Day-04: Building and Integrating Components for an E-Commerce Website

Overview:

This report outlines the Day 4 about building and integrating components for an e-commerce websites

First here comes the ,

Product Details:

The `Product detail` component is designed to display detailed information about a specific product. It retrieves product data from a Sanity CMS and displays attributes like:

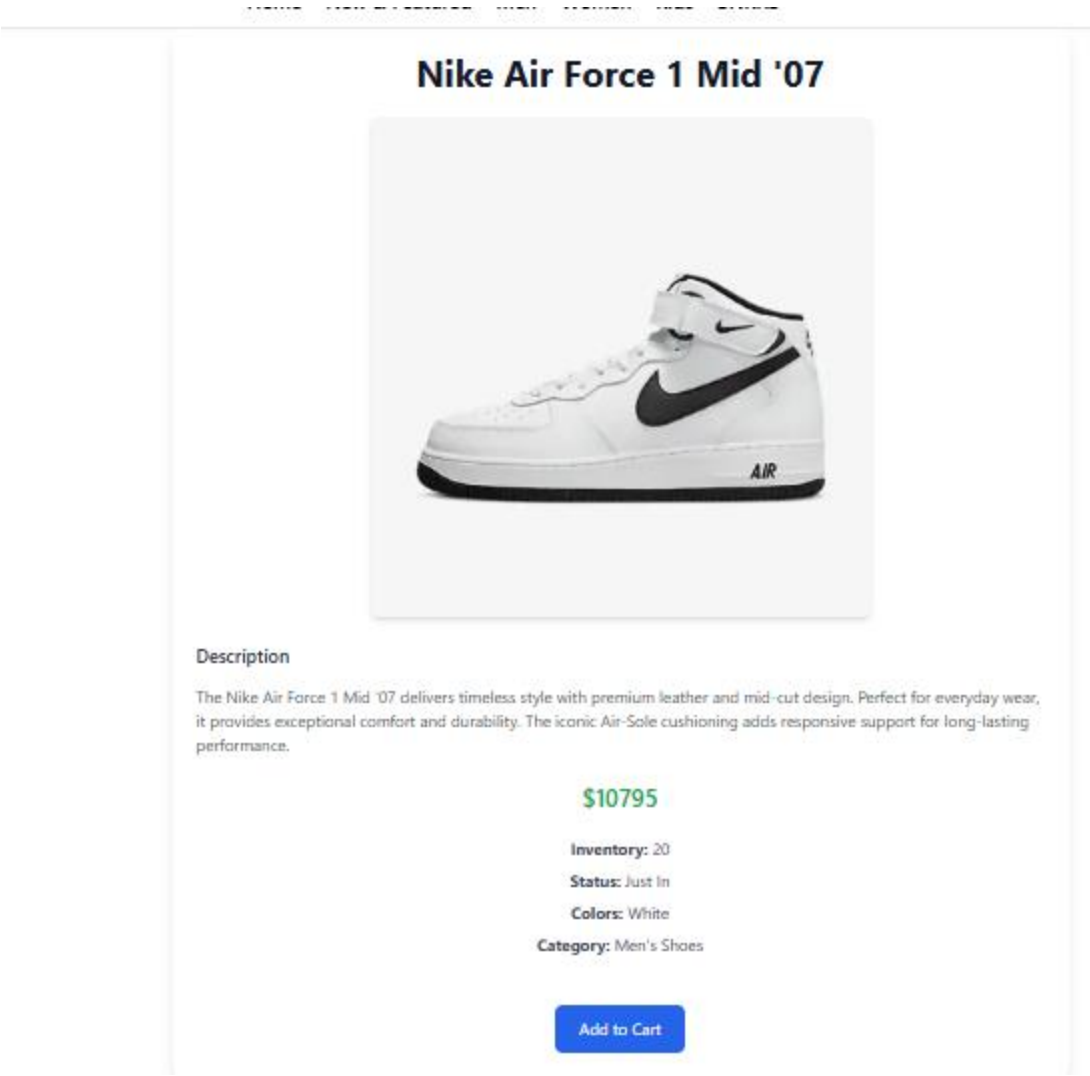
- Name
- Price
- Description
- Image
- Inventory status
- Available colors
- Category
- Other relevant product details.

The component also handles adding the product to the user's shopping cart, which is stored in the browser's `localStorage`.

Component Overview

The component uses the following libraries:

1. **React**: Handles state and side effects.
2. **Next.js**: Provides server-side rendering, routing, and dynamic parameters (`slug`).
3. **Sanity Client**: Fetches product data from the Sanity CMS backend.
4. **Tailwind CSS**: For styling.



Expected Results

1. **Loading State**:
   - If the `slug` is being resolved or product data is being fetched, a loading message will be shown.
2. **Error State**:
   - If an error occurs during the fetch (e.g., product not found or network issues), the error message is displayed, and the user can retry by clicking the "Retry" button.
3. **Product Display**:

- Once the product is successfully fetched, its details are displayed, including its name, description, price, inventory, status, available colors, and category.

4. **Add to Cart**:
   - When the user clicks the "Add to Cart" button, the product is added to the shopping cart stored in the `localStorage`. If the product is already in the cart, its quantity will be increased by 1.

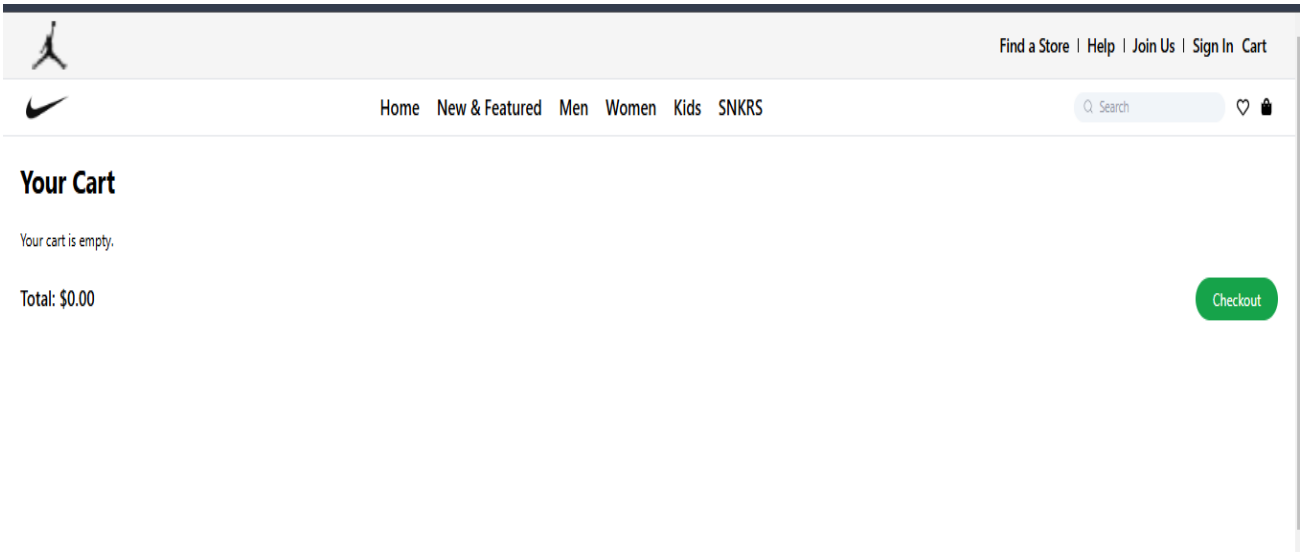
Cart

Overview:

The `CartPage` component displays the contents of a user's shopping cart, provides functionality to update the cart, and allows users to proceed to checkout. It is implemented using React hooks (`useState` and `useEffect`) and stores the cart data in the browser's `localStorage` for persistence across page reloads.

Features:

1. **Display Cart Items**: The cart contents are retrieved from `localStorage` and displayed in a list.
2. **Add Products to Cart**: Products can be added to the cart, updating the quantity if the product already exists in the cart.
3. **Remove Products from Cart**: Individual products can be removed from the cart.
4. **Checkout Button**: The cart page provides a button to navigate to the checkout page.
5. **Persistent Cart**: The cart persists across page reloads by storing cart data in `localStorage`.



Usage:

1. **Cart Persistence**: The cart items are stored in the browser's `localStorage`, which means the cart will persist across page reloads or browser restarts.
2. **Adding and Removing Products**: Products can be added or removed from the cart by interacting with the respective buttons.
3. **Checkout**: When the user clicks the **Checkout** button, they are redirected to the checkout page (`/checkOut`)

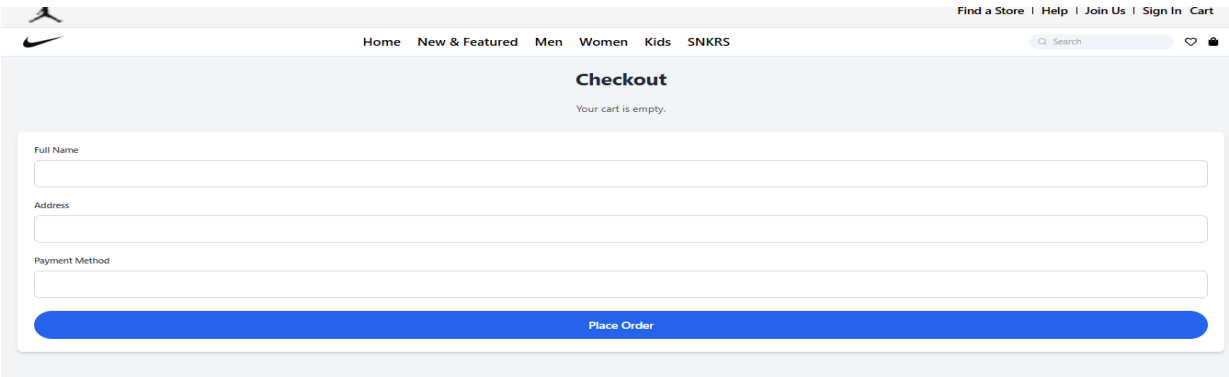
Checkout

The `CheckoutPage` component is a part of the checkout process in a shopping cart flow. It allows users to review their order, input their details (name, address, payment method), and submit the order. The page also includes form validation, success/error alerts, and cart persistence via `localStorage`.

Features:

1. **Cart Summary**: Displays a summary of the cart with product details, including name, quantity, price, and total amount.
2. **Form for User Details**: Users are prompted to enter their name, address, and payment method.
3. **Alert Messages**: Success and error messages appear based on form validation and submission.
4. **Checkout Button**: A button that submits the order and clears the cart.
5. **Persistent Cart**: The cart persists using `localStorage` so that items remain even after page reloads.

Responsive Design: Uses Tailwind CSS for styling, making it adaptable to various screen sizes

6. 

Next Steps:

1. **Form Validation**: The form currently only checks if all fields are filled out. You can enhance validation by adding more checks, such as validating the payment method or address format.
2. **Handle Payment**: You can implement an actual payment gateway (e.g., Stripe, PayPal) for processing payments.
3. **API Integration**: Instead of logging to the console, integrate with an API to submit the order details to a server.
4. **Error Handling**: Add more robust error handling, e.g., for network issues or failed API requests.

Let me know if you need any additional functionality or help with the next steps!

The End