

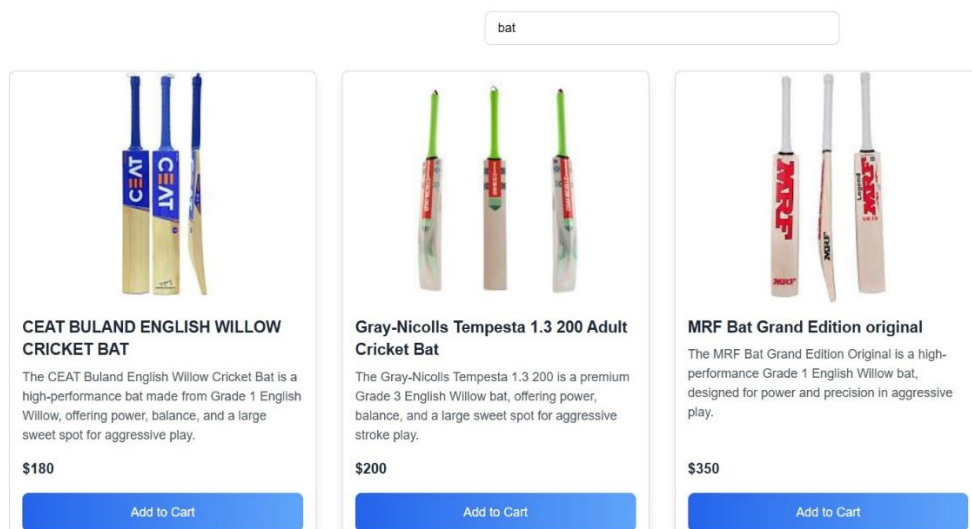
DAY 4 – DYNAMIC FRONEND COMPONENTS

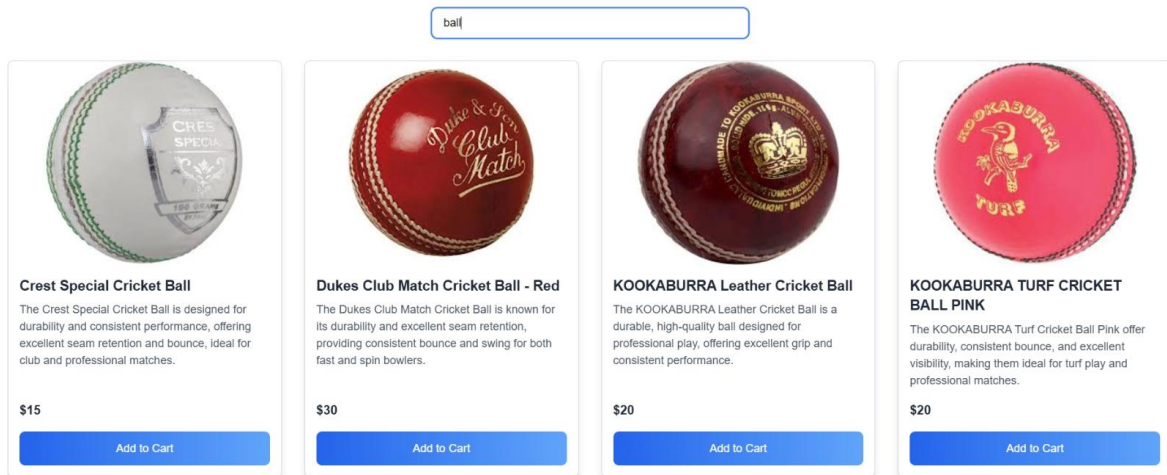
OVERVIEW : On Day 4, the goal is to build dynamic frontend components that interact with your backend (Sanity) to fetch, display, and manipulate data in real-time. These components will enhance user experience by providing interactive and data-driven content, such as product lists, a search bar, and shopping cart functionality. You'll integrate dynamic data fetching from Sanity, allowing your components to respond to changes in the content stored in your CMS.

KEY FUNCTIONALITIES

1. Search Bar

- **Purpose:** The search bar allows users to filter and search for products or other content dynamically. It interacts with the backend (Sanity) to fetch results based on the user's input.
- **How It Works:**
 - As the user types, a query is sent to Sanity to search for products (e.g., by name, category, or tags).
 - **Real-time updates:** The product list updates instantly as the user types in the search bar.
 - **Dynamic filtering:** You can filter based on specific attributes, such as product name or price range.





2. Dynamic Product Data

- **Purpose:** Fetch and display product details dynamically. The products could be anything like cricket bats, shoes, etc., fetched from Sanity.
- **How It Works:**
 - **Data Fetching:** Use Sanity's GROQ queries to fetch product data (name, description, image, price, etc.).
 - **Rendering:** Dynamically render products in cards or lists using React components. As data updates in the CMS, the frontend automatically reflects the new data without the need for manual code changes.
 - **Updates:** When new products are added or edited in Sanity, they will appear automatically on the frontend when the page reloads or the query is refreshed.

```
client
  .fetch(
    `*[_type == "product"]{
      _id,
      title,
      price,
      description,
      "imageUrl": image.asset->url
    }`
  )
  .then((data) => {
    const sortedProducts = data.sort((a: Product, b: Product) =>
      a.title.localeCompare(b.title)
    );
    setProducts(sortedProducts);
  })
  .catch((err) => {
    console.error("Error fetching products:", err);
  });
```

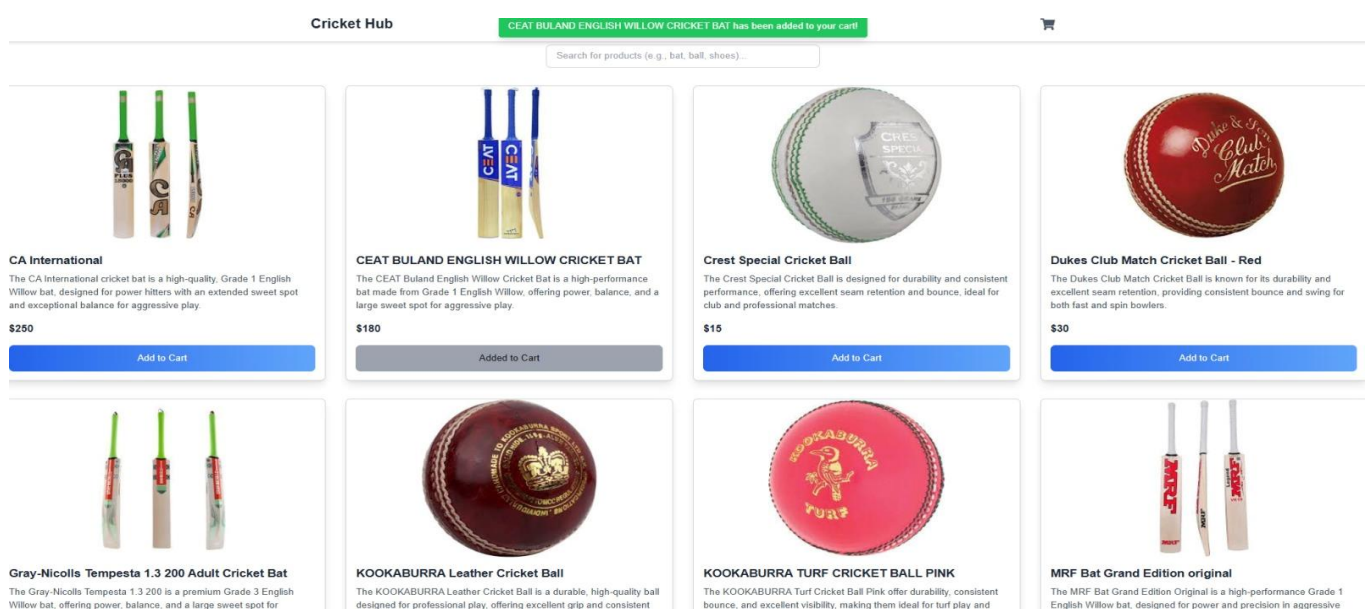
```


<div className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-4 gap-8 px-4">
  {filteredProducts.length > 0 ? (
    filteredProducts.map((product) => {
      <div
        key={product._id}
        className="bg-white border border-gray-300 rounded-lg shadow-lg overflow-hidden flex flex-col"
      >
        <img
          src={product.imageUrl}
          alt={product.title}
          className="w-full h-72 object-contain"
        />
        <div className="flex flex-col justify-between p-4 h-full">
          <h2 className="text-xl font-semibold text-gray-800 mb-2">
            {product.title}
          </h2>
          <p className="text-gray-600 mb-4 flex-grow">
            {product.description}
          </p>
          <p className="text-lg font-semibold text-gray-800 mb-4">
            ${product.price}
          </p>

```

3. Add to Cart Functionality

- **Purpose:** This feature allows users to add selected products to a cart for purchasing. It involves managing a cart state and handling user interactions.
- **How It Works:**
 - Each product card includes an Add to Cart button. When clicked, the corresponding product is added to the cart.
 - The cart's contents are stored in local storage to persist even if the user refreshes the page.
 - The cart can be viewed at any time, showing a list of selected items and their details (e.g., quantity, price).
 - **Checkout functionality:** Once items are added to the cart, users can proceed to checkout.



Your Shopping Cart					
Image	Product	Price (per item)	Quantity	Total Price	Action
	CEAT BULAND ENGLISH WILLOW CRICKET BAT	\$180	- 1 +	\$180.00	<button>Remove</button>
<button>Proceed to Checkout</button>					

The Secret Sauce of Dynamic Frontend Design

- **State Management:** Handling dynamic data, like the cart or product list, requires **state management**. In React, this is often done using `useState` or more complex state management solutions like `Redux` or `Context API` for larger applications.
- **Interactivity and Responsiveness:** Dynamic components also depend on how well the application responds to user actions. Using tools like `React Hooks` (`useEffect`, `useState`) allows the app to re-render components efficiently when data changes.
- **User Feedback:** Providing visual feedback (e.g., loading spinners, success messages) can improve the user experience, especially during data fetching or when adding items to the cart.

CONCLUSION : In conclusion, dynamic frontend components are the heart of creating engaging, responsive, and user-centric web applications. By leveraging state management, real-time data fetching, and performance optimization techniques, you can build intuitive and fast interfaces that adapt to user input and data changes seamlessly. Whether it's through a search bar that updates instantly or a shopping cart that dynamically reflects user selections, these components provide a fluid experience that keeps users engaged. With the right balance of performance and interactivity, dynamic components turn static websites into living, breathing applications, enhancing both functionality and user satisfaction. As you continue to integrate these components into your projects, you'll unlock the full potential of a responsive, modern web experience.