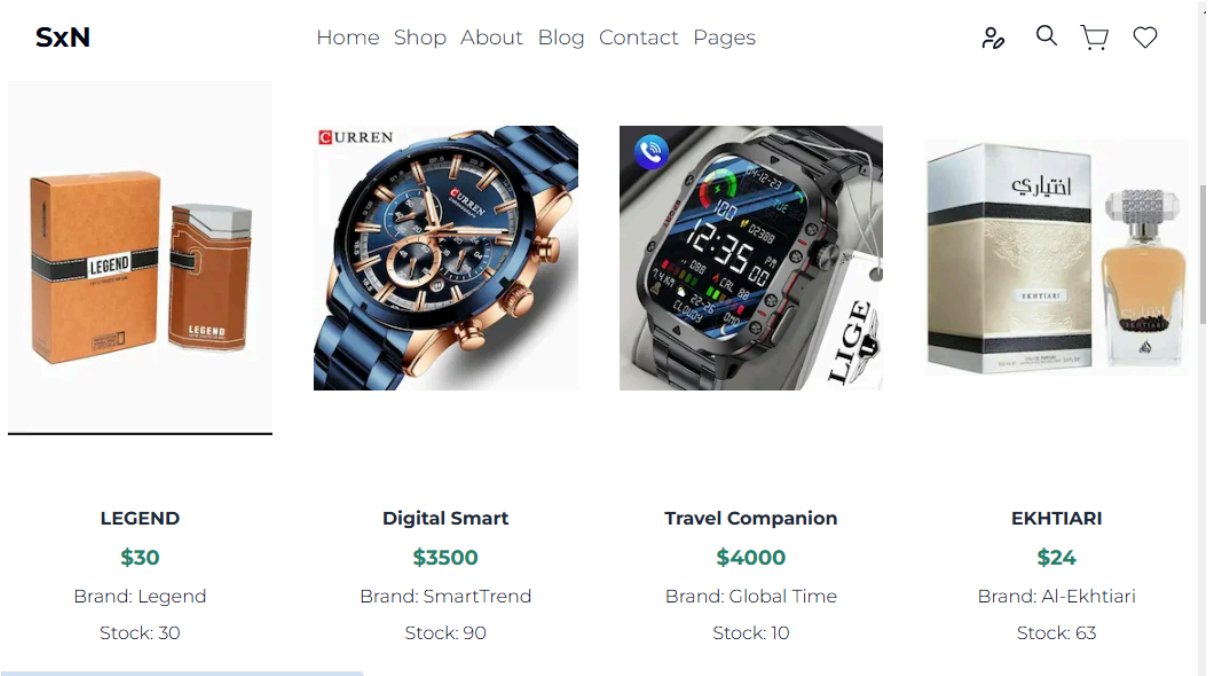
Day 4 - Dynamic Frontend Components SxN By NASH

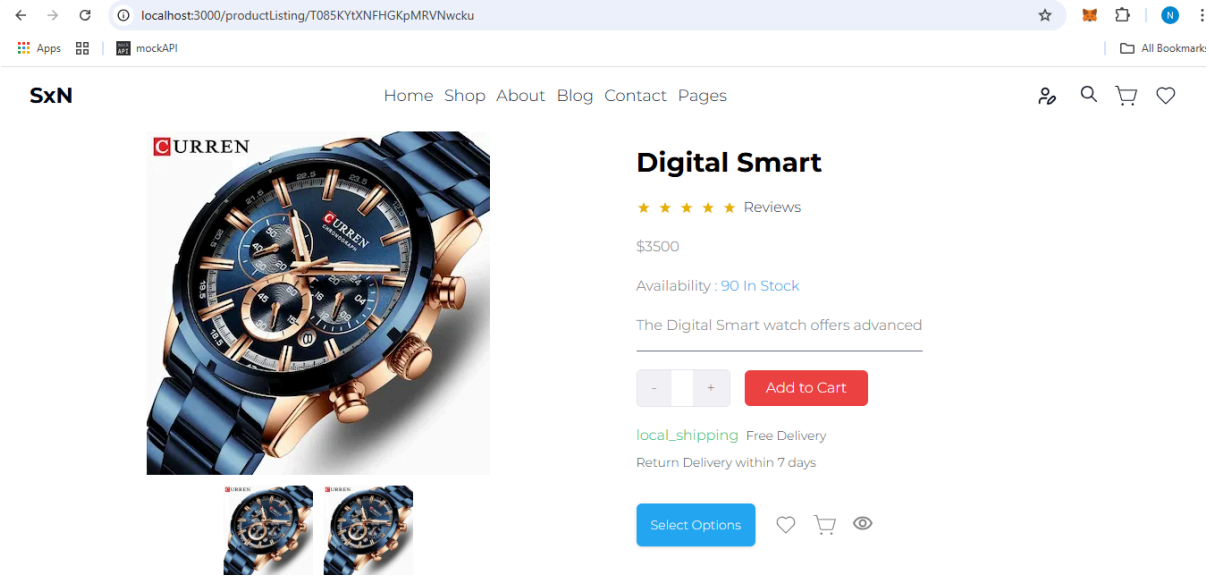
1. Functional Deliverables:

Screenshots or screen recordings showcasing:

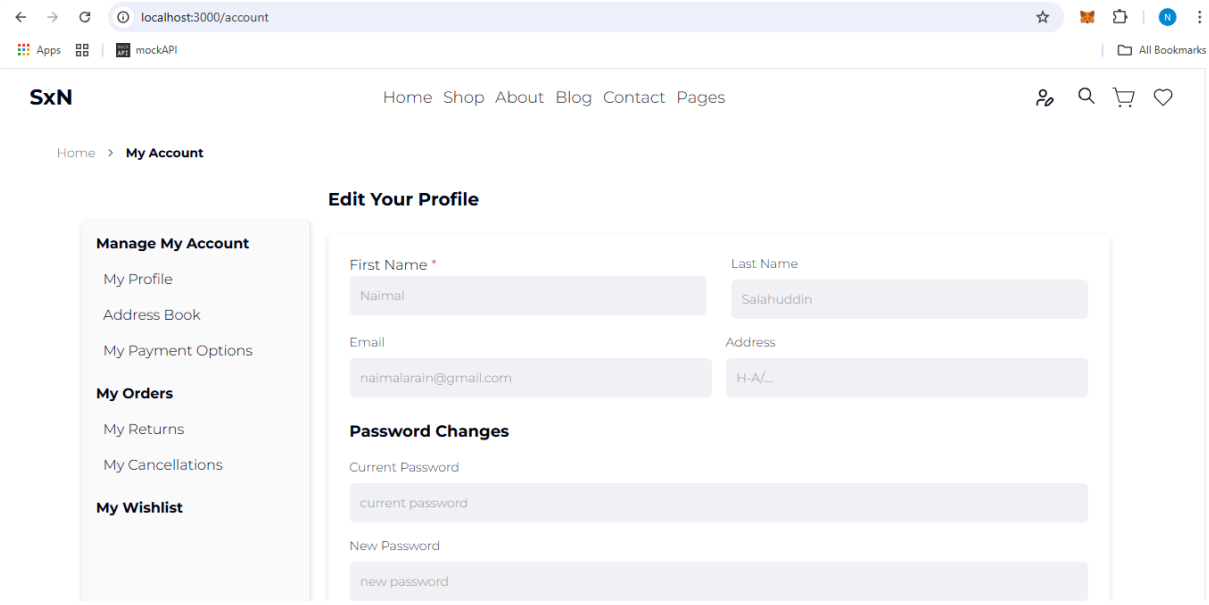
The product listing page with dynamic data.

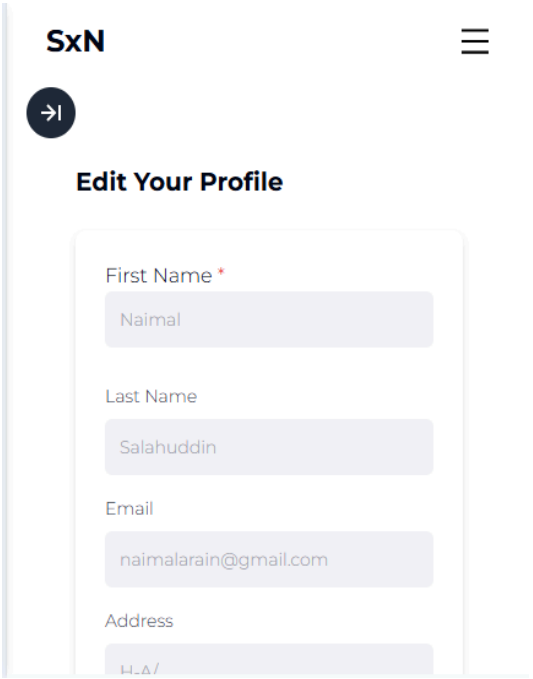


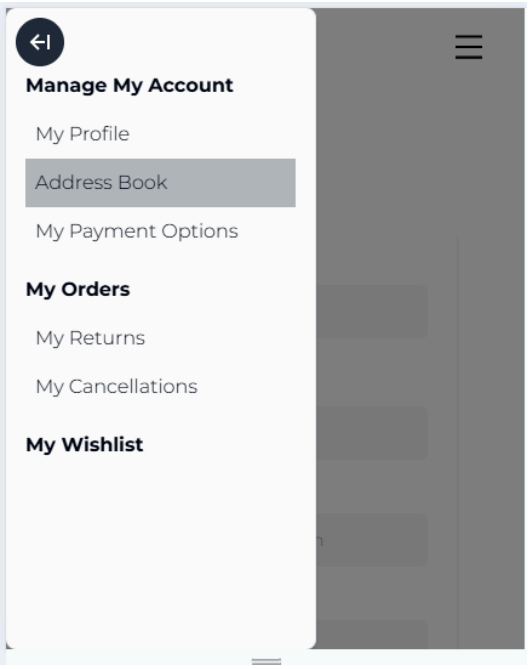
Individual product detail pages with accurate routing and data rendering.



Any additional features implemented, such as related products or user profile components.





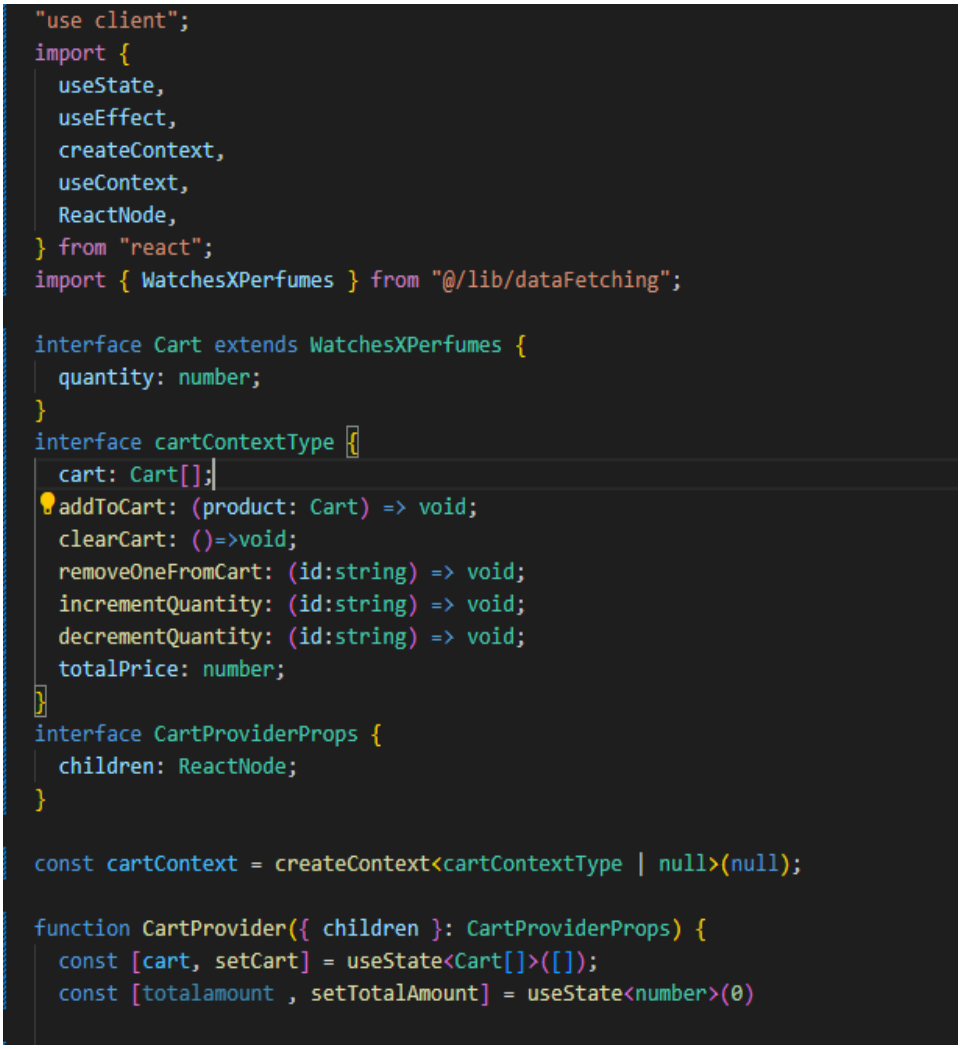


2. Code Deliverables:

Code snippets for key components (e.g., ProductCard, ProductList). Scripts or logic for API integration and dynamic routing.

```tsx
"use client";

import React from "react";
import Image from "next/image";
import { WatchesXPerfumes } from "@/lib/dataFetching";

export default function JustForYou({
  imageUrl,
  name,
  description,
  price,
  priceWithoutDiscount,
  stock_Quantity,
  brand,
}: WatchesXPerfumes) {
  if (
    imageUrl &&
    name &&
    description &&
    price &&
    priceWithoutDiscount &&
    stock_Quantity &&
    brand
  ) {
    return (
      <div className="w-full bg-white overflow-hidden">
        {/* Product Image */}
        <div className="w-full h-[400px] flex justify-center items-center">
          <Image
            src={imageUrl}
            alt={name}
            width={238}
            height={427}
            className="w-full p-2 object-cover"
          />
        </div>

        {/* Product Details */}
        <div className="p-4 flex flex-col gap-2 justify-center items-center">
          {/* Product Title */}
          <h3 className="text-base font-semibold text-gray-800 truncate">
            {name}
          </h3>

          {/* Product Price */}
          <div className="flex gap-2 items-center">
            <p className="text-lg text-[#23856D] font-bold">${price}</p>
          </div>
          <p>Brand: {brand}</p>
          <p>Stock: {stock_Quantity}</p>
        </div>
      </div>
    );
  }
}
```

This Product Card is used all over the website



```tsx
"use client";
import {
  useState,
  useEffect,
  createContext,
  useContext,
  ReactNode,
} from "react";
import { WatchesXPerfumes } from "@/lib/dataFetching";

interface Cart extends WatchesXPerfumes {
  quantity: number;
}
interface cartContextType {
  cart: Cart[];
  addToCart: (product: Cart) => void;
  clearCart: ()=>void;
  removeOneFromCart: (id:string) => void;
  incrementQuantity: (id:string) => void;
  decrementQuantity: (id:string) => void;
  totalPrice: number;
}
interface CartProviderProps {
  children: ReactNode;
}

const cartContext = createContext<cartContextType | null>(null);

function CartProvider({ children }: CartProviderProps) {
  const [cart, setCart] = useState<Cart[]>([]);
  const [totalamount , setTotalAmount] = useState<number>(0)
```

This Context manages the state globally

Day 3 Task Documentation

Overview

On Day 3 of the project, significant progress was made by implementing several dynamic frontend components and integrating the Context API to manage the "Add to Cart" functionality. The goal was to enhance the user interface by introducing interactive elements such as product listings, user accounts, and a functional cart system while ensuring smooth state management across the application.

This document provides a detailed breakdown of the implemented features, challenges faced during the process, solutions applied, and the best practices followed.

Dynamic Frontend Components

The following dynamic frontend components were added to the application on Day 3:

1. Product Listing

A component was added to display a list of products available for purchase. This component fetches dynamic data from a backend or mock data source and renders it in a structured layout, allowing users to view multiple products at once.

2. Single Product Detail

The single product detail page was implemented to provide a more in-depth view of individual products. This includes displaying product information, images, pricing, and a quantity selector, which is essential for adding items to the cart.

3. Cart

A dynamic cart component was developed to track items selected for purchase. Users can view the cart contents, adjust quantities, and proceed to checkout. This component provides real-time updates whenever items are added, removed, or modified.

4. Wishlist

The wishlist component allows users to save their favorite products for future reference. This component provides users with the ability to add or remove products from their wishlist without affecting the cart contents.

5. Checkout

The checkout component facilitates the completion of a purchase. It allows users to review their cart, enter shipping information, and finalize the transaction. This component interacts with the cart and user account systems.

6. User Account

A user account component was introduced to allow users to register, log in, and manage their profile details. It includes functionality for users to view their order history and manage preferences.

7. Header and Footer

The header and footer components were added to provide consistent navigation across the website. The header includes elements like the logo, navigation links, and cart icon, while the footer displays relevant information such as links to privacy policies and contact details.

8. Top Header

A top header was added to display promotional banners or notifications at the top of the page, which can be dynamically updated based on marketing campaigns or site-wide announcements.

9. Profile Sidebar Toggle

A profile sidebar toggle was integrated to offer users easy access to their profile settings. This feature enhances the user experience by providing an accessible and responsive way to navigate through account-related settings.

Integration of Context API for Cart Management

The Context API was introduced to efficiently manage the "Add to Cart" functionality and ensure that the cart's state is accessible throughout the entire website. The cart functionality allows users to:

- **Add Items to Cart**: Users can add products to their cart with specified quantities.
- **Remove Items from Cart**: Users can remove items from the cart if they no longer wish to purchase them.
- **Increase Item Quantity**: Users can increase the quantity of a specific item in the cart.
- **Decrease Item Quantity**: Users can reduce the quantity of a specific item in the cart.
- **Clear the Cart**: Users can clear all items from the cart at once.
- **Calculate Total Amount**: The system automatically calculates and updates the total cost of the items in the cart, including any applicable discounts or taxes.

Cart Provider in `layout.tsx`

The Cart Provider was implemented in `layout.tsx` to wrap the children component and provide the entire website with access to the context. By wrapping the app with `CartProvider`, every component has the ability to read from and modify the cart state, ensuring synchronization between different parts of the application.

Displaying Dynamic Data on the Frontend

The dynamic data for products, cart contents, and user information are now displayed dynamically on the frontend. Whenever the cart state changes (e.g., adding or removing an item), the components reflect these updates in real-time without requiring a page reload. This creates a seamless user experience and ensures that the cart data remains up-to-date across the site.

Challenges and Solutions

Challenge 1: Handling Complex Cart States

Managing the cart state across multiple components, especially with dynamic updates (adding, removing, and modifying items), posed a significant challenge. It was necessary to ensure that the cart state was consistent and updated correctly.

Solution: The Context API was implemented to manage the cart state globally. By using the `CartProvider`, all components that needed access to the cart could consume and update the state without causing inconsistencies.

Challenge 2: Synchronization Between Components

With the addition of several dynamic components (such as the product listing, cart, and wishlist), ensuring data synchronization across these components required careful planning. For instance, adding an item to the cart needed to trigger an update in both the cart view and the wishlist.

Solution: Context API's state management allowed all components to access and react to changes in the cart state. This ensured synchronization across the UI, making it easier to maintain consistent data and user actions.

Challenge 3: Dynamic UI Rendering

Rendering dynamic data and ensuring that the frontend UI reflected state changes (e.g., updated product quantities or cart totals) required efficient and responsive UI updates.

Solution: React's state management features, combined with the Context API, enabled real-time updates to the UI. Components were designed to rerender when necessary, ensuring that the UI reflected the latest data without manual intervention.

Best Practices Followed

1. Component Modularity

Each feature, such as the product listing, cart, and user profile, was implemented as a separate, reusable component. This modular approach ensures maintainability and scalability, allowing components to be easily modified or extended in the future.

2. Centralized State Management

The Context API was utilized to manage the cart state globally. This approach ensures that state is centrally managed, making it easier to debug, test, and update. The use of Context API also reduces the need for prop drilling and simplifies component interactions.

3. Responsive Design

All components were built with responsiveness in mind, ensuring that they adapt well to different screen sizes and devices. This was achieved by using CSS flexbox/grid layouts and media queries.

4. Code Reusability

Reusable functions and hooks were created where possible, particularly for handling cart operations like adding and removing items. This approach prevents code duplication and improves maintainability.

5. Error Handling and Validation

Basic error handling and validation were added to components that deal with user inputs, such as the checkout form and user account management. This ensures that invalid data is caught early, preventing issues during transactions.

Conclusion

Day 3's implementation focused on adding dynamic frontend components and integrating a robust cart management system using the Context API. The key features—product listings, user accounts, and cart management—work seamlessly together to provide an enhanced shopping experience. Despite challenges related to state management and UI synchronization, the solutions applied resulted in a stable and responsive application. The development followed best practices, ensuring the project is scalable, maintainable, and user-friendly.