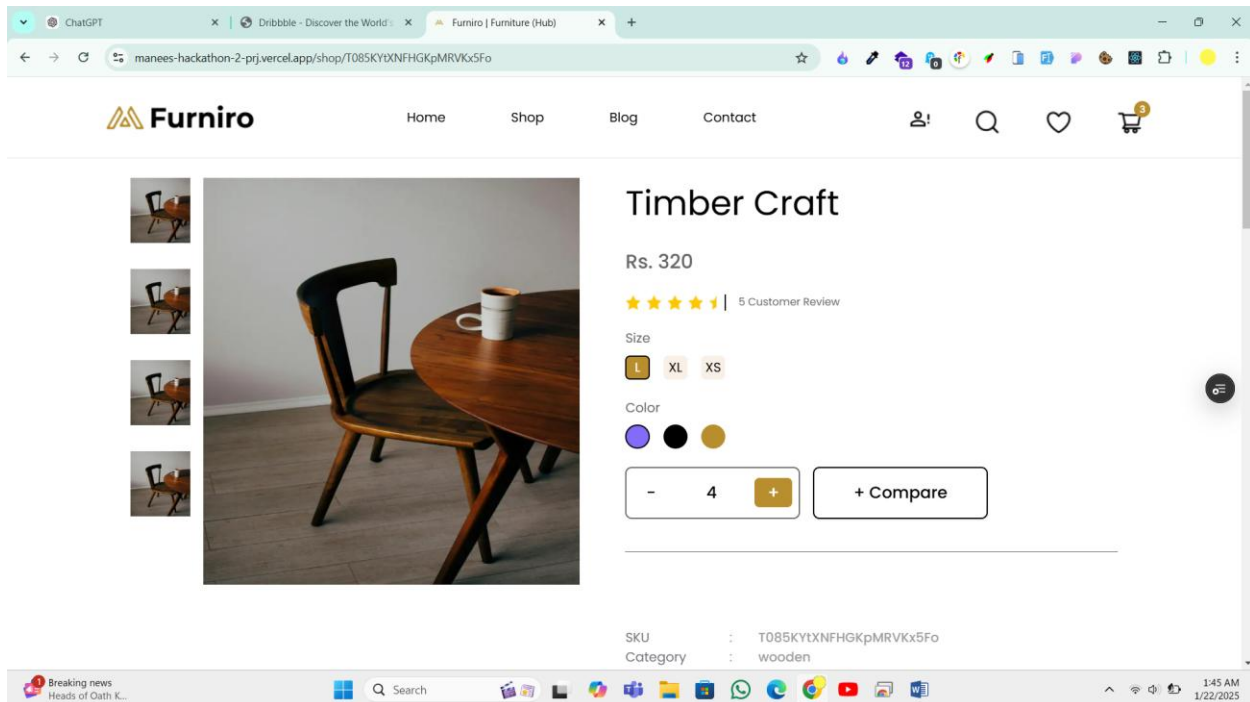
I began by implementing the product listing page, ensuring it dynamically fetched data from the API. Each product was displayed with accurate information, including the name, image, price, and description. The layout was designed to be responsive, providing a seamless experience across different devices.



- i) Designed a dynamic product listing page with data fetched via API integration.
- ii) Displayed key details for each product, including name, image, price, and description, in a responsive layout.

2. Individual Product Detail Pages

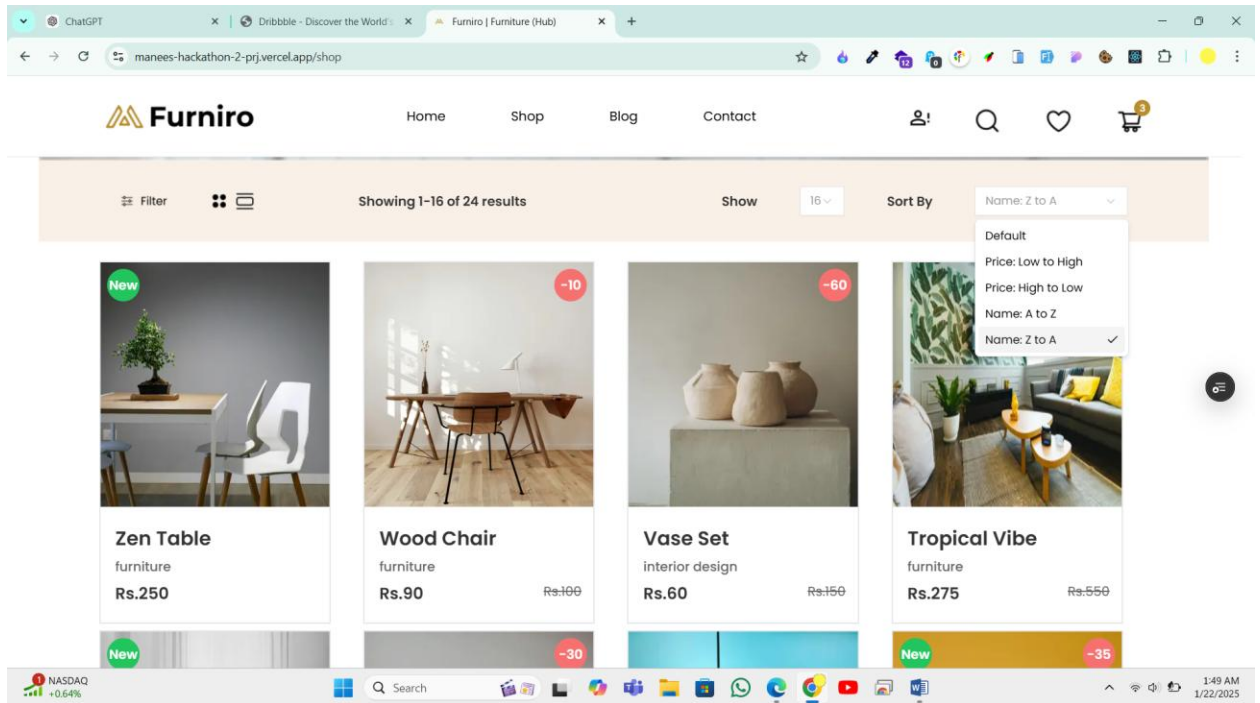
I set up dynamic routing to ensure each product had its own detail page. These pages rendered product-specific data accurately, such as specifications and availability. Users can now navigate between the listing and detail pages effortlessly.



- i) Implemented accurate routing to dynamically render detail pages for each product.
- ii) These pages included comprehensive product information, ensuring users could easily access specific details.

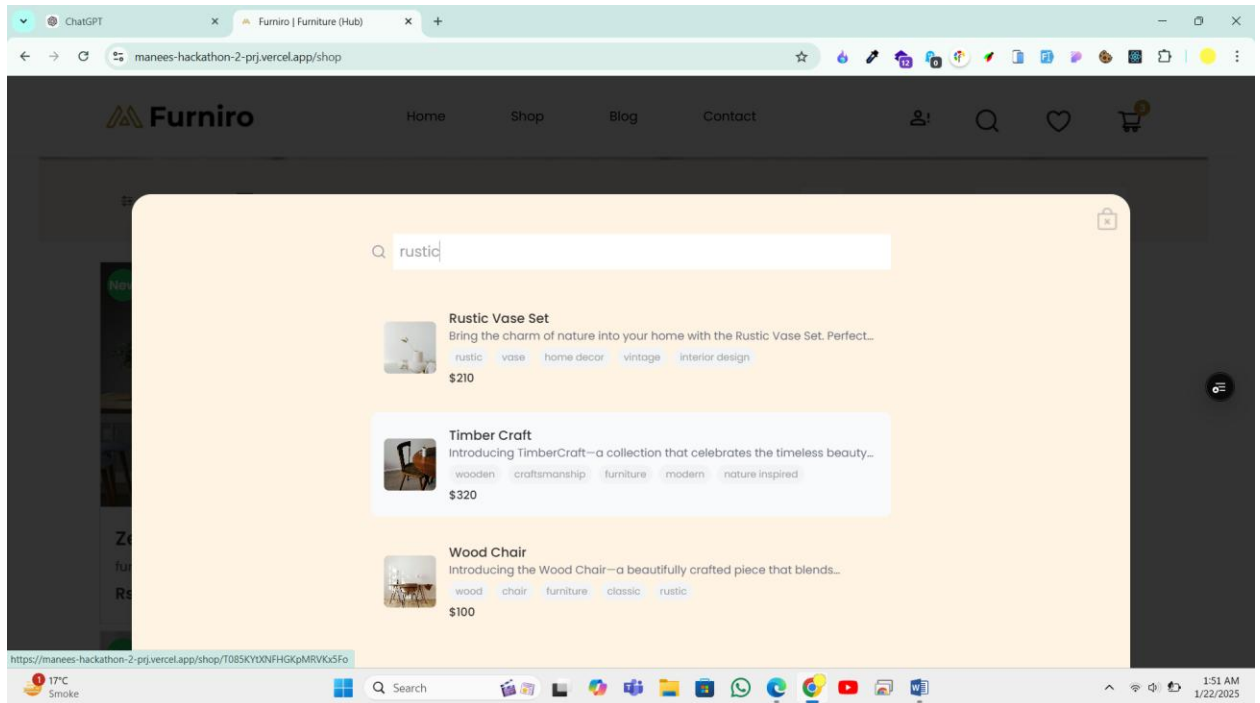
3. Category Filters

Developed a category filtering system that dynamically updated the product list based on user selections.



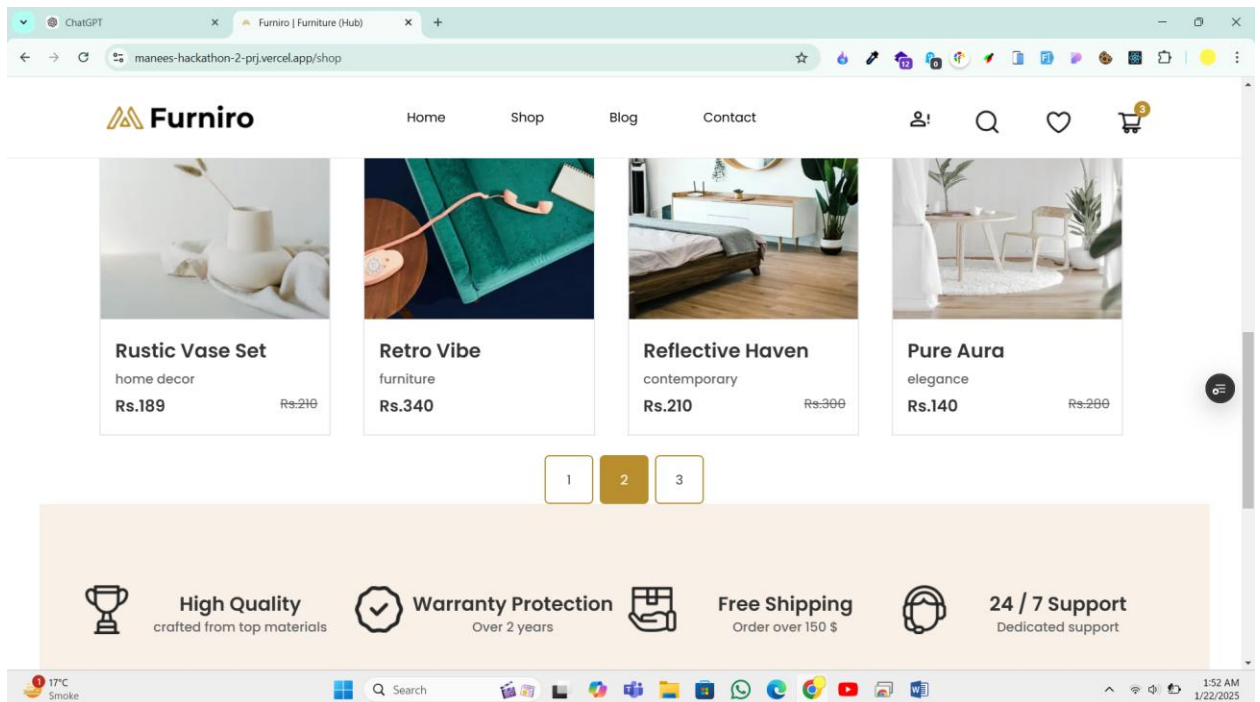
4. Search Bar

I implemented real-time search functionality, where users can type queries and see instant updates in the product list.



5. Pagination:

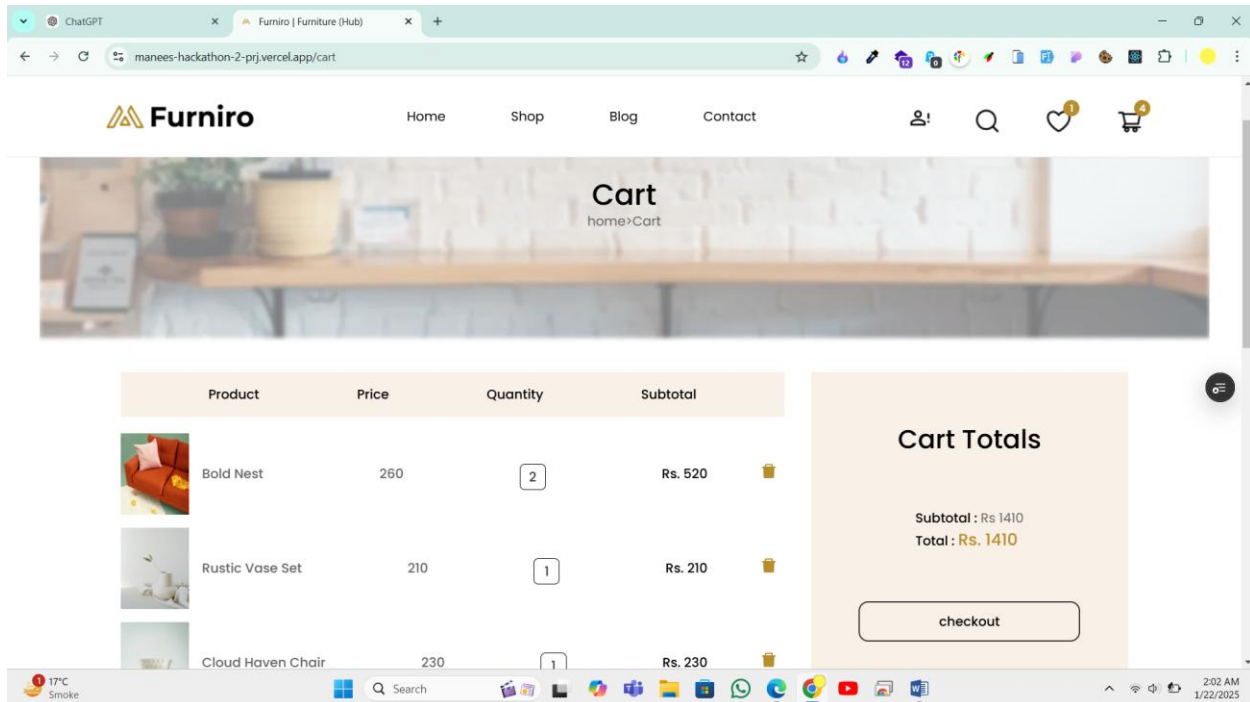
For datasets with numerous entries, pagination was introduced to allow smooth browsing without overwhelming the interface.



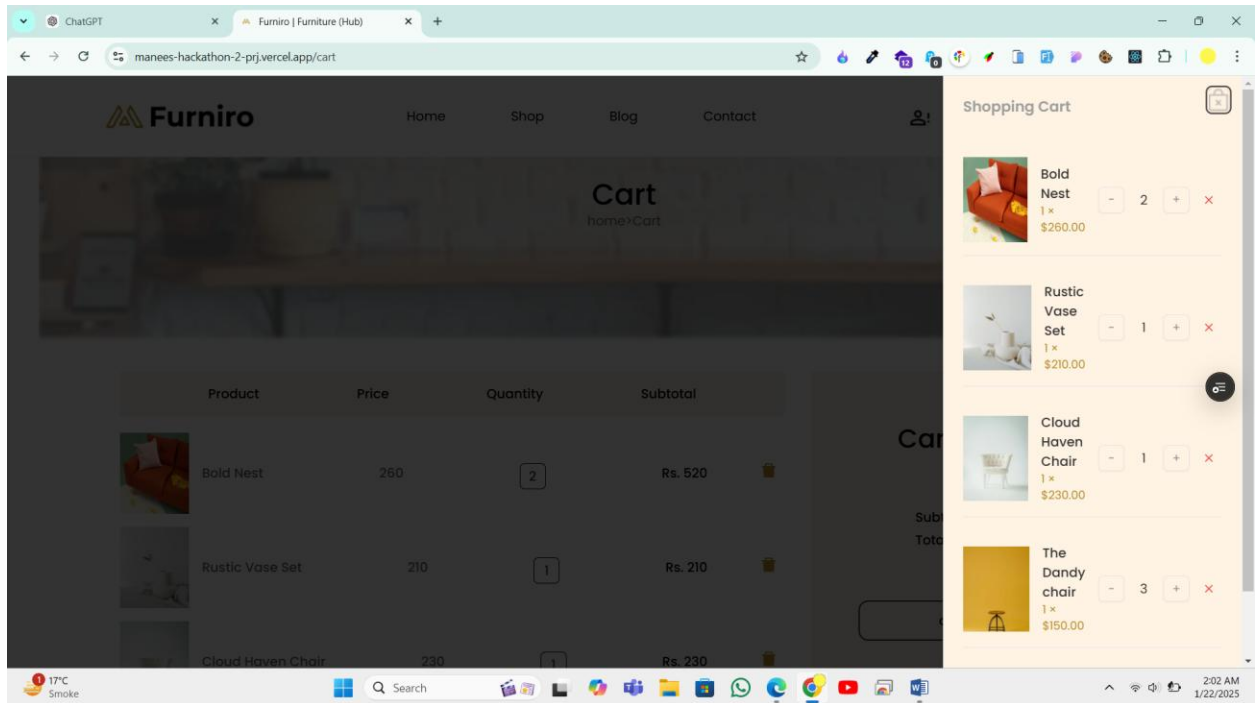
6. Add-to-Cart Route and sidebar.

1) Implemented an **Add to Cart** route and Sidebar to handle cart operations efficiently.

2) Ensured that products could be added to the cart dynamically and displayed on the cart page using Context API for state management.

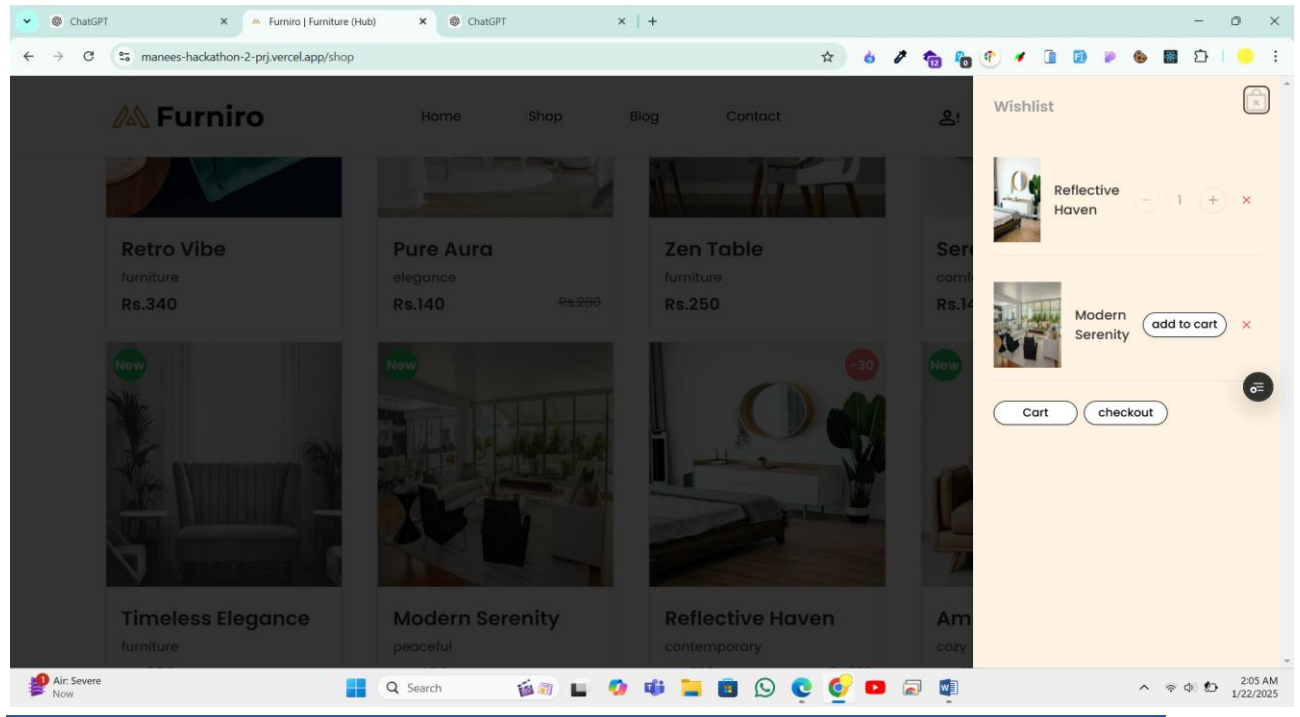


Below is the Visualization of the Cart Sidebar



6. Wishlist Side Bar using Context api.

- Developed a wishlist feature to allow users to save favorite products for later.
- Added a sidebar component for wishlist items, dynamically updated using Context API.



Code snippets for key components

Product Card Code.

The screenshot shows a VS Code editor with a React application. The Explorer sidebar on the left displays the file structure, including components, hooks, and pages. The main editor area shows the CardProd.jsx file, which contains a React component for a product card. The component uses useState for a wishlist, useEffect for toast notifications, and a router.push for navigation. The JSX part of the component renders a product card with a title, price, discount, and a wishlist toggle. It also includes a 'View More' button that triggers a console log. The styling is done using Tailwind CSS classes.

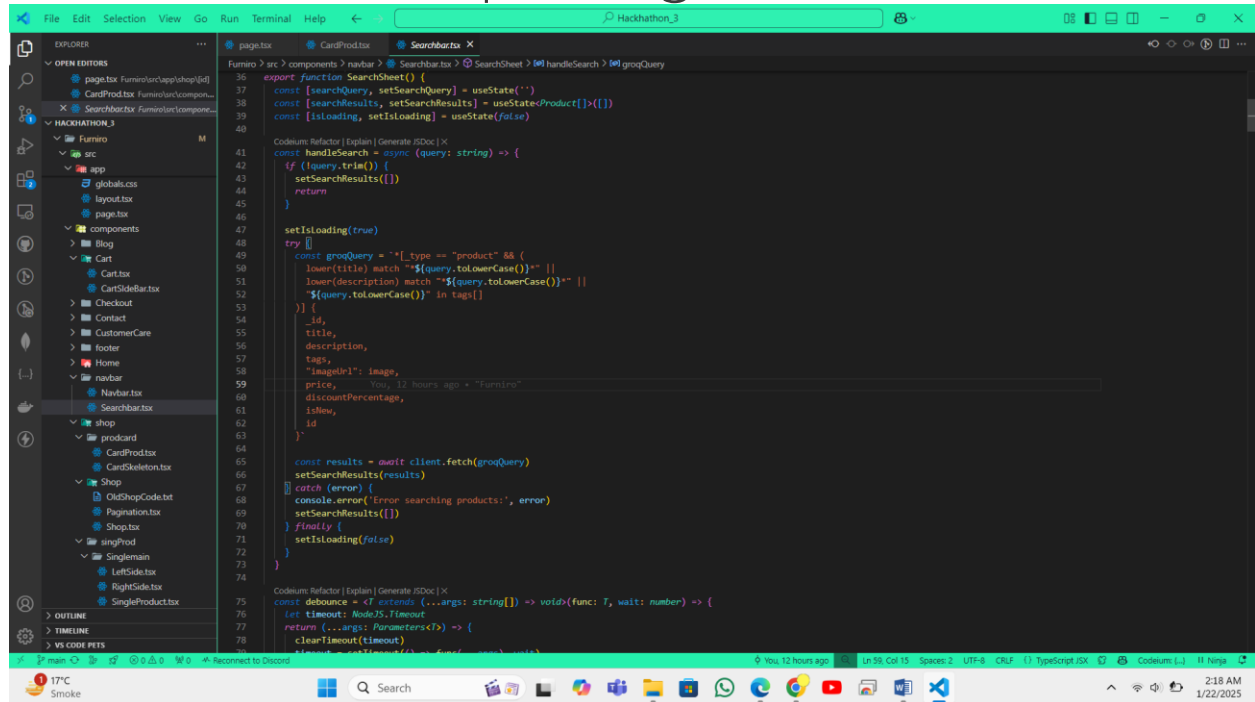
[illegible]

The screenshot displays a VS Code editor window with a React application. The Explorer sidebar on the left shows the project structure, including a 'components' folder containing 'Searchbar.jsx'. The main editor area shows the code for 'Searchbar.jsx', which includes a stateful search function and a list of products. The code is as follows:

```
35 import React, { useState } from 'react';
36
37 export function SearchSheet() {
111   const [searchQuery, setSearchQuery] = useState('');
112   const [searchResults, setSearchResults] = useState([]);
113   const onChange = (e) => {
114     setSearchQuery(e.target.value);
115     debouncedSearch(e.target.value);
116   };
117   const ariaLabel = "Search products"
118   useEffect(() => {
119     // ...
120   }, [searchQuery]);
121   </SheetHeader>
122
123   <div
124     className="mt-8 px-4 max-w-2xl mx-auto overflow-y-auto max-h-[calc(80vh-120px)]"
125     role="region"
126     aria-label="Search results"
127   >
128     {isLoading ? (
129       <div className="text-center py-6" > text-gray</div>
130       Searching...
131     ) : (
132       <div>
133         {searchResults.length > 0 ? (
134           <div className="space-y-4">
135             {searchResults.map(product => (
136               <SheetTrigger asChild key={product.id}>
137                 <a href="#"> /shop/${product.id}</a>
138             ))}
139             <div className="block">
140               <div className="flex items-center gap-4 p-4" >
141                 <div className="relative h-16 w-16 rounded-ad overflow-hidden" >
142                   <img
143                     src={getProductImageId(product)}
144                     alt={Product image of ${product.title}}
145                     className="object-cover"
146                   />
147                 <div>
148                   <div className="font-medium">{product.title}</div>
149                   <div>{product.description}</div>
150                 </div>
151               </div>
152             </div>
153           </div>
154         ) : (
155           <div className="text-sm"> No results found for "${searchQuery}"</div>
156         )
157       </div>
158     )}
159   </div>
160 }
161
162 export default SearchSheet;
```

The code defines a `SearchSheet` component that manages a search query and displays a list of products. It uses `useState` for state management, `useEffect` for debouncing the search, and `useRef` for a text input. The UI is styled with Tailwind CSS, featuring a search bar with a placeholder, a search button, and a list of products with images, titles, and descriptions. The search results are displayed in a scrollable container.

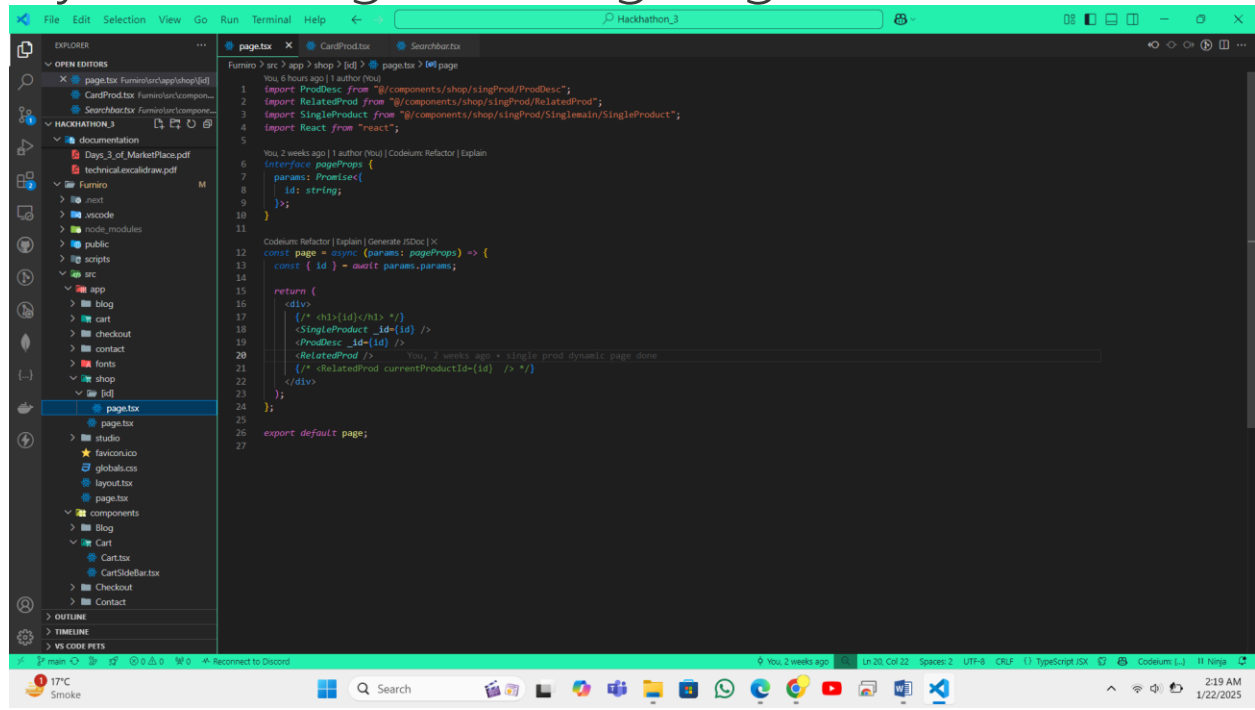
A overview of api integration code



The screenshot shows a VS Code editor with the Explorer sidebar on the left displaying a file tree for a project named 'Hackathon_3'. The main editor area shows the code for 'SearchSheet.js'. The code defines a 'SearchSheet' component with state for 'searchQuery', 'searchResults', and 'isLoading'. It includes a 'handleSearch' function that uses 'useQuery' to fetch data from a GraphQL endpoint. The fetched data is mapped to a list of product objects. A 'debounce' function is used to handle the search input. The component renders a search input, a loading spinner, and a list of product cards.

```
36 export function SearchSheet() {
37   const [searchQuery, setSearchQuery] = useState('')
38   const [searchResults, setSearchResults] = useState<Product[]>([])
39   const [isLoading, setIsLoading] = useState(false)
40
41   const handleSearch = useQuery<query: string> => {
42     if (!query.trim()) {
43       setSearchResults([])
44       return
45     }
46
47     setIsLoading(true)
48     try {
49       const groqQuery = `*_type == "product" && (
50         lower(title) match "${query.toLowerCase()}" ||
51         lower(description) match "${query.toLowerCase()}" ||
52         "${query.toLowerCase()}" in tags
53       ) {
54         _id,
55         title,
56         description,
57         tags,
58         "image[1]": image,
59         price,
60         "You, 12 hours ago" * "Furniro"
61       }`
62
63       const results = await client.fetch(groqQuery)
64       setSearchResults(results)
65     } catch (error) {
66       console.error("Error searching products:", error)
67       setSearchResults([])
68     } finally {
69       setIsLoading(false)
70     }
71   }
72
73   const debounce = <T extends (...args: string[]) => void>(func: T, wait: number) => {
74     let timeout: NodeJS.Timeout
75     return (...args: Parameters<T>) => {
76       clearTimeout(timeout)
77       timeout = setTimeout(() => {
78         func(...args)
79       }, wait)
80     }
81   }
82 }
```

Dynamic Page Coding Logic.



The screenshot shows a VS Code editor with the Explorer sidebar on the left displaying a file tree for a project named 'Hackathon_3'. The main editor area shows the code for 'page.js'. The code defines a 'page' component that takes 'pageProps' as a prop. It uses 'useQuery' to fetch data from a GraphQL endpoint. The fetched data is mapped to a list of product objects. The component renders a search input, a loading spinner, and a list of product cards.

```
1 import Product from "@components/shop/singProd/ProductDesc";
2 import RelatedProd from "@components/shop/singProd/RelatedProd";
3 import SingleProduct from "@components/shop/singProd/Singlemain/SingleProduct";
4 import React from "react";
5
6 You, 2 weeks ago | 1 author (You) | Codium Refactor | Explain
7 interface PageProps {
8   params: Promise<{
9     id: string;
10   }>;
11 }
12
13 const page = async (params: PageProps) => {
14   const { id } = await params.params;
15
16   return (
17     <div>
18       {/* <h1>{id}</h1> */}
19       <SingleProduct _id={id} />
20       <ProductDesc _id={id} />
21       {/* <h1>{id}</h1> */}
22       <RelatedProd currentProductId={id} />
23     </div>
24   );
25 }
26
27 export default page;
```

Steps Taken to Build and Integrate Components

1. **Dynamic Product Listing Page**
 - Initiated API integration to fetch product data and displayed it dynamically in a grid layout.
 - Utilized reusable components like ProductCard to ensure consistency and modularity across the application.
2. **Individual Product Detail Pages**
 - Configured dynamic routing to render product-specific detail pages.
 - Designed these pages to display comprehensive product details, including descriptions, specifications, and availability.
3. **Category Filters and Search Bar**
 - Implemented category-based filtering to dynamically update the product listing.
 - Developed a real-time search bar using controlled inputs, ensuring instant results.
4. **Pagination System**
 - Added pagination to handle large datasets efficiently, improving page performance and user experience.
5. **Add-to-Cart Route**
 - Designed and implemented the Add-to-Cart feature, ensuring smooth functionality and state persistence.
 - Managed the cart state using Context API, allowing global access to the cart data.
6. **Sidebar and Wishlist Sidebar with Context API**
 - Developed a collapsible sidebar for navigation, with state managed using Context API.
 - Created a wishlist sidebar to store and display favorite products dynamically.

Challenges Faced and Solutions Implemented

1. **Challenge: State Management for Filters and Cart**
 - **Issue:** Managing state across components, especially for filters, cart, and wishlists.
 - **Solution:** Utilized Context API for centralized state management, ensuring seamless data flow and synchronization.
2. **Challenge: API Latency**
 - **Issue:** Slow response times during data fetching led to delays in displaying content.
 - **Solution:** Implemented loading indicators and debouncing for search functionality to enhance user experience.

3. **Challenge: Large Dataset Pagination**

- **Issue:** Rendering large datasets caused performance lags.
- **Solution:** Implemented pagination to fetch and display data in smaller,

Best Practices Followed During Development

1. **Modular Design**

- Components like ProductCard, ProductList, and SearchBar were designed to be reusable, improving maintainability.

2. **Responsive Design**

- Ensured all pages and components adapted seamlessly to various screen sizes for a consistent user experience.

3. **Optimized State Management**

- Leveraged Context API to efficiently manage and share state across components without prop-drilling.

4. **Efficient API Integration**

- Used optimized API calls and caching strategies to reduce latency and improve performance.

5. **Clean and Scalable Codebase**

- Followed clean coding principles and ensured the project was scalable for future enhancements.