Day 4 - Dynamic Frontend Components -Furniture Marketplace

Overview: Developing Dynamic Frontend Components for a Furniture Marketplace

In building a niche furniture marketplace, I focused on creating dynamic, reusable, and responsive frontend components tailored to the unique requirements of e-commerce. The development was powered by **Next.js**, **TailwindCSS**, and **ShadcnUI**, leveraging their combined strengths to deliver a robust and seamless user experience.

Objective

The primary objective was to design and implement a dynamic frontend that could:

1. Efficiently fetch, display, and interact with data from **Sanity** (used as the backend database) via **GROQ queries**.
2. Provide a responsive and visually appealing interface that adapts to diverse user preferences and device types.
3. Ensure modularity and reusability to allow for easy scalability and maintenance.

Challenges Faced

- Managing asynchronous data updates without causing flickers or redundant renders.
- Balancing reusability with customizability in components to cater to various scenarios.
- Debugging complex GROQ queries for real-time database interaction.
- Optimizing performance for large datasets and maintaining responsiveness.

Best Practices Employed

- **Component Modularity:** Designed components with clear boundaries and reusable logic, reducing code duplication and increasing maintainability.
- **State Management:** Leveraged hooks and context API effectively to handle state across components with minimal overhead.
- **Error Handling:** Implemented robust error handling mechanisms for API calls and database queries to improve reliability.
- **Performance Optimization:** Utilized Next.js's built-in optimizations like image handling and server-side rendering (SSR) for faster load times.
- **Accessibility:** Ensured all components adhered to accessibility standards for a more inclusive user experience.

This dynamic approach ensured that the frontend was both user-friendly and developer-friendly, making it adaptable for future enhancements and iterations.

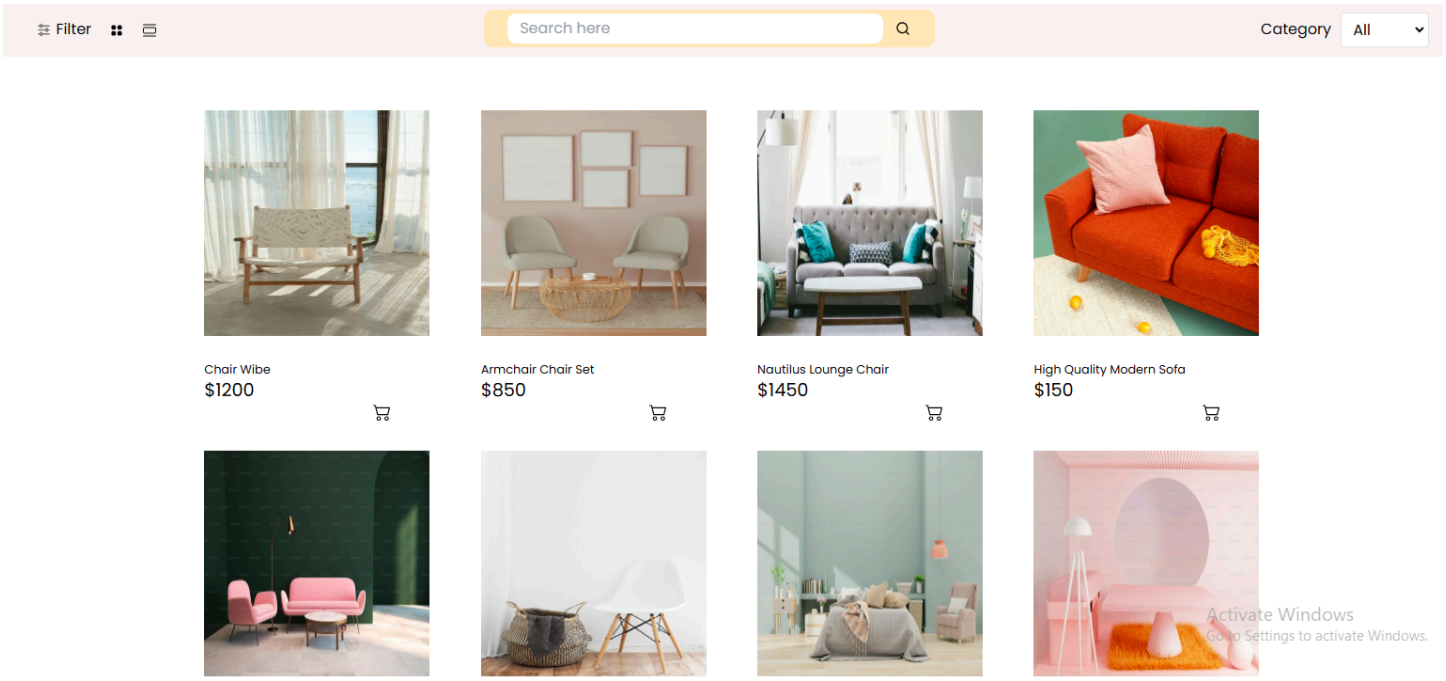
Here are the of few those dynamic components:

Product Listing Component

The Product Listing component dynamically displays a grid of products fetched from the Sanity database. It includes features for filtering and sorting products by categories, allowing users to view active products with images, names, and prices.

Steps:

1. **GROQ Query:**
   Fetch product data (name, price, image, category) from Sanity.
2. **Data Fetching:**
   Use useEffect to fetch data on component mount.
3. **Display Products:**
   Map through the products and render product cards (image, name, price).
4. **Filter & Sort:**
   Implement filtering and sorting options using React state.



Code Snippet:

```jsx
{/* Product List Section */}
<div className="flex max-w-[3000px] items-center justify-center flex-col pt-12 md:pt-16 font-poppin">
    <div className="flex gap-12 flex-wrap w-[90%] justify-center">
        {isLoading ? <p>Loading products..</p> :
            products && products.length > 0 ? (
                products.map((product: product) => {
                    return (
                        <Card2
                            key={Number(product.id)}
                            imageUrl={product.imagePath}
                            text={product.name}
                            id={product.id}
                            Product={product}
                            Cost={product.price}
                        />
                    );
                })
            ) : (
                <p className='text-red-500 text-lg'>No Search Results</p>
            )}
    </div>
```

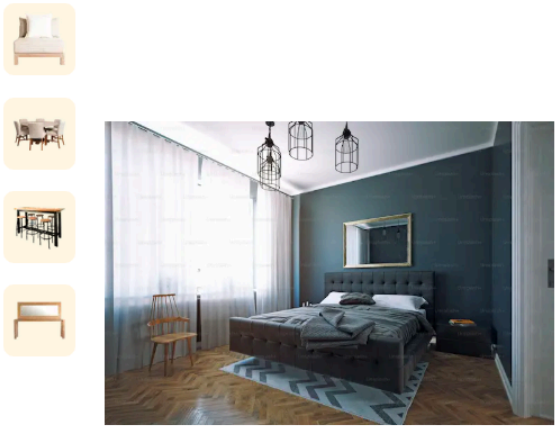
Product Details Component

The Product Details component shows detailed information about a single product, including its name, description, price, image, and other relevant details. It fetches the product data dynamically based on the product ID.

Steps:

1. **GROQ Query:**
   Fetch detailed product data (name, price, description, image) based on the product ID.
2. **Data Fetching:**
   Use useEffect to fetch the product details once the component mounts, based on the dynamic product ID.
3. **Display Product:**
   Render the product's name, image, description, price, and any other details.
4. **Add Interactivity:**
   Include "Add to Cart" or "Buy Now" buttons with relevant logic.

Home > Shop > Luxury Flower Bed



Luxury Flower Bed

Rs-2500

5 Customer Review

A luxurious shell-shaped chair with gold brass metal legs.

Size

L XL XS

Color

- 1 + Add To Cart

SKU : SS001

Category : Sofas

Tags : Sofa, Chair, Home, Shop

Share :

Activat
Go to Se

Code Snippet:

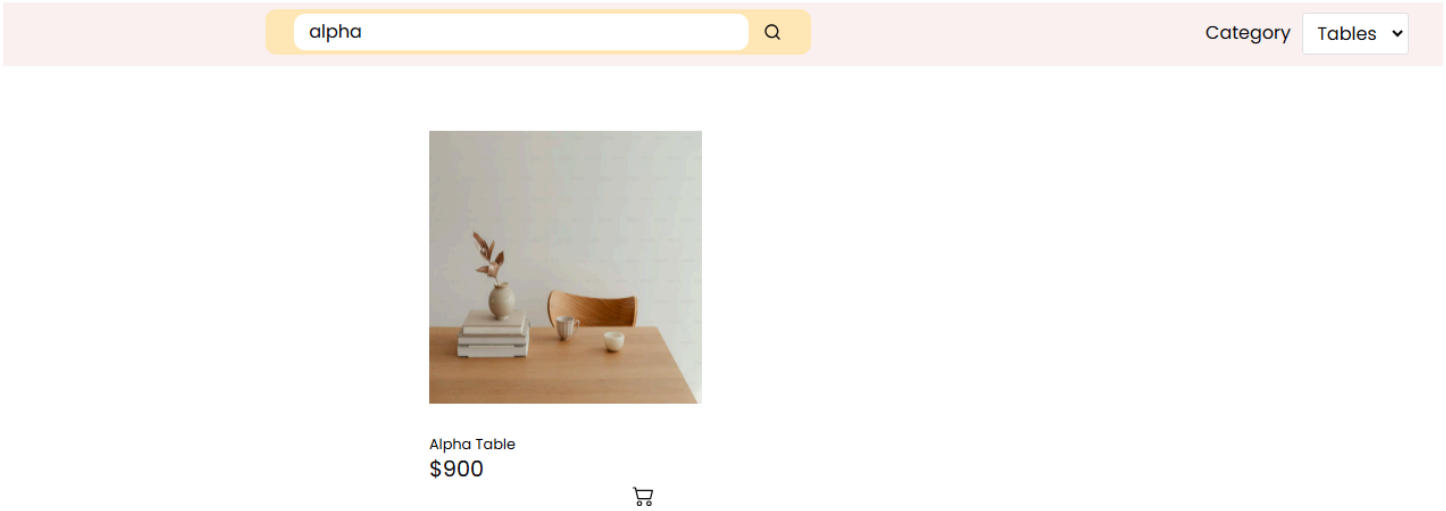
```tsx
export default function Singleproduct(props: Propstype) {
    const { toast } = useToast()
    const obj = useContext(CartContext)
    const wishlistobj = useContext(WishListContext)
    const [RelatedProducts, setRelatedProducts] = useState<product[]>([])
    const [Allproducts, setAllproducts] = useState<product[]>([])
    const [selectedProduct, setselectedProduct] = useState<product | null>(null)
    function heart() {···
    }

    useEffect(() => {
        const isInWishlist = wishlistobj.wishList.some((item: product) => item.id === props.prod.id);
        // Update heart state immediately for user feedback
        wishlistobj.settoggleHeartIcon(isInWishlist);
    }, [wishlistobj.wishList]);

    useEffect(() => {
        const fetchSelectedProducts = async () => {
            if (!props.prod?.id) {
                console.error("Product ID is undefined");
                return;
            }

            const query = `*[_type == 'product' && id == $id][0] {
                id,
                description,
                name,
                stockLevel,
                discountPercentage,
                price,
                isFeaturedProduct,
                category,
                "imagePath": imagePath.asset->url
            }`;

            try {
                const Rdata = await client.fetch(query, { id: props.prod.id }); // Using parameterized query
                setselectedProduct(Rdata);
```

Search Bar and Category Filters

The Search Bar and Category Filters allow users to quickly find products by searching for keywords or filtering by product category.

Steps:

1. **State Management:**
   Use useState to manage search input and selected category filter.
2. **Search Functionality:**
   Filter products by name based on the search query.
3. **Category Filter:**
   Filter products by category using a dropdown or buttons.
4. **Combine Filters:**
   Apply both search and category filters together to refine product listings.



Code Snippet:

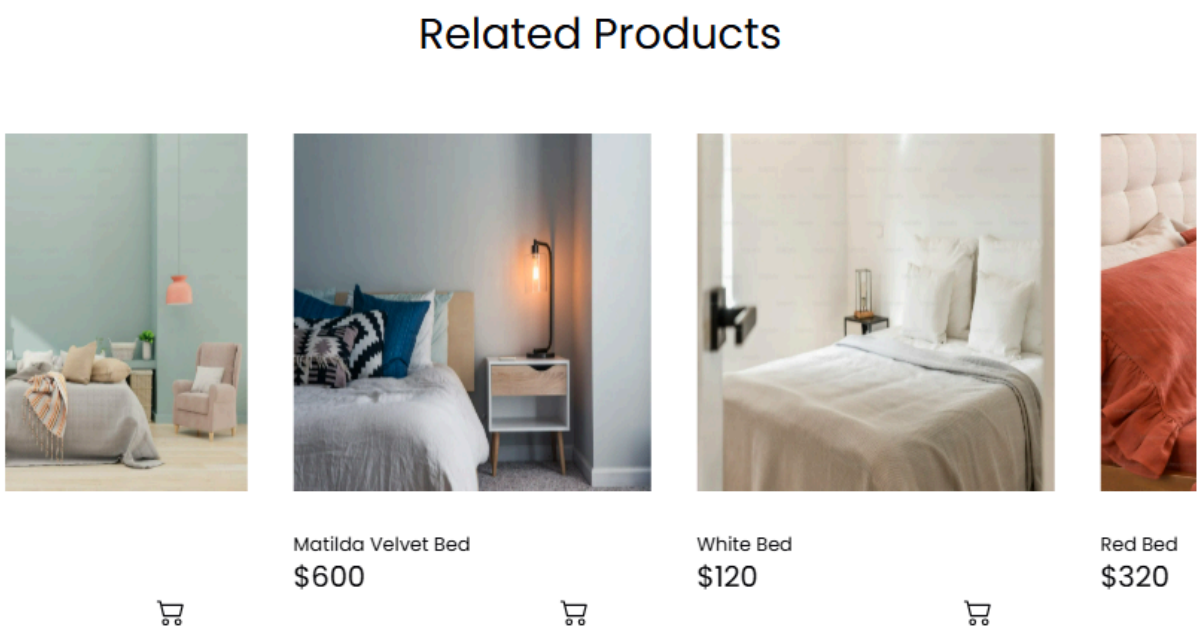
```
// state variable to store selected category value
const [Categoryvalue, setCaterogyvalue] = useState<string>('All');
const categoryHandling = (event: { target: { value: string } }) => {
    setCaterogyvalue(event.target.value)
}
// state variable to store searched value
const [typedSearch, settypedsearch] = useState<string>('');
const [Searchedvalue, setSearchedvalue] = useState<string>('');
const SearchHandling = (event: { target: { value: string } }) => {
    settypedsearch(event.target.value)

}
// To filter data
useEffect(() => {
    if (!Searchedvalue) {
        if (Categoryvalue === "All") {
            setProducts(allProducts.filter(item => item.price >= filterMinPrice && item.price <= filterMaxPrice))
        } else {
            setProducts(allProducts.filter(item => item.category.toLowerCase() === Categoryvalue.toLowerCase() && (item.price >=
            filterMinPrice && item.price <= filterMaxPrice)));
        }

    } else if (Searchedvalue) {
        if (Categoryvalue === "All") {
            setProducts(allProducts.filter(item =>
                (item.category.toLowerCase().includes(Searchedvalue.toLowerCase()) ||
                    item.name.toLowerCase().includes(Searchedvalue.toLowerCase()) ||
                    item.description.toLowerCase().includes(Searchedvalue.toLowerCase())) && (item.price >= filterMinPrice && item.pri
                    <= filterMaxPrice)
            ))
        } else {
            setProducts(allProducts.filter(item => item.category.toLowerCase() === Categoryvalue.toLowerCase() &&
                (item.category.toLowerCase().includes(Searchedvalue.toLowerCase()) ||
                    item.name.toLowerCase().includes(Searchedvalue.toLowerCase()) ||
                    item.description.toLowerCase().includes(Searchedvalue.toLowerCase())) && (item.price >= filterMinPrice && item.pri
                    <= filterMaxPrice)
            ));
        }
```


Related Products Component

The Related Products component displays products that are similar to the current product, typically based on category or other shared attributes, to encourage cross-selling.

Steps:

1. **Determine Similarity Criteria:**
   Use the product's category (or other attributes like tags) to filter related products.
2. **Data Fetching:**
   Fetch related products using useEffect when the current product is loaded.
3. **Display Related Products:**
   Render related product cards with images, names, and prices.
4. **Avoid Duplicates:**
   Ensure the current product is not included in the related products list.



Note

I have outlined the key and required components of my e-commerce marketplace, including Product Listing, Product Details, Search Bar & Category Filters, and Related Products. These components form the foundation of the user experience, providing dynamic product displays, interactive search, and efficient filtering.

For additional components and further insights into the architecture, you are welcome to explore the full project by visiting my website and GitHub repository.

Thank you for taking the time to review my work!